# Alibaba Cloud
# ApsaraDB for PolarDB

## Developer Guide for PolarDB-O

Issue: 20200701

# Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.
2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company, or individual in any form or by any means without the prior written consent of Alibaba Cloud.
3. The content of this document may be changed due to product version upgrades, adjustments, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and the updated versions of this document will be occasionally released through Alibaba Cloud-authorized channels. You shall pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.
4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides the document in the context that Alibaba Cloud products and services are provided on an "as is", "with all faults" and "as available" basis. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not bear any liability for any errors or financial losses incurred by any organizations, companies, or individual s arising from their download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, bear responsibility for any indirect, consequential, exemplary , incidental, special, or punitive damages, including lost profits arising from the use or trust in this document, even if Alibaba Cloud has been notified of the possibility of such a loss.

5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.
6. Please contact Alibaba Cloud directly if you discover any errors in this document.

# Document conventions

| Style | Description | Example |
|---|---|---|
| ⛔ | A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results. | ⛔ **Danger:**<br>Resetting will result in the loss of user configuration data. |
| ⚠ | A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results. | ⚠ **Warning:**<br>Restarting will cause business interruption. About 10 minutes are required to restart an instance. |
| ⓘ | A caution notice indicates warning information, supplementary instructions, and other content that the user must understand. | ⓘ **Notice:**<br>If the weight is set to 0, the server no longer receives new requests. |
| 📋 | A note indicates supplemental instructions, best practices, tips, and other content. | 📋 **Note:**<br>You can use Ctrl + A to select all files. |
| > | Closing angle brackets are used to indicate a multi-level menu cascade. | Click **Settings** > **Network** > **Set network type**. |
| **Bold** | Bold formatting is used for buttons , menus, page names, and other UI elements. | Click **OK**. |
| `Courier font` | Courier font is used for commands. | Run the `cd /d C:/window` command to enter the Windows system folder. |
| Italic | Italic formatting is used for parameters and variables. | `bae log list --instanceid`<br>Instance_ID |
| [] or [a\|b] | This format is used for an optional value, where only one item can be selected. | `ipconfig [-all\|-t]` |

| Style | Description | Example |
|---|---|---|
| {} or {a\|b} | This format is used for a required value, where only one item can be selected. | `switch {active|stand}` |

# Contents

# 1 Oracle compatibility

This topic introduces the features supported by the PolarDB database engine that is compatible with Oracle syntax.

**Note:**

This topic lists only the common features.

PolarDB is highly compatible with Oracle. The following table lists common features.

| Type | Sub-type | Compatibility |
|---|---|---|
| Partitioned table | PARTITION BY RANGE | Compatible |
| | PARTITION BY HASH | Compatible |
| | PARTITION BY LIST | Compatible |
| | SUB-PARTITIONING | Compatible |
| Data type | NUMBER | Compatible |
| | VARCHAR2 , NVARCHAR2 | Compatible |
| | CLOB | Compatible |
| | BLOB | Compatible |
| | RAW | Compatible |
| | LONG RAW | Compatible |
| | DATE | Compatible |
| SQL syntax | HIERARCHICAL QUERIES | Compatible |
| | SYNONYMS (PUBLIC AND PRIVATE) | Compatible |
| | SEQUENCE GENERATOR | Compatible |
| | HINT | Compatible |
| Function | The number of supported functions | 3155 |
| | DUAL | Compatible |
| | DECODE | Compatible |
| | ROWNUM | Compatible |
| | SYSDATE | Compatible |
| | SYSTIMESTAMP | Compatible |
| | NVL | Compatible |

| Type | Sub-type | Compatibility |
|---|---|---|
| | NVL2 | Compatible |
| Security | DATA REDACTION | Compatible |
| | Database Firewall Only (SQL/Protect) | Compatible |
| | VPD | Compatible |
| | PL/SQL code encryption | Compatible |
| | PROFILES FOR PASSWORDS | Compatible |
| PL/SQL | PL/SQL Compatible | Compatible |
| | NAMED PARAMETER NOTATION FOR STORED PROCEDURES | Compatible |
| | TRIGGERS | Compatible |
| | REF CURSORS | Compatible |
| | IMPLICIT / EXPLICIT CURSORS | Compatible |
| | ANONYMOUS BLOCKS | Compatible |
| | BULK COLLECT/BIND | Compatible |
| | ASSOCIATIVE ARRAYS | Compatible |
| | NESTED TABLES | Compatible |
| | VARRAYS | Compatible |
| | PL/SQL SUPPLIED PACKAGES | Compatible |
| | PRAGMA RESTRICT_REFERENCES | Compatible |
| | PRAGMA EXCEPTION_INIT | Compatible |
| | PRAGMA AUTONOMOUS_TRANSACTION | Compatible |
| | USER DEFINED EXCEPTIONS | Compatible |
| | OBJECT TYPES | Compatible |
| | SUB-TYPES | Compatible |
| Package | The number of supported packages | 26 |
| | Built-in functions | 317 |
| Advanced feature | DATABASE LINKS | Compatible |
| | AWR | Compatible |
| | SQL profile | Compatible |
| | Index recommendation | Compatible |

| Type | Sub-type | Compatibility |
|---|---|---|
| | CPU and memory resource isolation by user | Compatible |
| | TUNING PACKAGE | Compatible |
| System view | The number of system views | 88 |
| Embedded C programming | Pro*C | Compatible |
| Client driver | OCI | Compatible |

### References

Oracle-compatible operations

# 2 Connect to a POLARDB cluster compatible with Oracle

In addition to connecting to a POLARDB cluster compatible with Oracle through the ApsaraDB for POLARDB console, you can also connect to the cluster through the pgAdmin 4 client. This topic describes how to use the pgAdmin 4 client to connect to a POLARDB cluster compatible with Oracle.

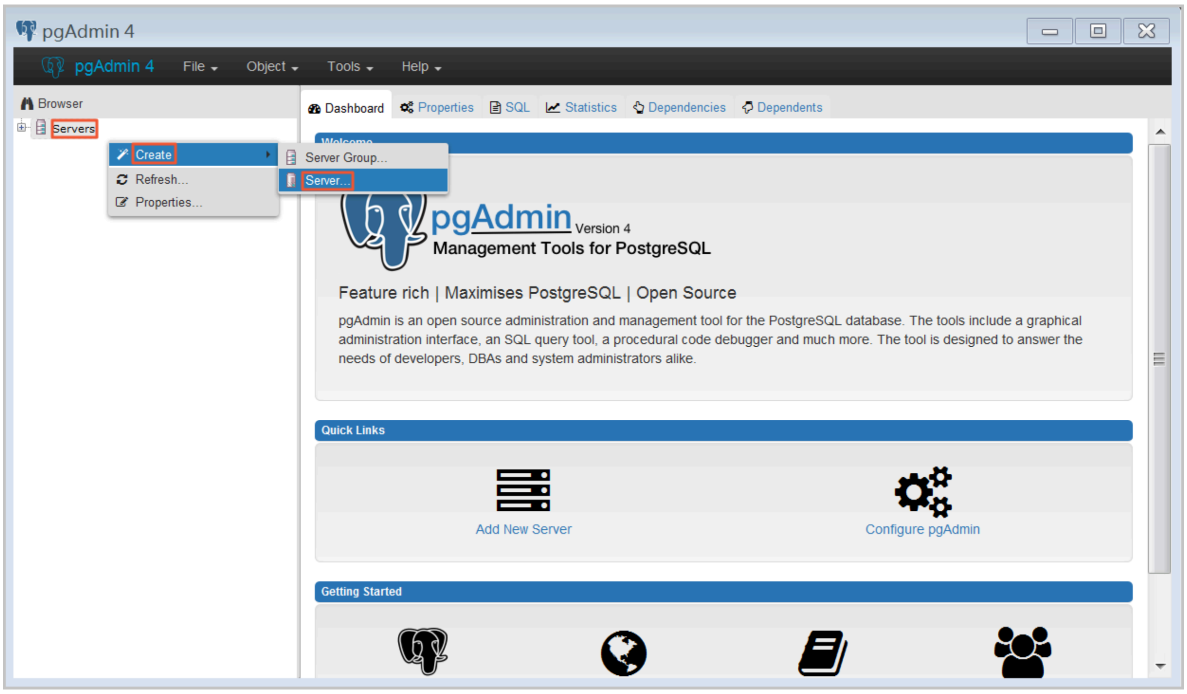
## Prerequisites

- You have created a privileged or standard account for an existing database cluster. For more information, see #unique_6.
- You have installed pgAdmin 4 in a server that can connect to POLARDB clusters compatible with Oracle such as ECS.
- POLARDB compatible with Oracle only provides the private endpoint. You must connect to the POLARDB cluster compatible with Oracle by using an ECS instance that is in the same VPC.
- You must use a Windows-based ECS instance.

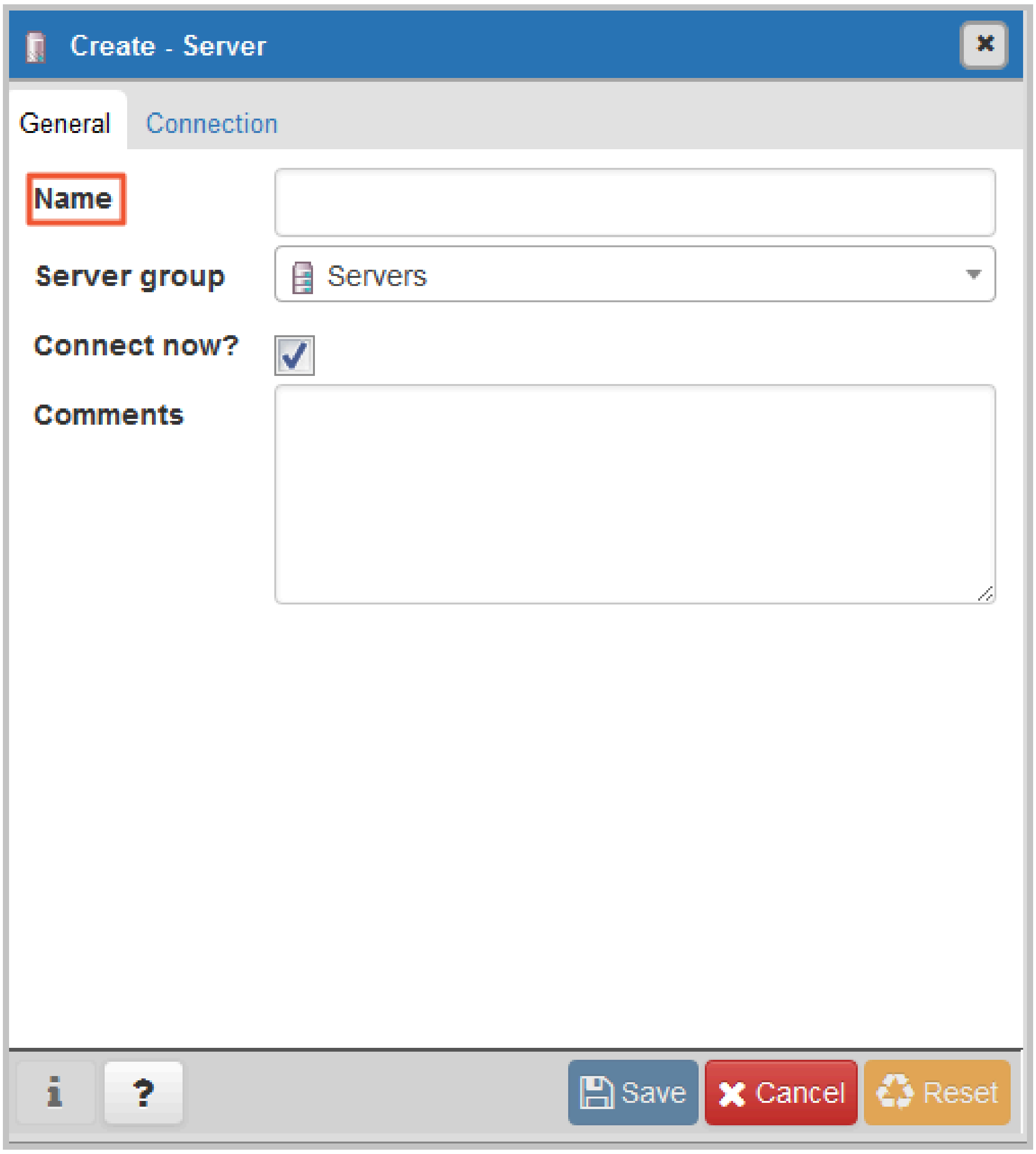
## Procedure

1. Start the pgAdmin 4 client.

2. Right-click **Servers** and choose **Create** > **Server** from the shortcut menu, as shown in the following figure.


pgAdmin 4
pgAdmin 4
File
Object
Tools
Help
Browser
Servers
Create
Refresh...
Properties...
Server Group...
Server...
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Welcome
pgAdmin Version 4
Management Tools for PostgreSQL
Feature rich | Maximises PostgreSQL | Open Source
pgAdmin is an open source administration and management tool for the PostgreSQL database. The tools include a graphical administration interface, an SQL query tool, a procedural code debugger and much more. The tool is designed to answer the needs of developers, DBAs and system administrators alike.
Quick Links
Add New Server
Configure pgAdmin
Getting Started

3. On the **General** tab of the **Create - Server** dialog box, enter the name of the server, as shown in the following figure.



Create - Server

General Connection

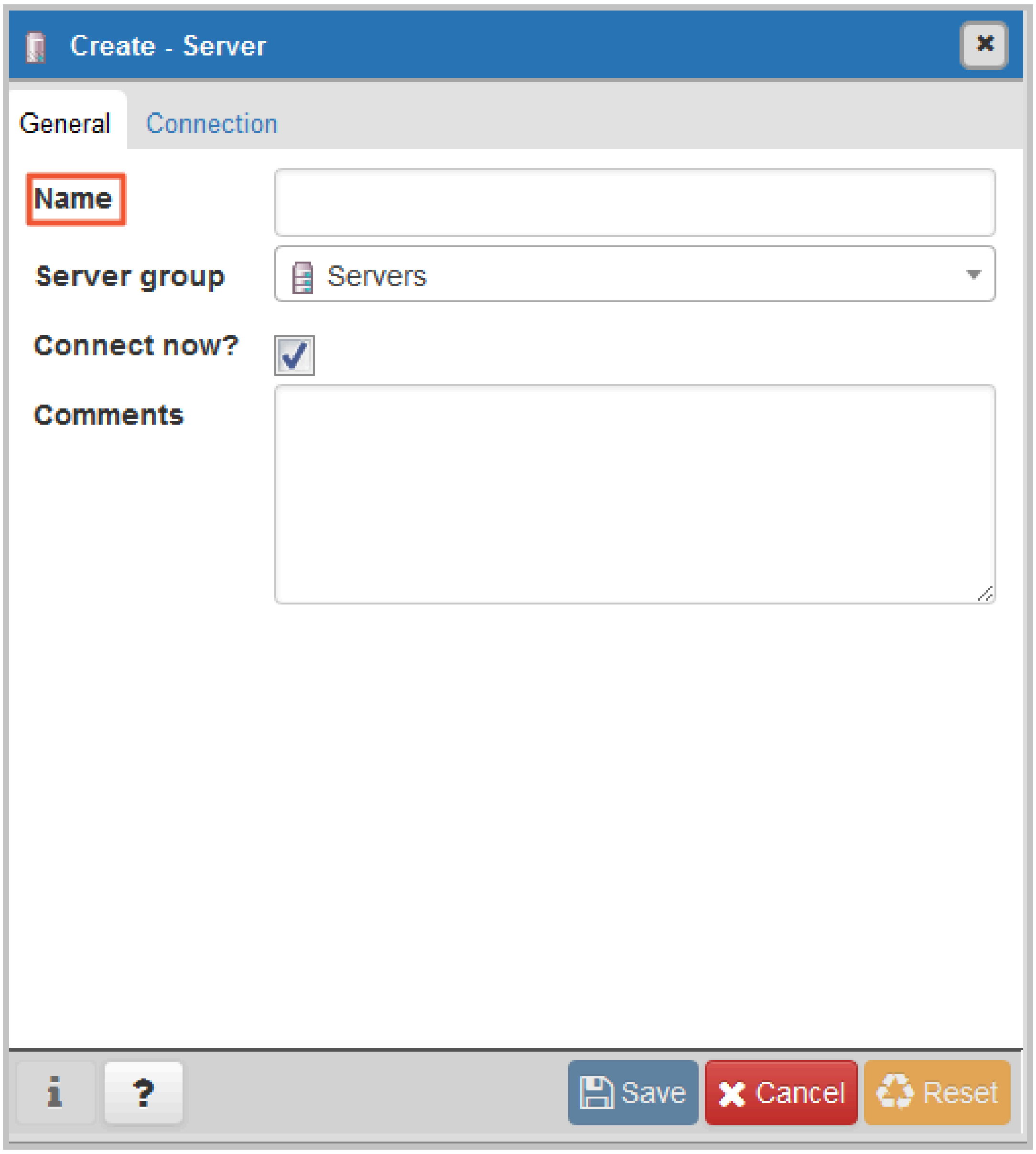
Name

Server group: Servers

Connect now? ☑

Comments

4. Click the **Connection** tab and enter the information of the destination instance, as shown in the following figure.



Name

Server group: Servers

Connect now? ☑

Comments

Parameters:

- Hostname or endpoint: the primary endpoint of the POLARDB cluster compatible with Oracle. You can view the endpoint on the **Basic Information** page in the ApsaraDB for POLARDB console, as shown in the following figure.

**Note:**

> Do not include the port number when specifying the endpoint. Example: abc.o.polardb.cn.rds.aliyuncs.com.

- Port: The port of the POLARDB cluster compatible with Oracle is 1521.
- Username: the account name of the cluster.
- Password: the password of the cluster.

5. Confirm the settings and click **Save**.

> **Note:**
>
> Additionally, you can run commands on clients to connect to a POLARDB cluster compatible with Oracle. For more information about how to download and install the clients, see Download clients and drivers.

# 3 Clients and Drivers

## 3.1 Download clients and drivers

This topic provides you with the download addresses of the clients and related drivers that are used to connect to ApsaraDB for PolarDB clusters compatible with Oracle.

In addition to logging on to a database from the ApsaraDB for PolarDB console, you can also download and install a client and then use the client to connect to an ApsaraDB for PolarDB cluster. For more information, see Connect to a POLARDB cluster compatible with Oracle.

For your application to connect to the ApsaraDB for PolarDB cluster, you can download and install a driver based on your actual conditions.

**Clients**

The following client package contains client installation packages for the Windows and Linux systems. For information about the installation method, see the Readme document in the package.

PolarDB-client.zip

**Drivers**

- PolarDB JDBC:

  polardb-jdbc_installer.zip

  For more information about how to use PolarDB JDBC, see PolarDB JDBC.
- PolarDB .NET:

  polardb-.net_installer.zip

  For more information about how to use PolarDB .NET, see PolarDB .NET.
- PolarDB OCI:

  polardb-oci_installer.zip

  For more information about how to use PolarDB OCI, see PolarDB (compatible with Oracle) OCI.

- PolarDB ODBC:

  polardb-odbc_installer.tar.gz

  For more information about how to use PolarDB ODBC, see PolarDB ODBC.

## 3.2 polartools

This topic describes how to download and install polartools.

### Context

polartools is a collection of Apsara PolarDB client tools for Linux. polartools includes the following tools:

- polarplus: the tool used by clients to connect to PolarDB databases compatible with Oracle.

  For more information, see polarplus.
- psql: the tool used by clients to connect to native PostgreSQL databases.

  For more information, see Documentation of PostgreSQL psql.
- pg_basebackup: a physical backup tool for PostgreSQL.

  For more information, see Documentation of PostgreSQL pg_basebackup.
- pg_dump: the logical backup tool of PostgreSQL. You can use this tool to back up one database of a cluster at a time.

  For more information, see Documentation of PostgreSQL pg_pgdump.
- pg_dumpall: the logical backup tool of PostgreSQL. You can use this tool to back up all the databases of a cluster at a time.

  For more information, see Documentation of PostgreSQL pg_pgdumpall.
- pg_restore: the tool used to restore PostgreSQL databases based on backup files. The backup files are created by pg_dump and pg_dump.

  For more information, see Documentation of PostgreSQL pg_restore.

This topic describes how to use polarplus. For more information about other tools, see the PostgreSQL documentation.

### Download polartools

To use polartools, click here to download the polartools package. After you have downloaded polartools, you must uncompress the package. You do not need to install the package. You can download and use polartools free of charge.

polartools has the following directory structure:

```
polartools
├──── bin
├──── etc
│     └──── sysconfig
├──── help
└──── lib
```

All tools are located in the bin directory. If you want to use a tool, you must add the tool location to the PATH environmental variable.

```
bin
├──── pg_basebackup
├──── pg_dump
├──── pg_dumpall
├──── pg_restore
├──── polarplusLauncher.sh
├──── polarplus.sh
└──── psql
```

### polarplus

polarplus is a utility that provides a command-line interface (CLI) for Apsara PolarDB. polarplus supports SQL statements, SPL anonymous blocks, and polarplus statements.

polarplus provides the following features:

- Queries a specified database object.
- Executes a stored procedure.
- Formats SQL statement output.
- Runs multiple scripts in a query.
- Runs operating system commands.
- Retains output logs.

To download and configure polarplus, follow these steps:

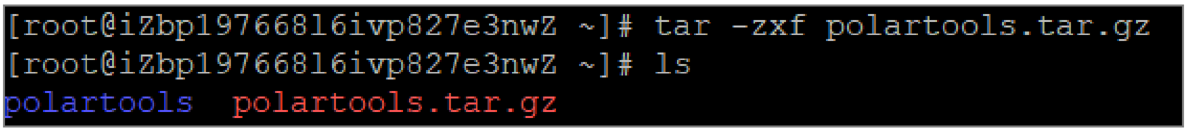
**Note:**

polarplus is dependent on JDK 1.8.

1. On the command line, enter wget to download polartools. For more information about the download address, see Download polartools.
2. Uncompress the polartools-linux.zip file.

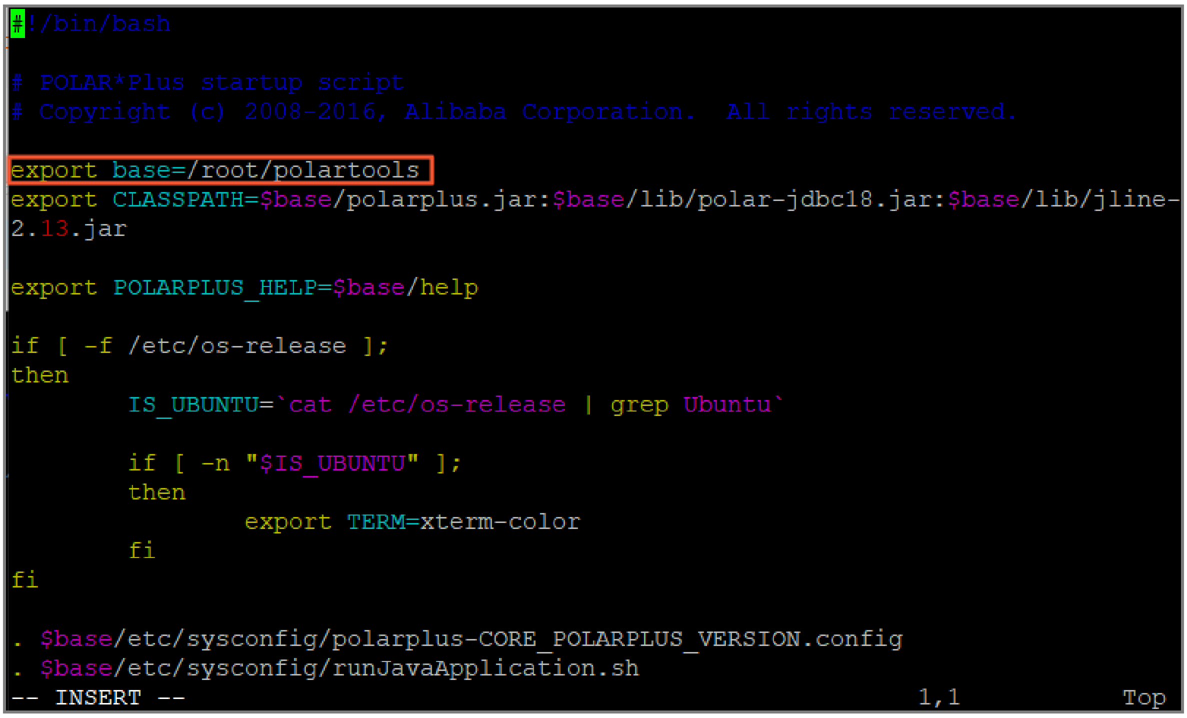
```
tar -zxf polartools.tar.gz
```

The following figure shows the uncompressed file.



```
[root@iZbp19766816ivp827e3nwZ ~]# tar -zxf polartools.tar.gz
[root@iZbp19766816ivp827e3nwZ ~]# ls
polartools  polartools.tar.gz
```

3. Go to the bin directory.
4. Open the polarplus.sh file.
5. Modify the setting of export base={pwd}/polartools by replacing pwd with the absolute path where the polartools folder is located.



```
#!/bin/bash

# POLAR*Plus startup script
# Copyright (c) 2008-2016, Alibaba Corporation.  All rights reserved.

export base=/root/polartools
export CLASSPATH=$base/polarplus.jar:$base/lib/polar-jdbc18.jar:$base/lib/jline-
2.13.jar

export POLARPLUS_HELP=$base/help

if [ -f /etc/os-release ];
then
        IS_UBUNTU=`cat /etc/os-release | grep Ubuntu`

        if [ -n "$IS_UBUNTU" ];
        then
                export TERM=xterm-color
        fi
fi

. $base/etc/sysconfig/polarplus-CORE_POLARPLUS_VERSION.config
. $base/etc/sysconfig/runJavaApplication.sh
-- INSERT --                                                 1,1           Top
```

6. To start polarplus, on the command line, execute the following statement on the Elastic Compute Service (ECS) instance or the server that connects to a PolarDB database:

```
polarplus [ -S[ILENT ] ] [ login | /NOLOG ] [ @scriptfile[.ext ] ]
```

| Parameter | Description |
|---|---|
| -S[ILENT ] | If you set this parameter, the polarplus logon banner and all relevant messages are disabled. |

| Parameter | Description |
|---|---|
| login | The logon information used to connect to the database server and databases.<br>Enter the logon information in the following format:<br>`username[/password][@{connectstring \| variable } ]`<br>For more information, see Table 3-1: Logon information.<br>The variable parameter specifies a variable defined in the login. sql file. This file contains a database connection string. |
| /NOLOG | When you start polarplus, if you specify /NOLOG, no database connection is established. To connect to a database and execute SQL statements or polarplus statements, do not use this mode.<br>**Note:**<br>After you start polarplus by specifying / NOLOG, you can execute the CONNECT statement to connect to a database. |
| scriptfile[.ext ] | scriptfile specifies the file name that is located in the current directory. This file contains SQL statements and polarplus statements that are automatically executed after you start polarplus.<br>.ext specifies the file extension. If the file extension is .sql, when you specify a script file, you can omit the .sql extension. When you create a script file, name the file with the extension. Otherwise, polarplus cannot access the file.<br>**Note:**<br>polarplus processes the files without extensions as .sql files. |

**Table 3-1: Logon information**

| Parameter | Description |
|---|---|
| username | The username used to connect to a database. |
| password | The password associated with the specified username. |

| Parameter | Description |
|---|---|
| connectstring | The database connection string is provided in the following format:<br>host[:port][/dbname][? ssl={true \| false}]<br>• host specifies the hostname or IP address of a database server.<br>**Note:** If you have not specified connectstring, variable, or NOLOG, the default host is the local host.<br>• If you use an Internet Protocol version 6 (IPv6) address to connect to a database, you must place the IP address in brackets ([]).<br>The following example shows how to use an IPv6 address to connect to a database:<br>polarplus  polardb/password@[fe80::20c:29ff:fe7c:78b2]:5444/polardb<br>• port specifies the port number on the database server to receive connection requests.<br>**Note:** If you have not specified a port number, the default value is 5444.<br>• dbname is the name of the database to connect to.<br>• If you require SSL connections, the connection string must include ? ssl = true and host:port. If you have not set the ssl parameter, the default value is false. |

The following example shows how to use polarplus to connect to a PolarDB database:

```
polarplus  polardb/password@pc-bp1zxxxxxxxxxxx.o.polardb.rds.aliyuncs.com:1521/
polardb
```

## 3.3 PolarDB JDBC

This topic describes how to use the PolarDB Java Database Connectivity (JDBC) driver to connect a Java application to an ApsaraDB for PolarDB database.

### Prerequisites

- You have created an account for an ApsaraDB for PolarDB cluster. For more information about how to create an account, see #unique_6.
- You have added the IP address of the host that you want to connect to the ApsaraDB for PolarDB cluster to the whitelist. For more information, see #unique_14.

### Context

JDBC is an application programming interface for the programming language Java, which defines how a client may access a database. ApsaraDB for PolarDB provides the Oracle JDBC driver based on the open-source PostgreSQL JDBC driver. The Oracle JDBC driver uses the PostgreSQL protocols for LAN communications, and it allows Java applications to connect to databases by using standard and database-independent Java code.

The PolarDB JDBC driver uses the PostgreSQL 3.0 protocol and is compatible with Java 6 ( JDBC 4.0), Java 7 (JDBC 4.1), and Java 8 (JDBC 4.2).

### Download the PolarDB JDBC driver

Download the PolarDB JDBC driver. Alibaba Cloud provides three JDBC versions compatible with Java 6, Java 7, and Java 8. The three JAR packages are named as `polardb-jdbc16.jar`, `polardb-jdbc17.jar`, and `polardb-jdbc18.jar`, respectively. You can select an appropriate JDBC version based on the JDK version used by your application.

### Configure the PolarDB JDBC driver

Before you use the PolarDB JDBC driver in a Java application, you must add the path of the JDBC driver package to `CLASSPATH`. For example, if the path of your JDBC driver is /usr /local/polardb/share/java/, run the following command to add the JDBC driver path to `CLASSPATH`:

```
export CLASSPATH=$CLASSPATH:/usr/local/polardb/share/java/<Name of the JAR
package.jar>
```

Example:

```
export CLASSPATH=$CLASSPATH:/usr/local/polardb/share/java/polardb-jdbc18.jar
```

You can run the following command to view the current JDBC version:

```
#java -jar <Name of the JAR package.jar>
```

Example:

```
#java -jar polardb-jdbc18.jar
```

```
POLARDB JDBC Driver 42.2.5.2.0
```

### Set up a Java project with Maven

If your Java project is built using Maven, run the following command to install the JDBC driver package to your local repository:

```
 mvn install:install-file -DgroupId=com.aliyun -DartifactId=<Name of the JAR package> -
Dversion=1.1.2 -Dpackaging=jar -Dfile=/usr/local/polardb/share/java/<Name of the JAR
 package.jar>
```

Example:

```
 mvn install:install-file -DgroupId=com.aliyun -DartifactId=polardb-jdbc18 -Dversion=1.1.
2 -Dpackaging=jar -Dfile=/usr/local/polardb/share/java/polardb-jdbc18.jar
```

Add the following dependency to the pom.xml file of the Maven project:

```
<dependency>
   <groupId>com.aliyun</groupId>
   <artifactId>parent</artifactId>
   <version>1.1.2</version>
</dependency>
```

Example:

```
<dependency>
   <groupId>com.aliyun</groupId>
   <artifactId>odps-jdbc</artifactId>
   <version>1.1.2</version>
</dependency>
```

### Set up a project for a Hibernate application

If your project uses Hibernate to connect to the database, open the Hibernate configuration file hibernate.cfg.xml and configure the driver class and dialect of the ApsaraDB for PolarDB database.

**Note:**

Only Hibernate version 3.6 and later support PostgresPlusDialect.

```
<property name="connection.driver_class">com.aliyun.polardb.Driver</property>
<property name="connection.url">jdbc:polardb://pc-***.o.polardb.rds.aliyuncs.com:
1521/polardb_test</property>
```

```
<property name="dialect">org.hibernate.dialect.PostgresPlusDialect</property>
```

### Load the PolarDB JDBC driver

```
Class.forName("com.aliyun.polardb.Driver");
```

### Example

```
package com.aliyun.polardb;

import java.sql.Connection;
import java.sql.Driver;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

/**
 * POLARDB JDBC DEMO
 * <p>
 * Please make sure the host ip running this demo is in you cluster's white list.
 */
public class PolarDBJdbcDemo {
  /**
   * Replace the following information.
   */
  private final String host = "***.o.polardb.rds.aliyuncs.com";
  private final String user = "***";
  private final String password = "***";
  private final String port = "1921";
  private final String database = "db_name";

  public void run() throws Exception {
    Connection connect = null;
    Statement statement = null;
    ResultSet resultSet = null;

    try {
      Class.forName("com.aliyun.polardb.Driver");

      Properties props = new Properties();
      props.put("user", user);
      props.put("password", password);
      String url = "jdbc:polardb://" + host + ":" + port + "/" + database;
      connect = DriverManager.getConnection(url, props);

      /**
       * create table foo(id int, name varchar(20));
       */
      String sql = "select id, name from foo";
      statement = connect.createStatement();
      resultSet = statement.executeQuery(sql);
      while (resultSet.next()) {
        System.out.println("id:" + resultSet.getInt(1));
        System.out.println("name:" + resultSet.getString(2));
      }
    } catch (Exception e) {
      e.printStackTrace();
      throw e;
    } finally {
      try {
        if (resultSet ! = null)
          resultSet.close();
        if (statement ! = null)
          statement.close();
        if (connect ! = null)
          connect.close();
      } catch (SQLException e) {
        e.printStackTrace();
        throw e;
      }
    }
  }

  public static void main(String[] args) throws Exception {
    PolarDBJdbcDemo demo = new PolarDBJdbcDemo();
    demo.run();
  }
}
```

In JDBC, a database is usually represented by a URL, for example:

```
jdbc:polardb://pc-***.o.polardb.rds.aliyuncs.com:1521/polardb_test? user=test&
password=Pw123456
```

| Parameter | Example | Description |
|---|---|---|
| URL prefix | jdbc:polardb:// | Set the prefix of the URL to jdbc:polardb://. |
| Endpoint | pc-***.o.polardb.rds.aliyuncs.com | The endpoint of the ApsaraDB for PolarDB cluster. For more information about how to query the endpoint, see #unique_15. |
| Port | 1521 | The port of the ApsaraDB for PolarDB cluster. Default value: 1521. |
| Database | polardb_test | The name of the database to be connected. |
| Username | test | The username for connecting to the ApsaraDB for PolarDB cluster. |
| Password | Pw123456 | The password of the username. |

When you perform a query on a database, you must create a Statement, PreparedStatment, or CallableStatement object.

In the preceding example, a Statement object is created. The following sample code creates a PreparedStatment object:

```
PreparedStatement st = conn.prepareStatement("select id, name from foo where id > ?") ;
st.setInt(1, 10);
resultSet = st.executeQuery();
while (resultSet.next()) {
    System.out.println("id:" + resultSet.getInt(1));
    System.out.println("name:" + resultSet.getString(2));
}
```

CallableStatement is used to process a stored procedure, as shown in the following example:

```
String sql = "{? =call getName (?, ?, ?)}" ;
CallableStatement stmt = conn.prepareCall(sql);
stmt.registerOutParameter(1, java.sql.Types.INTEGER);

//Bind IN parameter first, then bind OUT parameter
int id = 100;
stmt.setInt(2, id); // This would set ID as 102
stmt.registerOutParameter(3, java.sql.Types.VARCHAR);
stmt.registerOutParameter(4, java.sql.Types.INTEGER);

//Use execute method to run stored procedure.
stmt.execute();

//Retrieve name with getXXX method
String name = stmt.getString(3);
Integer msgId = stmt.getInt(4);
Integer result = stmt.getInt(1);
System.out.println("Name with ID:" + id + " is " + name + ", and messegeID is " + msgId +
 ", and return is " + result);
```

The following code shows how to create the stored procedure getName used in the preceding code:

```
CREATE OR REPLACE FUNCTION getName(
    id        In      Integer,
    name      Out     Varchar2,
    result    Out     Integer
  ) Return Integer
Is
  ret     Int;
Begin
  ret := 0;
  name := 'Test';
  result := 1;
  Return(ret);
End;
```

# 3.4 PolarDB .NET

This topic describes how to use the ADO.NET Data Provider for PolarDB (PolarDB .NET) driver to connect a C# application to an ApsaraDB for PolarDB database.

### Prerequisites

- You have created an account for an ApsaraDB for PolarDB cluster. For more information about how to create an account, see #unique_6.
- You have added the IP address of the host that you want to connect to the ApsaraDB for PolarDB cluster to the whitelist. For more information, see #unique_14.

### Context

PolarDB .NET is a driver used to connect to ApsaraDB for PolarDB by using a programming language, including C#, Visual Basic, and F #. The driver is compatible with Entity Framework Core and Entity Framework 6.x. You can use this driver with Entity Framework to quickly develop applications.

The current driver uses the PostgreSQL 3.0 protocol and is compatible with .NETFramework 4.x and .NET Core 2.x.

### Entity Framework overview

Entity Framework is a popular object-relational mapper (O/RM) on the .NET platform. It works with Language-Integrated Query (LINQ) technologies to greatly accelerate the development of backend applications if the C# language is used.

The PolarDB .NET driver provides the PolarDB Entity Framework 5 and 6 dlls to help you use Entity Framework.

For more information about Entity Framework, visit its official website at https://docs.microsoft.com/en-au/ef/.

### Download the PolarDB .NET driver

Download the PolarDB .NET driver.

### Install the PolarDB .NET driver

1. Decompress the PolarDB .NET driver.

   ```
   unzip POLARDB-for-Oracle-.net_installer.zip
   ```

2. Import the driver to the Visual Studio project.

   Add the following content to the <Project> node of sample. csproj or the GUI of Visual Studio.

   ```
   <Project>
     ...
     <ItemGroup>
       <Reference Include="POLARDB.POLARDBClient, Version=4.0.4.1, Culture=neutral,
   PublicKeyToken=5d8b90d52f46fda7">
         <HintPath>${your path}\POLARDB.POLARDBClient.dll</HintPath>
       </Reference>
     </ItemGroup>
     ...
   ```

```
</Project>
```

### Example

In the Samples folder, you can see the polardb-sample.sql file and multiple sample project files. The following procedure shows how to run these sample projects.

1. Connect to a database. For more information, see Connect to a POLARDB cluster compatible with Oracle.
2. Run the following command to create a project namedsampledb.

   ```
   CREATE DATABASE sampledb;
   ```
3. Import the databases, tables, data, and functions that are required for testing to database sampledb.

   ```
   \i ${your path}/polardb-sample.sql
   ```
4. After the data is imported, write the C # code.

   The following sample code shows how to query, update, and call stored procedures.

   ```
   using System;
   using System.Data;
   using POLARDB.POLARDBClient;
   /*
    * This class provides a simple way to perform DML operation in POLARDB
    *
    * @revision 1.0
    */

   namespace POLARDBClientTest
   {

       class SAMPLE_TEST
       {

           static void Main(string[] args)
           {
               POLARDBConnection conn = new POLARDBConnection("Server=localhost;Port=
   1521;User Id=polaruser;Password=password;Database=sampledb");
               try
               {
                   conn.Open();

                   //Simple select statement using POLARDBCommand object
                   POLARDBCommand POLARDBSeletCommand = new POLARDBCommand("
   SELECT EMPNO,ENAME,JOB,MGR,HIREDATE FROM EMP",conn);
                   POLARDBDataReader SelectResult =  POLARDBSeletCommand.ExecuteReader
   ();
                   while (SelectResult.Read())
                   {
                       Console.WriteLine("Emp No" + " " + SelectResult.GetInt32(0));
                       Console.WriteLine("Emp Name" + " " + SelectResult.GetString(1));
                       if (SelectResult.IsDBNull(2) == false)
                           Console.WriteLine("Job" + " " + SelectResult.GetString(2));
                       else
   ```

```
                    Console.WriteLine("Job" + " null ");
                if (SelectResult.IsDBNull(3) == false)
                    Console.WriteLine("Mgr" + " " + SelectResult.GetInt32(3));
                else
                    Console.WriteLine("Mgr" + "null");
                if (SelectResult.IsDBNull(4) == false)
                    Console.WriteLine("Hire Date" + " " + SelectResult.GetDateTime(4));
                else
                    Console.WriteLine("Hire Date" + " null");
                Console.WriteLine("---------------------------------");
            }

            //Insert statement using POLARDBCommand Object
            SelectResult.Close();
            POLARDBCommand POLARDBInsertCommand = new POLARDBCommand
("INSERT INTO EMP(EMPNO,ENAME) VALUES((SELECT COUNT(EMPNO) FROM EMP),'
JACKSON')",conn);
            POLARDBInsertCommand.ExecuteScalar();
            Console.WriteLine("Record inserted");

            //Update  using POLARDBCommand Object
            POLARDBCommand  POLARDBUpdateCommand = new POLARDBCommand("
UPDATE EMP SET ENAME ='DOTNET' WHERE EMPNO < 100",conn);
            POLARDBUpdateCommand.ExecuteNonQuery();
            Console.WriteLine("Record has been updated");
            POLARDBCommand POLARDBDeletCommand = new POLARDBCommand("
DELETE FROM EMP WHERE EMPNO < 100",conn);
            POLARDBDeletCommand.CommandType= CommandType.Text;
            POLARDBDeletCommand.ExecuteScalar();
            Console.WriteLine("Record deleted");

            //procedure call example
            try
            {
                POLARDBCommand callable_command = new POLARDBCommand("
emp_query(:p_deptno,:p_empno,:p_ename,:p_job,:p_hiredate,:p_sal)", conn);
                callable_command.CommandType = CommandType.StoredProcedure;
                callable_command.Parameters.Add(new POLARDBParameter("p_deptno
",POLARDBTypes.POLARDBDbType.Numeric,10,"p_deptno",ParameterDirection.Input,
false ,2,2,System.Data.DataRowVersion.Current,20));
                callable_command.Parameters.Add(new POLARDBParameter("p_empno
", POLARDBTypes.POLARDBDbType.Numeric,10,"p_empno",ParameterDirection.
InputOutput,false ,2,2,System.Data.DataRowVersion.Current,7369));
                callable_command.Parameters.Add(new POLARDBParameter("p_ename
", POLARDBTypes.POLARDBDbType.Varchar,10,"p_ename",ParameterDirection.
InputOutput,false ,2,2,System.Data.DataRowVersion.Current,"SMITH"));
                callable_command.Parameters.Add(new POLARDBParameter("p_job",
POLARDBTypes.POLARDBDbType.Varchar,10,"p_job",ParameterDirection.Output,false ,
2,2,System.Data.DataRowVersion.Current,null));
                callable_command.Parameters.Add(new POLARDBParameter("p_hiredate
", POLARDBTypes.POLARDBDbType.Date,200,"p_hiredate",ParameterDirection.Output,
false ,2,2,System.Data.DataRowVersion.Current,null));
                callable_command.Parameters.Add(new POLARDBParameter("p_sal",
POLARDBTypes.POLARDBDbType.Numeric,200,"p_sal",ParameterDirection.Output,false
 ,2,2,System.Data.DataRowVersion.Current,null));
                callable_command.Prepare();
                callable_command.Parameters[0].Value = 20;
                callable_command.Parameters[1].Value = 7369;
                POLARDBDataReader result = callable_command.ExecuteReader();
                int fc = result.FieldCount;
                for(int i=0;i<fc;i++)
                    Console.WriteLine("RESULT["+i+"]="+ Convert.ToString(callable_command
.Parameters[i].Value));
                result.Close();
            }
            catch(POLARDBException exp)
            {
                if(exp.ErrorCode.Equals("01403"))
                    Console.WriteLine("No data found");
                else if(exp.ErrorCode.Equals("01422"))
                    Console.WriteLine("More than one rows were returned by the query");
                else
                    Console.WriteLine("There was an error Calling the procedure. \nRoot
Cause:\n");
                Console.WriteLine(exp.Message.ToString());
            }

            //Prepared statement
            string updateQuery  = "update emp set ename = :Name where empno = :ID";
            POLARDBCommand Prepared_command = new POLARDBCommand(
updateQuery, conn);
            Prepared_command.CommandType = CommandType.Text;
            Prepared_command.Parameters.Add(new POLARDBParameter("ID",
POLARDBTypes.POLARDBDbType.Integer));
            Prepared_command.Parameters.Add(new POLARDBParameter("Name",
POLARDBTypes.POLARDBDbType.Text));
            Prepared_command.Prepare();
            Prepared_command.Parameters[0].Value = 7369;
            Prepared_command.Parameters[1].Value = "Mark";
            Prepared_command.ExecuteNonQuery();
            Console.WriteLine("Record Updated...");
        }

        catch(POLARDBException exp)
        {
            Console.WriteLine(exp.ToString() );
        }
        finally
        {
            conn.Close();
        }

    }
}
}
```

Where, the code string `Server=localhost;Port=1521;User Id=polaruser;Password=password;Database=sampledb` is a connection string used to connect to the database.

The connection string consists of the `Server`, `Port`, `User Id`, `Password`, and `Database` parameters, as described in the following table.

| Parameter | Example | Description |
|---|---|---|
| Server | `localhost` | The endpoint of the ApsaraDB for PolarDB cluster. For information about how to query the endpoint, see #unique_15. |
| Port | `1521` | The port of the ApsaraDB for PolarRDB cluster. Default value: 1521. |

| Parameter | Example | Description |
|---|---|---|
| User Id | polaruser | The username for connecting to the ApsaraDB for PolarDB cluster. |
| Password | password | The password of the username. |
| Database | sampledb | The name of the database to be connected. |

## 3.5 PolarDB ODBC

This topic describes how to use the PolarDB Open Database Connectivity (ODBC) driver to connect a Unix or Linux application to an ApsaraDB for PolarDB cluster.

### Prerequisites

- You have created an account for an ApsaraDB for PolarDB cluster. For more information about how to create an account, see #unique_6.
- You have added the IP address of the host that you want to connect to the ApsaraDB for PolarDB cluster to the whitelist. For more information, see #unique_14.
- The server where the PolarDB ODBC driver is installed must run 64-bit Linux.

### Download the PolarDB ODBC driver

Download the PolarDB ODBC driver.

### Install the PolarDB ODBC driver

ApsaraDB for PolarDB provides an ODBC driver package. You can use it after decompression without installation. Run the following command to decompress the package:

```
tar -zxvf polardb-odbc.tar.gz
```

### Connect to an ApsaraDB for PolarDB cluster

1. Install Libtool on the Linux server. Libtool must be version 1.5.1 or later.

   ```
   yum install -y libtool
   ```

2. Install unixODBC-devel on the Linux server.

   ```
   yum install -y unixODBC-devel
   ```

3. Edit the `odbcinst.ini` file in the /etc directory.

   ```
   vim /etc/odbcinst.ini
   ```

4. Add the following information to the `odbcinst.ini` file.

   ```
   [POLARDB]
   Description = ODBC for POLARDB
   Driver      = /root/target/lib/unix/polar-odbc.so
   Setup       = /root/target/lib/unix/libodbcpolarS.so
   Driver64    = /root/target/lib/unix/polar-odbc.so
   Setup64     = /root/target/lib/unix/libodbcpolarS.so
   Database    = <Database name>
   Servername  = <Endpoint of the ApsaraDB for POLARDB cluster>
   Password    = <Password>
   Port        = <Port>
   Username    = <Username>
   Trace       = yes
   TraceFile   = /tmp/odbc.log
   FileUsage   = 1
   ```

   **Note:**

   - For more information about how to query the endpoint of an ApsaraDB for PolarDB cluster, see #unique_15.
   - Replace /root in the sample code with the actual path of the `target` folder.

5. Connect to the ApsaraDB for PolarDB cluster.

   ```
   $isql -v POLARDB
   +---------------------------------------+
   | Connected!                            |
   |                                       |
   | sql-statement                         |
   | help [tablename]                      |
   | quit                                  |
   |                                       |
   ```

```
+---------------------------------------+
SQL>
```

### Example

The following examples show how to run the `Test1` and `Test2` files.

1. Open the `samples` folder in the ODBC driver folder.

   ```
   cd samples
   ```

2. Compile the sample test. The following test files are generated: `Test1` and `Test2`.

   ```
   make
   ```

3. Run `Test1` and `Test2`.

   ```
   . /Test1
   ## Run Test1

   . /Test2
   ## Run Test2
   ```

   **Note:**

   - `Test1` contains the sample code to perform the add, delete, modify, and query operations. `Test2` contains the sample code to print the values of cursors as output parameters.
   - The following sample code is only a snippet of the source code. To check the complete sample code, reference the `Test1` and `Test2` files in the `samples` folder of the ODBC driver package.

Sample code for Test1:

```
...

int main(int argc, char* argv[])
{
    /*Initialization*/
    RETCODE rCode;
    HENV *hEnv = (HENV*)malloc(sizeof(HENV));
    HDBC *hDBC = (HDBC*)malloc(sizeof(HDBC));
    HSTMT *hStmt = (HSTMT*)malloc(sizeof(HSTMT));
    Connect("POLARDB","user","",&hEnv,&hDBC);
    rCode = SQLAllocStmt(*hDBC,hStmt);
    rCode = SQLAllocHandle(SQL_HANDLE_STMT,*hDBC,hStmt);
    /*Add, delete, modify, and query operations*/
    ExecuteInsertStatement(&hStmt,(UCHAR*) "INSERT INTO EMP(EMPNO,ENAME) VALUES((
SELECT COUNT(EMPNO) FROM EMP),'JACKSON')");
    ExecuteUpdate(&hStmt,(UCHAR*) "UPDATE EMP SET ENAME='ODBC Test' WHERE EMPNO
 < 100");
    ExecuteDeletStatement(&hStmt,(UCHAR*) "DELETE FROM EMP WHERE EMPNO<100");
```

```
    ExecuteSimple_Select(&hStmt,(UCHAR*) "SELECT EMPNO,ENAME,JOB,MGR,HIREDATE
FROM EMP where empno = 7369");
    /*Disconnection*/
    Disconnect(&hEnv,&hDBC,&hStmt);
    /*clean up*/
    free(hEnv);
    free(hDBC);
    free(hStmt);

    return 0;
}
```

Sample code for Test2:

```
int main(int argc, char* argv[])
{
    /*Definition*/
    RETCODE rCode;
    SQLUSMALLINT a;
    SQLINTEGER Num1IndOrLen;
    SQLSMALLINT iTotCols = 0;

    int j;
    SDWORD cbData;
    /*Initialization*/
    HENV *hEnv = (HENV*)malloc(sizeof(HENV));
    HDBC *hDBC = (HDBC*)malloc(sizeof(HDBC));
    HSTMT *hStmt = (HSTMT*)malloc(sizeof(HSTMT));
    HSTMT *hStmt1 = (HSTMT*)malloc(sizeof(HSTMT));
    /**Connection establishment**/
    Connect("POLARDB","user","***",&hEnv,&hDBC);
    rCode = SQLAllocStmt(*hDBC,hStmt);
    rCode = SQLAllocStmt(*hDBC,hStmt1);

    rCode = SQLAllocHandle(SQL_HANDLE_STMT,*hDBC,hStmt);
    rCode = SQLAllocHandle(SQL_HANDLE_STMT,*hDBC,hStmt1);
    /*begin*/
    ExecuteSimple_Select(&hStmt1,(UCHAR*) "BEGIN;");
    /*prepare*/
    RETCODE rc = SQLPrepare((*hStmt),(SQLCHAR*)"{ call refcur_inout_callee2(?,?)}",
SQL_NTS);

    rc = SQLBindParameter((*hStmt),1, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR,
SQL_REFCURSOR,0, 31,
                strName, 31, &Num1IndOrLen);
    rc = SQLBindParameter((*hStmt),2, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR,
SQL_REFCURSOR,0, 31,
                    &strName1, 31, &Num1IndOrLen);

    Num1IndOrLen=0;
    /*execute*/
    rc = SQLExecute((*hStmt));

    if(rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO)
    {
        printf("\nstrName __________ = %s\n",strName);
        printf("\nstrName 1__________ = %s\n",strName1);


    }
    printf("\n First Cursor as OUT Parameter \n")   ;
```

## 3.6 PolarDB (compatible with Oracle) OCI

This topic describes how to use the PolarDB Oracle Call Interface (OCI) driver to connect to a PolarDB database compatible with Oracle.

### Prerequisites

- You have created an account for an ApsaraDB for PolarDB cluster. For more information about how to create an account, see #unique_6.
- You have added the IP address of the host that you want to connect to the ApsaraDB for PolarDB cluster to the whitelist. For more information, see #unique_14.
- The operating system of the server where PolarDB OCI is installed must be 64-bit Linux or Windows.
- Make sure that the development kit of the Oracle OCI driver is installed.

### Context

PolarDB OCI is the native C language interface to databases of Apsara PolarDB compatible with Oracle. You can use PolarDB OCI to build other language-specific interfaces, including PolarDB JDBC, PolarDB .Net, and PolarDB ODBC. It allows you to execute query statements and make SQL function calls for PolarDB databases compatible with Oracle.

The driver version is PostgreSQL 3.0.

### Download the PolarDB OCI driver

polardb-oci.tar.gz

### Install the PolarDB OCI driver

Decompress the driver package and manually import the following driver files to environment variables. This allows you to find the location of the driver when you compile a demo.

You can manually import the driver files to the environment variables in Linux and Windows as follows:

- Linux

    1. Copy the libpolaroci.so.10.2, libiconv.so.2, and libpq.so.5.11 files to the /usr/lib directory.
    2. Create a symbolic link.

        ```
        ln -s /usr/lib/libpolaroci.so.10.2 /usr/lib/libpolaroci.so
        ln -s /usr/lib/libiconv.so.2 /usr/lib/libiconv.so
        ln -s /usr/lib/libpq.so.5.11 /usr/lib/libpq.so
        ln -s /usr/lib/libpq.so.5.11 /usr/lib/libpq.so.5
        ```

    3. Set environment variables in Linux.

        ```
        export LD_LIBRARY_PATH= /usr/lib
        ```

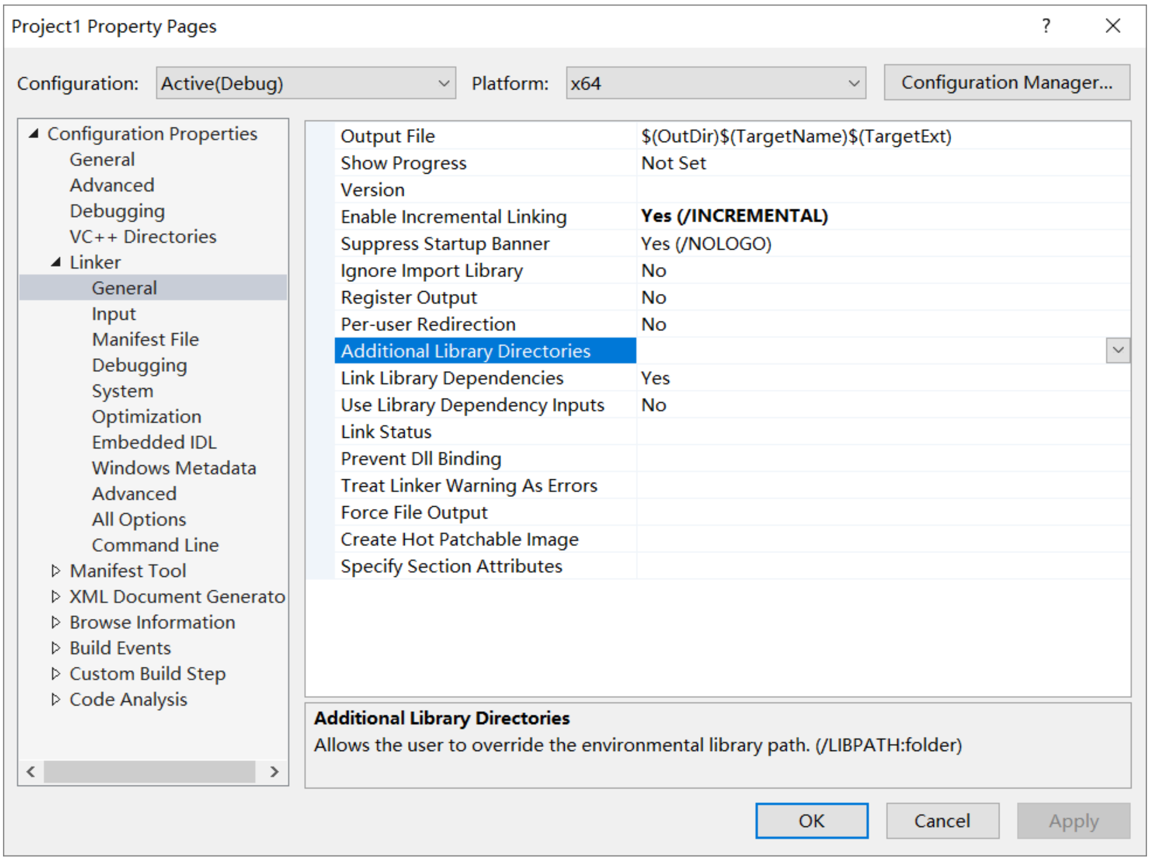
**Note:**

- If the libiconv.so files already exist in the Linux, you can directly use these files. You can also follow the instructions in libiconv documentation to download and install libiconv, and then use the compiled .so files.
- In Linux, the libiconv.so files provided by the PolarDB-O OCI driver are for reference only.

- Windows

1. Set environment variables.

   The IDE editor in Windows is capable of importing the paths of linked files. In this topic, Visual Studio is used to demonstrate how to import linked file paths, as shown in the following figure.



Configuration: Active(Debug) Platform: x64

| | |
|---|---|
| Output File | $(OutDir)$(TargetName)$(TargetExt) |
| Show Progress | Not Set |
| Version | |
| Enable Incremental Linking | **Yes (/INCREMENTAL)** |
| Suppress Startup Banner | Yes (/NOLOGO) |
| Ignore Import Library | No |
| Register Output | No |
| Per-user Redirection | No |
| Additional Library Directories | |
| Link Library Dependencies | Yes |
| Use Library Dependency Inputs | No |
| Link Status | |
| Prevent Dll Binding | |
| Treat Linker Warning As Errors | |
| Force File Output | |
| Create Hot Patchable Image | |
| Specify Section Attributes | |

**Additional Library Directories**
Allows the user to override the environmental library path. (/LIBPATH:folder)

2. On the properties tab of the project, add Additional Library Directories. Then, add the .dll files in the driver directory to the **Additional Library Directories**.

## Sample code

The demo polardb_demo in the sample directory is used as an example. It demonstrates how to create tables, run queries, and perform other operations.

```
/*
 ========================================================================
 * Copyright (c) 2004-2019 POLARDB Corporation. All Rights Reserved.
 * =======================================================================
 */
#include <stdio.h>
#include <stdlib.h>

#include <string.h>
#include <oci.h>

#ifdef WIN32
```

```
#include <time.h>
#else
#include <sys/time.h>
#endif

/* Define a macro to handle errors */
#define HANDLE_ERROR(x,y) check_oci_error(x,y)

#define DATE_FMT "DAY, MONTH DD, YYYY"
#define DATE_LANG "American"

sword ConvertStringToDATE( char *datep, char *formatp, dvoid *datepp );
/* A Custom Routine to handle errors,     */

/* this demonstrates the Error/ Exception Handling in OCI */
void check_oci_error (dvoid * errhp, sword status);

/*
 * <<<<<<<<<<<<<<<<<<< FUNCTION PROTOTYPES
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
 */

/* Initialize    & Allocate all handles */
void
initHandles (OCISvcCtx **, OCIServer **, OCISession **, OCIError **,
        OCIEnv **);

/* logon to the database and begin user-session */
void
logon (OCISvcCtx **, OCIServer **, OCISession **, OCIError **,
      OCIEnv **, text *, text *, text *);

/* Create required table(s)  */
void create_table (OCISvcCtx *, OCIError *, OCIEnv *);

/* prepare data for our examples */
void prepare_data (OCISvcCtx *, OCIError *, OCIEnv *);

/* create procedures/functions to demonstrate in the example */
void create_stored_procs (OCISvcCtx *, OCIError *, OCIEnv *);

/* select and print data by iterating through resultSet */
void select_print_data (OCISvcCtx *, OCIError *, OCIEnv *);

/* demonstrate calling stored procedures and retrieving values */

/* proc1 demonstrates IN OUT */
void call_stored_proc1 (OCISvcCtx *, OCIError *, OCIEnv *);

/* proc2 demonstrates OUT */
void call_stored_proc2 (OCISvcCtx *, OCIError *, OCIEnv *);

/* drop required table(s) */
void drop_table (OCISvcCtx *, OCIError *, OCIEnv *);

/* drop stored procedures and functions */
void drop_stored_procs (OCISvcCtx *, OCIError *, OCIEnv *);

/* clean-up main handles before exit */
void
cleanup (OCISvcCtx **, OCIServer **, OCISession **, OCIError **, OCIEnv **);

/*
```

```
 * <<<<<<<<<<<<<<<<<<<<<<<<< END OF FUNCTION PROTOYPES
<<<<<<<<<<<<<<<<<<<<<<<<<<
 */


/* <<<<<<<<<< Global Variables */
ub4 init_mode = OCI_DEFAULT;
ub4 auth_mode = OCI_CRED_RDBMS;

/* <<<<<<<<<< End Global Variables */

int
main (void)
{

  /*
   * Declare Handles, a typical OCI program would need atleast
   * following handles Enviroment Handle Error Handle Service Context
   * Handle Server Handle User Session (Authentication Handle)
   */

  /* Enviroment */
  OCIEnv *envhp;

  /* Error */
  OCIError *errhp;

  /* Service Context */
  OCISvcCtx *svchp;

  /* Server */
  OCIServer *srvhp;

  /* Session(authentication) */
  OCISession *authp;


  /*
   * End of Handle declaration
   */

  /*
   * Declare local variables,
   */
  text *username = (text *) "parallels";
  text *passwd = (text *) "";

  /*
   * Oracle Instant Client Connection String
   */
  text *server = (text *) "//localhost:5432/postgres";

  /*
   * Initialize and Allocate handles
   */
  initHandles (&svchp, &srvhp, &authp, &errhp, &envhp);

  /*
   * logon to the database
   */
  logon (&svchp, &srvhp, &authp, &errhp, &envhp, username, passwd, server);

  /*
   * Create table(s) required for this example
   */
  create_table (svchp, errhp, envhp);

  /*
   * insert data into table
   */
  prepare_data (svchp, errhp, envhp);

  /*
   * create stored procedures & functions
   */
  create_stored_procs (svchp, errhp, envhp);

  /*
   * select and print data by iterating through simple resultSet
   */
  select_print_data (svchp, errhp, envhp);

  /*
   * demonstrate calling stored procedures and retrieving values
   */
  call_stored_proc1 (svchp, errhp, envhp);

  /*
   * demonstrate OUT parameters
   */
  call_stored_proc2 (svchp, errhp, envhp);

  /*
   * Drop table(s) used in this example
   */
  drop_table (svchp, errhp, envhp);

  /*
   * Drop stroed procedures & functions used in this example
   */
  drop_stored_procs (svchp, errhp, envhp);

  /*
   * clean up resources
   */
  cleanup (&svchp, &srvhp, &authp, &errhp, &envhp);

  return 0;
}

/* A Custom Routine to handle errors,     */

/* this demonstrates the Error/ Exception Handling in OCI */

void
check_oci_error (dvoid * errhp, sword status)
{
  text errbuf[512];
  sb4 errcode;

  if (status == OCI_SUCCESS)
    {
      return;
    }
  switch (status)
    {
    case OCI_SUCCESS_WITH_INFO:
      printf ("OCI_SUCCESS_WITH_INFO:\n");
      OCIErrorGet (errhp, (ub4) 1, (text *) 0, &errcode,
          errbuf, (ub4) sizeof (errbuf), OCI_HTYPE_ERROR);
      printf ("%s", errbuf);
      break;
    case OCI_NEED_DATA:
      printf ("Error - OCI_NEED_DATA\n");
      break;
    case OCI_NO_DATA:
      printf ("Error - OCI_NO_DATA\n");
      break;
    case OCI_ERROR:
      printf ("Error - OCI_ERROR:\n");
      OCIErrorGet (errhp, (ub4) 1, (text *) 0, &errcode,
          errbuf, (ub4) sizeof (errbuf), OCI_HTYPE_ERROR);
      printf ("%s", errbuf);
      break;
    case OCI_INVALID_HANDLE:
      printf ("Error - OCI_INVALID_HANDLE\n");
      break;
    case OCI_STILL_EXECUTING:
      printf ("Error - OCI_STILL_EXECUTING\n");
      break;
    case OCI_CONTINUE:
      printf ("Error - OCI_CONTINUE\n");
      break;
    default:
      break;
    }

  /*
   * exit app
   */
  exit((int)status);
}

/* Initialize & Allocate required handles */
void
initHandles (OCISvcCtx ** svchp, OCIServer ** srvhp, OCISession ** authp,
        OCIError ** errhp, OCIEnv ** envhp)
{

  /*
   * Now Starts the Section where we have to initialize & Allocate
   * basic handles. This is a compulsory setup or initilization which
   * is required before we can proceed to logon and work with the
   * database. This initialization and prepration will include the
   * following steps
   *
   * 1. Initialize the OCI (OCIInitialize()) 2. Initialize the
   * Environment (OCIEnvInit()) 3. Initialize & Allocate Error Handle
   * 4. Initialize & Allocate Service Context Handle 5. Initialize &
   * Allocate Session Handle 6. Initialize & Allocate Server Handle
   *
   * As per the new versions of OCI , instead of using OCIInitialize()
   * and OCIEnvInit(), we can do this with one API Call called
   * OCIEnvCreate().
   */

  /*
   * Initialize OCI
   */
  if (OCIInitialize (init_mode, (dvoid *) 0,
           (dvoid * (*)(dvoid *, size_t)) 0,
           (dvoid * (*)(dvoid *, dvoid *, size_t)) 0,
          (void (*)(dvoid *, dvoid *)) 0) ! = OCI_SUCCESS)
    {
      printf ("ERROR: failed to initialize OCI\n");
      exit (1);
    }
  /*
   * Initialize Enviroment.
   */
  HANDLE_ERROR (*envhp,
      OCIEnvInit (&(*envhp), OCI_DEFAULT, (size_t) 0,
            (dvoid **) 0));

  /*
   * Initialize & Allocate Error Handle
   */
  HANDLE_ERROR (*envhp,
      OCIHandleAlloc (*envhp, (dvoid **) & (*errhp),
            OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0));

  /*
   * Initialize & Allocate Service Context Handle
   */
  HANDLE_ERROR (*errhp,
      OCIHandleAlloc (*envhp, (dvoid **) & (*svchp),
            OCI_HTYPE_SVCCTX, (size_t) 0, (dvoid **) 0));

  /*
   * Initialize & Allocate Session Handle
   */
  HANDLE_ERROR (*errhp,
      OCIHandleAlloc (*envhp, (dvoid **) & (*authp),
            OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0));

  /*
   * Initialize & Allocate Server Handle
   */
  HANDLE_ERROR (*errhp,
      OCIHandleAlloc (*envhp, (dvoid **) & (*srvhp),
            OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0));

}

void
logon (OCISvcCtx ** svchp, OCIServer ** srvhp, OCISession ** authp,
      OCIError ** errhp, OCIEnv ** envhp, text * username, text * passwd,
      text * server)
{

  /*
   * Now Starts our Logon to the Database Server which includes two
   * steps
   *
   * 1. Attaching to the Server 2. Starting or Begining of the Session
   *
   * This is the complex logon. The easy ways to logon is to avoid
   * server attach and session begin and simply use OCILogon() or
   * OCILogon2() and then logoff using OCILogoff()
   */

  /*
   * Attach to the server
   */

  HANDLE_ERROR (*errhp,
      OCIServerAttach (*srvhp, *errhp, server,
              (ub4) strlen ((char *) server),
              OCI_DEFAULT));

  /*
   * The following code will start a session but before we start a
   * session we have to 1. Set the Server Handle which is now attached
   * into Service Context Handle 2. Set the Username and password into
   * Session Handle
   */

  /*
   * Set the Server Handle into Service Context Handle
   */

  HANDLE_ERROR (*errhp,
      OCIAttrSet (*svchp, OCI_HTYPE_SVCCTX,
             (dvoid *) (*srvhp), (ub4) 0, OCI_ATTR_SERVER,
             *errhp));

  /*
   * Set the username and password into session handle
   */

  HANDLE_ERROR (*errhp,
      OCIAttrSet (*authp, OCI_HTYPE_SESSION,
             (dvoid *) username,
             (ub4) strlen ((char *) username),
             OCI_ATTR_USERNAME, *errhp));
  HANDLE_ERROR (*errhp,
      OCIAttrSet (*authp, OCI_HTYPE_SESSION, (dvoid *) passwd,
             (ub4) strlen ((char *) passwd), OCI_ATTR_PASSWORD,
             *errhp));

  /*
   * Now FINALLY Begin our session
   */

  HANDLE_ERROR ((*errhp),
      OCISessionBegin (*svchp, *errhp,
              *authp, auth_mode, OCI_DEFAULT));

  printf ("********************************************\n");
  printf ("Milestone  : Logged on as --> '%s'\n", username);
  printf ("********************************************\n");

  /*
   * After we Begin our session we will have to set the Session
   */

  /*
   * (authentication) handle into Service Context Handle
   */

  HANDLE_ERROR (*errhp,
      OCIAttrSet (*svchp, OCI_HTYPE_SVCCTX,
             (dvoid *) (*authp), (ub4) 0,
             OCI_ATTR_SESSION, *errhp));
}

/* Create table(s) required for this example */
void
create_table (OCISvcCtx * svchp, OCIError * errhp, OCIEnv * envhp)
{
  OCIStmt *stmhp;
  text *create_statement =
    (text *)"CREATE TABLE OCISPEC \n (ENAME VARCHAR2(20)\n, MGR NUMBER\n, HIREDATE
 DATE)";
  ub4 status = OCI_SUCCESS;

  /*
   * Initialize & Allocate Statement Handle
   */
  HANDLE_ERROR (errhp,
        OCIHandleAlloc (envhp, (dvoid **) & stmhp,
                OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));


  /*
   * Prepare the Create statement
   */

  HANDLE_ERROR (errhp,
        OCIStmtPrepare (stmhp, errhp,
                create_statement,
                strlen ((const char *) create_statement),
                OCI_NTV_SYNTAX, OCI_DEFAULT));


  /*
   * Execute the Create Statement
   */
  if ((status = OCIStmtExecute (svchp, stmhp, errhp,
                   (ub4) 1, (ub4) 0, NULL, NULL, OCI_DEFAULT)) < OCI_SUCCESS)
    {
      printf ("FAILURE IN CREATING TABLE(S)\n");
      HANDLE_ERROR (errhp, status);
      return;
    }
  else
    {
      printf ("************************************************\n");
      printf ("MileStone : Table(s) Successfully created\n");
      printf ("************************************************\n");
    }
  HANDLE_ERROR (errhp, OCIHandleFree (stmhp, OCI_HTYPE_STMT));
}

/* prepare data for our examples */
void
prepare_data (OCISvcCtx * svchp, OCIError * errhp, OCIEnv * envhp)
{
  OCIStmt *stmhp;
  text *insstmt =
    (text *)
    "INSERT INTO OCISPEC (ename,mgr, hiredate) VALUES (:ENAME,:MGR, CAST(:HIREDATE
AS timestamp))";
  OCIBind *bnd1p = (OCIBind *) 0;     /* the first bind handle   */
  OCIBind *bnd2p = (OCIBind *) 0;     /* the second bind handle */
  OCIBind *bnd3p = (OCIBind *) 0;     /* the third bind handle   */
  ub4 status = OCI_SUCCESS;
  int i = 0;

  char *ename[3] = { "SMITH", "ALLEN", "KING" };

  sword mgr[] = { 7886, 7110, 7221 };

  char *date_buffer[3] = { "02-AUG-07", "02-APR-07", "02-MAR-07" };
```

```
/*
 * Initialize & Allocate Statement Handle
 */
HANDLE_ERROR (errhp,
    OCIHandleAlloc (envhp, (dvoid **) & stmhp,
         OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

/*
 * Prepare the insert statement
 */
HANDLE_ERROR (errhp,
    OCIStmtPrepare (stmhp, errhp, insstmt,
         (ub4) strlen ((char *) insstmt),
         (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

/*
 * In this loop we will bind data from the arrays to insert multi
 * rows in the database a more elegant and better way to do this is
 * to use Array Binding (Batch Inserts). POLARDB OCI Replacement
 * Library WILL support Array Bindings even if it is not used here
 * right now
 */
for (i = 0; i < 3; i++)
 {
  /*
   * Bind Variable for ENAME
   */
  HANDLE_ERROR (errhp,
      OCIBindByName (stmhp, &bnd1p, errhp, (text *) ":ENAME",
          -1, (dvoid *) ename[i],
          (sb4) strlen (ename[i]) + 1, SQLT_STR,
          (dvoid *) 0, 0, (ub2 *) 0, (ub4) 0,
          (ub4 *) 0, OCI_DEFAULT));

  /*
   * Bind Variable for MGR
   */
  HANDLE_ERROR (errhp,
      OCIBindByName (stmhp, &bnd2p, errhp, (text *) ":MGR",
          -1, (dvoid *) & mgr[i], sizeof (mgr[i]),
          SQLT_INT, (dvoid *) 0, 0, (ub2 *) 0,
          (ub4) 0, (ub4 *) 0, OCI_DEFAULT));

  /*
   * Bind Variable for HIREDATE
   */
  HANDLE_ERROR (errhp,
      OCIBindByName (stmhp, &bnd3p, errhp, (text *) ":HIREDATE",
          -1, (dvoid *)  date_buffer[i],
          strlen(date_buffer[i])+1, SQLT_STR, (dvoid *) 0, 0,
          (ub2 *) 0, (ub4) 0, (ub4 *) 0,
          OCI_DEFAULT));

  /*
   * Execute the statement and insert data
   */
  if ((status = OCIStmtExecute (svchp, stmhp, errhp,
          (ub4) 1, (ub4) 0, NULL, NULL, OCI_DEFAULT)) < OCI_SUCCESS)
 {
  printf ("FAILURE IN INSERTING DATA\n");
  HANDLE_ERROR (errhp, status);
  return;
 }
```

```
    }

  OCITransCommit (svchp, errhp, (ub4) 0);
  printf ("*************************************************\n");
  printf
    ("MileStone : Data Sucessfully inserted \n & Committed via Transaction\n");
  printf ("*************************************************\n");
  HANDLE_ERROR (errhp, OCIHandleFree (stmhp, OCI_HTYPE_STMT));

}

/* Create Stored procedures and functions to be used in this example */
void
create_stored_procs (OCISvcCtx * svchp, OCIError * errhp, OCIEnv * envhp)
{
  /*
   * This function created 2 stored procedures and one stored function
   * 1. StoredProcedureSample1 - is to exhibit exeucting procedure and
   * recieving values from an IN OUT parameter 2.
   * StoredProcedureSample2 - is to exhibit executing procedure and
   * recieving values from an OUT parameter 3. StoredProcedureSample3 -
   * is to exhibit executing a function and recieving the value
   * returned by the function in a Callable Statement way
   */
  OCIStmt *stmhp;
  OCIStmt *stmhp2;
  OCIStmt *stmhp3;

  text *create_statement =
    (text *)"CREATE OR REPLACE PROCEDURE StoredProcedureSample1\n (mgr1 int,
ename1 IN OUT varchar2)\n   is\nbegin\ninsert into ocispec (mgr, ename) values (7990,'
STOR1');\nename1 := 'Successful';\n end;\n";

  text *create_statement2 =
    (text *)"CREATE OR REPLACE PROCEDURE StoredProcedureSample2\n(mgr1 int, ename1
 varchar2,eout1 OUT varchar2)\nis\nbegin\ninsert into ocispec(mgr,ename) values (
7991, 'STOR2');\neout1 := 'Successful';\n    end;";

  text *create_statement3 =
    (text *)"CREATE OR REPLACE FUNCTION f1\nRETURN VARCHAR2\nis\nv_Sysdate DATE;\
nv_charSysdate VARCHAR2(20);\nbegin\nSELECT TO_CHAR(SYSDATE, 'dd-mon-yyyy') into
 v_charSysdate FROM DUAL;\n    return(v_charSysdate);\nend;";



  ub4 status = OCI_SUCCESS;

  /*
   * Initialize & Allocate Statement Handles
   */
  HANDLE_ERROR (errhp,
        OCIHandleAlloc (envhp, (dvoid **) & stmhp,
                OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));
  HANDLE_ERROR (errhp,
        OCIHandleAlloc (envhp, (dvoid **) & stmhp2,
                OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));
  HANDLE_ERROR (errhp,
        OCIHandleAlloc (envhp, (dvoid **) & stmhp3,
                OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  /*
   * Prepare the Create statements
   */
```

```
HANDLE_ERROR (errhp,
    OCIStmtPrepare (stmhp, errhp,
        create_statement,
        strlen ((const char *) create_statement),
        OCI_NTV_SYNTAX, OCI_DEFAULT));
HANDLE_ERROR (errhp,
    OCIStmtPrepare (stmhp2, errhp, create_statement2,
        strlen ((const char *) create_statement2),
        OCI_NTV_SYNTAX, OCI_DEFAULT));
HANDLE_ERROR (errhp,
    OCIStmtPrepare (stmhp3, errhp, create_statement3,
        strlen ((const char *) create_statement3),
        OCI_NTV_SYNTAX, OCI_DEFAULT));

/*
 * Execute the Create Statement SampleProcedure1
 */
if ((status = OCIStmtExecute (svchp, stmhp, errhp,
          (ub4) 1, (ub4) 0, NULL, NULL, OCI_DEFAULT)) < OCI_SUCCESS)
  {
    printf ("FAILURE IN CREATING PROCEDURE 1\n");
    HANDLE_ERROR (errhp, status);
    return;
  }
else
  {
    printf ("*********************************************\n");
    printf ("MileStone : Sample Procedure 1 Successfully created\n");
    printf ("*********************************************\n");

  }

/*
 * Execute the Create Statement Sample Procedure2
 */
if ((status = OCIStmtExecute (svchp, stmhp2, errhp,
          (ub4) 1, (ub4) 0, NULL, NULL, OCI_DEFAULT)) < OCI_SUCCESS)
  {
    printf ("FAILURE IN CREATING PROCEDURE 2\n");
    HANDLE_ERROR (errhp, status);
    return;
  }
else
  {
    printf ("*********************************************\n");
    printf ("MileStone : Sample Procedure 2 Successfully created\n");
    printf ("*********************************************\n");
  }

/*
 * Execute the Create Statement Sample Procedure3
 */
if ((status = OCIStmtExecute (svchp, stmhp3, errhp,
          (ub4) 1, (ub4) 0, NULL, NULL, OCI_DEFAULT)) < OCI_SUCCESS)
  {
    printf ("FAILURE IN CREATING PROCEDURE 3\n");
    HANDLE_ERROR (errhp, status);
    return;
  }
else
  {
    printf ("*********************************************\n");
    printf ("MileStone : Sample Procedure 3 Successfully created\n");
    printf ("*********************************************\n");
    }


  HANDLE_ERROR (errhp, OCIHandleFree (stmhp, OCI_HTYPE_STMT));
  HANDLE_ERROR (errhp, OCIHandleFree (stmhp2, OCI_HTYPE_STMT));
  HANDLE_ERROR (errhp, OCIHandleFree (stmhp3, OCI_HTYPE_STMT));
}

/* select and print data by iterating through resultSet */
void
select_print_data (OCISvcCtx * svchp, OCIError * errhp, OCIEnv * envhp)
{

  /* Statement */
  OCIStmt *stmhp;

  /* Define */
  OCIDefine *define;

  /* Buffer for employee Name */
  char ename_buffer[10] ;

  /* Buffer for mgr */
  sword mgr_buffer;

  /*Buffer for hiredate */
  char hire_date[20];

  /*
   * a simple select statement
   */
  text * sql_statement =
    (text *) "select ename,mgr,hiredate from ocispec";

  /*
   * additional local variables
   */

  ub4 rows = 1;
  ub4 fetched = 1;
  ub4 status = OCI_SUCCESS;

  sb2 null_ind_ename = 0;

  /* null indicator for ename */
  sb2 null_ind_mgr = 0;

  /* null indicator for mgr */
  sb2 null_ind_hiredate = 0;

  /* null indicator for hiredate */

  /*
   * Now we are going to start the Milestone of a Simple Query of the
   * database and loop through the resultSet This would include
   * following steps
   *
   * 1. Initialize and Allocate the Statement Handle 2. Prepare the
   * Statement 3. Define Output variables to recieve the output of the
   * select statement 4. Execute the statement 5. Fetch the resultset
   * and Print values
   *
   */
memset( ename_buffer, 0, sizeof(ename_buffer) );
memset( hire_date, 0, sizeof(hire_date) );
  /*
   * Initialize & Allocate Statement Handle
   */

  HANDLE_ERROR (errhp,
       OCIHandleAlloc (envhp, (dvoid **) & stmhp,
               OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  /*
   * Prepare the statement
   */

  HANDLE_ERROR (errhp,
       OCIStmtPrepare (stmhp, errhp,
               sql_statement,
               strlen ((const char *) sql_statement),
               OCI_NTV_SYNTAX, OCI_DEFAULT));


  /*
   * Bind a String (OCIString) variable on position 1. Datatype used
   * SQLT_VST
   */
  HANDLE_ERROR (errhp,
       OCIDefineByPos (stmhp, &define, errhp,
               (ub4) 1, ename_buffer, 10,
               (ub2) SQLT_STR, &null_ind_ename, 0, 0,
               OCI_DEFAULT));

  /*
   * Bind a Number (OCINumber) variable on position 2. Datatype used
   * SQLT_VNU
   */
  HANDLE_ERROR (errhp,
       OCIDefineByPos (stmhp, &define, errhp,
               (ub4) 2, &mgr_buffer, sizeof (sword),
               (ub2) SQLT_INT, &null_ind_mgr, 0, 0,
               OCI_DEFAULT));

  /*
   * Bind a Date (OCIDate) variable on position 3. Datatype used
   * SQLT_ODT
   */
  HANDLE_ERROR (errhp,
       OCIDefineByPos (stmhp, &define, errhp,
               (ub4) 3, hire_date, 20,
               (ub2) SQLT_STR, &null_ind_hiredate, 0, 0,
               OCI_DEFAULT));



  /*
   * Execute the simple SQL Statement
   */
  status = OCIStmtExecute (svchp, stmhp, errhp,
              rows, (ub4) 0, NULL, NULL, OCI_DEFAULT);


  /*
   * Print the Resultset
   */
  if (status == OCI_NO_DATA)
    {
      /*
       * indicates didn't fetch anything (as we're not array
       * fetching)
       */
      fetched = 0;
    }
  else
    {
      HANDLE_ERROR (errhp, status);
    }

  if (fetched)
    {
      /*
       * print string
       */
      if (null_ind_ename == -1)
    printf ("name -> [NULL]\t");
      else
    printf ("name -> [%s]\t",  ename_buffer);


      /*
       * print number by converting it into int
       */
      if (null_ind_mgr == -1)
    printf ("mgr -> [NULL]\n");
      else
    {
      printf ("mgr -> [%d]\n", mgr_buffer);
    }

      if (null_ind_hiredate == -1)
    printf ("hiredate -> [NULL]\n");
      else
    {
      printf ("hiredate -> [%s]\n",hire_date );
    }

      /*
       * loop through the resultset one by one through
       * OCIStmtFetch()
       */

      /*
       * untill we find nothing
       */
      while (1)
    {
      status = OCIStmtFetch (stmhp, errhp,
                  rows, OCI_FETCH_NEXT, OCI_DEFAULT);
      if (status == OCI_NO_DATA)
       {
         /*
          * indicates couldn't fetch anything
          */
         break;
       }
      else
       {
         HANDLE_ERROR (errhp, status);
       }

      /*
```

```
       * print string
       */
      if (null_ind_ename == -1)
        printf ("name -> [NULL]\t");
      else
        printf ("name -> [%s]\t", ename_buffer);

      /*
       * print number by converting it into int
       */
      if (null_ind_mgr == -1)
        printf ("mgr -> [NULL]\n");
      else
        {
        printf ("mgr -> [%d]\n", mgr_buffer);
        }

      /*
       * print date after converting to text
       */
      if (null_ind_hiredate == -1)
        printf ("hiredate -> [NULL]\n");
      else
        {

          printf ("hiredate -> [%s]\n", hire_date);
        }
    }
    }
  HANDLE_ERROR (errhp, OCIHandleFree (stmhp, OCI_HTYPE_STMT));

}

void
call_stored_proc1 (OCISvcCtx * svchp, OCIError * errhp, OCIEnv * envhp)
{
  OCIStmt *p_sql;
  OCIBind *p_Bind1 = (OCIBind *) 0;
  OCIBind *p_Bind2 = (OCIBind *) 0;

  char field2[20];


  /*
   * char field3[20];
   */
  sword field1 = 3;
  text *mySql = (text *) "Begin StoredProcedureSample1(:MGR, :ENAME); END";

  memset( field2, 0, sizeof(field2) );
  strcpy( field2, "Entry 3" );

  printf ("*************************************************\n");
  printf ("Example 1 - Using an IN OUT Parameter\n");
  printf ("*************************************************\n");


  /*
   * Initialize & Allocate Statement Handle
   */

  HANDLE_ERROR (errhp,
        OCIHandleAlloc (envhp, (dvoid **) & p_sql,
                OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));
  HANDLE_ERROR (errhp,
        OCIStmtPrepare (p_sql, errhp, mySql,
                (ub4) strlen ((char *)mySql), OCI_NTV_SYNTAX,
                OCI_DEFAULT));

  HANDLE_ERROR (errhp,
        OCIBindByPos (p_sql, &p_Bind1, errhp, 1,
                  (dvoid *) & field1, sizeof (sword),
                  SQLT_INT, 0, 0, 0, 0, 0, OCI_DEFAULT));

  HANDLE_ERROR (errhp,
        OCIBindByPos (p_sql, &p_Bind2, errhp, 2,
                  field2, (sizeof (field2)),
                  SQLT_STR, 0, 0, 0, 0, 0, OCI_DEFAULT));

  printf (" Field2 Before:\n");
  printf (" size ---> %d\n", sizeof (field2));
  printf (" length ---> %d\n", strlen (field2));
  printf (" value ---> %s\n", field2);

  HANDLE_ERROR (errhp,
        OCIStmtExecute (svchp, p_sql, errhp, (ub4) 1, (ub4) 0,
                (OCISnapshot *) NULL, (OCISnapshot *) NULL,
                (ub4) OCI_COMMIT_ON_SUCCESS));

  printf (" Field2 After:\n");
  printf (" size ---> %d\n", sizeof (field2));
  printf (" length ---> %d\n", strlen (field2));
  printf (" value ---> %s\n", field2);

  HANDLE_ERROR (errhp, OCIHandleFree (p_sql, OCI_HTYPE_STMT));
}

void
call_stored_proc2 (OCISvcCtx * svchp, OCIError * errhp, OCIEnv * envhp)
{
  OCIStmt *p_sql;
  OCIBind *p_Bind1 = (OCIBind *) 0;
  OCIBind *p_Bind2 = (OCIBind *) 0;
  OCIBind *p_Bind3 = (OCIBind *) 0;

  char field2[20] = "Entry 3";
  char field3[20];
  sword field1 = 3;
  text *mySql =
    (text *) "Begin StoredProcedureSample2(:MGR, :ENAME, :EOUT); END";


  memset( field2, 0, sizeof(field2) );
  strcpy( field2, "Entry 3" );

  memset( field3, 0, sizeof(field3) );


  printf ("*************************************************\n");
  printf ("Example 2 - Using an OUT Parameter\n");
  printf ("*************************************************\n");

  /*
   * Initialize & Allocate Statement Handle
   */

  HANDLE_ERROR (errhp,
      OCIHandleAlloc (envhp, (dvoid **) & p_sql,
          OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  HANDLE_ERROR (errhp,
      OCIStmtPrepare (p_sql, errhp, mySql,
          (ub4) strlen ((char *)mySql), OCI_NTV_SYNTAX,
          OCI_DEFAULT));

  HANDLE_ERROR (errhp,
      OCIBindByPos (p_sql, &p_Bind1, errhp, 1,
            (dvoid *) & field1, sizeof (sword),
            SQLT_INT, 0, 0, 0, 0, 0, OCI_DEFAULT));

  HANDLE_ERROR (errhp,
      OCIBindByPos (p_sql, &p_Bind2, errhp, 2,
            field2, strlen (field2) + 1,
            SQLT_STR, 0, 0, 0, 0, 0, OCI_DEFAULT));

  HANDLE_ERROR (errhp,
      OCIBindByPos (p_sql, &p_Bind3, errhp, 3,
            field3, 20,
            SQLT_STR, 0, 0, 0, 0, 0, OCI_DEFAULT));

  printf (" Field3 Before:\n");
  printf (" size ---> %d\n", sizeof (field3));
  printf (" length ---> %d\n", strlen (field3));
  printf (" value ---> %s\n", field3);

  HANDLE_ERROR (errhp,
      OCIStmtExecute (svchp, p_sql, errhp, (ub4) 1, (ub4) 0,
          (OCISnapshot *) NULL, (OCISnapshot *) NULL,
          (ub4) OCI_COMMIT_ON_SUCCESS));


  printf (" Field3 After:\n");
  printf (" size ---> %d\n", sizeof (field3));
  printf (" length ---> %d\n", strlen (field3));
  printf (" value ---> %s\n", field3);

  HANDLE_ERROR (errhp, OCIHandleFree (p_sql, OCI_HTYPE_STMT));
}

/* drop table(s) required for this example */
void
drop_table (OCISvcCtx * svchp, OCIError * errhp, OCIEnv * envhp)
{
  OCIStmt *stmhp;
  text *statement = (text *)"DROP TABLE OCISPEC";
  ub4 status = OCI_SUCCESS;

  /*
   * Initialize & Allocate Statement Handle
   */
  HANDLE_ERROR (errhp,
      OCIHandleAlloc (envhp, (dvoid **) & stmhp,
          OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  /*
   * Prepare the drop statement
   */
  HANDLE_ERROR (errhp,
      OCIStmtPrepare (stmhp, errhp,
          statement, strlen ((const char *) statement),
          OCI_NTV_SYNTAX, OCI_DEFAULT));
  /*
   * Execute the drop Statement
   */
  if ((status = OCIStmtExecute (svchp, stmhp, errhp,
                   (ub4) 1, (ub4) 0, NULL, NULL, OCI_DEFAULT)) < OCI_SUCCESS)
    {
      printf ("FAILURE IN DROPING TABLE(S)\n");
      HANDLE_ERROR (errhp, status);
      return;
    }
  else
    {
      printf ("********************************************\n");
      printf ("MileStone : Table(s) Successfully Dropped\n");
      printf ("********************************************\n");
    }
  HANDLE_ERROR (errhp, OCIHandleFree (stmhp, OCI_HTYPE_STMT));
}

void
drop_stored_procs (OCISvcCtx * svchp, OCIError * errhp, OCIEnv * envhp)
{
  OCIStmt *stmhp;
  OCIStmt *stmhp2;
  OCIStmt *stmhp3;

  text *create_statement = (text *)"DROP PROCEDURE StoredProcedureSample1";
  text *create_statement2 = (text *)"DROP PROCEDURE StoredProcedureSample2";
  text *create_statement3 = (text *)"DROP FUNCTION  f1";


  ub4 status = OCI_SUCCESS;
  OCITransCommit( svchp, errhp, OCI_DEFAULT );
  /*
   * Initialize & Allocate Statement Handles
   */
  HANDLE_ERROR (errhp,
        OCIHandleAlloc (envhp, (dvoid **) & stmhp,
                OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));
  HANDLE_ERROR (errhp,
        OCIHandleAlloc (envhp, (dvoid **) & stmhp2,
                OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));
  HANDLE_ERROR (errhp,
        OCIHandleAlloc (envhp, (dvoid **) & stmhp3,
                OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  /*
   * Prepare the Create statements
   */

  HANDLE_ERROR (errhp,
        OCIStmtPrepare (stmhp, errhp,
                create_statement,
                strlen ((const char *) create_statement),
                OCI_NTV_SYNTAX, OCI_DEFAULT));
  HANDLE_ERROR (errhp,
        OCIStmtPrepare (stmhp2, errhp, create_statement2,
                strlen ((const char *) create_statement2),
                OCI_NTV_SYNTAX, OCI_DEFAULT));
  HANDLE_ERROR (errhp,
        OCIStmtPrepare (stmhp3, errhp, create_statement3,
                strlen ((const char *) create_statement3),
                OCI_NTV_SYNTAX, OCI_DEFAULT));
  /*
   * Execute the Create Statement SampleProcedure1
   */
  if ((status = OCIStmtExecute (svchp, stmhp, errhp,
                (ub4) 1, (ub4) 0, NULL, NULL, OCI_DEFAULT)) < OCI_SUCCESS)
    {
      printf ("FAILURE IN DROPPING PROCEDURE 1\n");
      HANDLE_ERROR (errhp, status);
      return;
    }
  else
    {
      printf ("********************************************\n");
      printf ("MileStone : Sample Procedure 1 Successfully dropped\n");
      printf ("********************************************\n");
    }

  /*
   * Execute the Create Statement Sample Procedure2
   */
  if ((status = OCIStmtExecute (svchp, stmhp2, errhp,
                (ub4) 1, (ub4) 0, NULL, NULL, OCI_DEFAULT)) < OCI_SUCCESS)
    {
      printf ("FAILURE IN DROPPING PROCEDURE 2\n");
      HANDLE_ERROR (errhp, status);
      return;
    }
  else
    {
      printf ("********************************************\n");
      printf ("MileStone : Sample Procedure 2 Successfully dropped\n");
      printf ("********************************************\n");
    }

  /*
   * Execute the Create Statement Sample Procedure3
   */
  if ((status = OCIStmtExecute (svchp, stmhp3, errhp,
                (ub4) 1, (ub4) 0, NULL, NULL, OCI_DEFAULT)) < OCI_SUCCESS)
    {
      printf ("FAILURE IN DROPPING PROCEDURE 3\n");
      HANDLE_ERROR (errhp, status);
      return;
    }
  else
    {
      printf ("********************************************\n");
      printf ("MileStone : Sample Procedure 3 Successfully dropped\n");
      printf ("********************************************\n");
    }


  HANDLE_ERROR (errhp, OCIHandleFree (stmhp, OCI_HTYPE_STMT));
  HANDLE_ERROR (errhp, OCIHandleFree (stmhp2, OCI_HTYPE_STMT));
  HANDLE_ERROR (errhp, OCIHandleFree (stmhp3, OCI_HTYPE_STMT));

}

/* Clean your mess up */
void
cleanup (OCISvcCtx ** svchp, OCIServer ** srvhp, OCISession ** authp,
     OCIError ** errhp, OCIEnv ** envhp)
{
  /*
   * log off
   */
  HANDLE_ERROR (*errhp, OCISessionEnd (*svchp, *errhp, *authp, OCI_DEFAULT));
  printf ("logged off\n");

  /*
   * detach from server
   */
  HANDLE_ERROR (*errhp, OCIServerDetach (*srvhp, *errhp, OCI_DEFAULT));
  printf ("detached form server\n");

  /*
   * free up handles
   */
  HANDLE_ERROR (*errhp, OCIHandleFree (*authp, OCI_HTYPE_SESSION));
  /* free session handle */
  *authp = 0;
  HANDLE_ERROR (*errhp, OCIHandleFree (*srvhp, OCI_HTYPE_SERVER));
  /* free server handle */
  *srvhp = 0;
  HANDLE_ERROR (*errhp, OCIHandleFree (*svchp, OCI_HTYPE_SVCCTX));
  /* free service context */
  *svchp = 0;
  HANDLE_ERROR (*errhp, OCIHandleFree (*errhp, OCI_HTYPE_ERROR));
  /* free error handle */
  *errhp = 0;
  OCIHandleFree (*envhp, OCI_HTYPE_ENV);
  /* free environment handle */
  *envhp = 0;
  printf ("free'd all handles\n");
}
```

In the preceding sample code, you must replace the following parameters with the connection information of your Apsara PolarDB cluster.

| Parameter | Example | Description |
|---|---|---|
| text *username | (text *) "postgres" | The username of the Apsara PolarDB cluster. |
| text *passwd | (text *) "" | The password of the Apsara PolarDB cluster. |
| text *server | (text *) "//localhost:5432" | The endpoint and port of the Apsara PolarDB cluster. For more information about how to query the endpoint, see #unique_15. |

**Note:**

For more information about the Oracle native OCI driver, see OCI: Introduction.

### Sample code

- Linux

1. Modify the Makefile file to dynamically link to the path where the polaroci.so file is located.

   The following is an example of the Makefile file:

```
#
 ===============================================================================
# Copyright (c) 2004-2012 PolarDB Corporation. All Rights Reserved.
#
 ===============================================================================

# Makefile to build C testcases for OCILib
#

CC=gcc
CFLAGS=-Wall -g -I$(ORACLE_HOME)/ -L $(POLARDBOCI_LIB) -lpolardboci -lpq -
liconv

SAMPLES = polardb_demo

all: $(SAMPLES)

%:%.o
   $(CC) $(CFLAGS) -o $@
clean:
  rm -rf $(SAMPLES)
```

- Link ORACLE_HOME to the directory instantclient_12_1/sdk/include of the oracle oci header file that is downloaded from the driver directory.
- Link POLARDBOCI_LIB to the directory where the libpolardboci.so, libpq.so, and libiconv.so files are located.

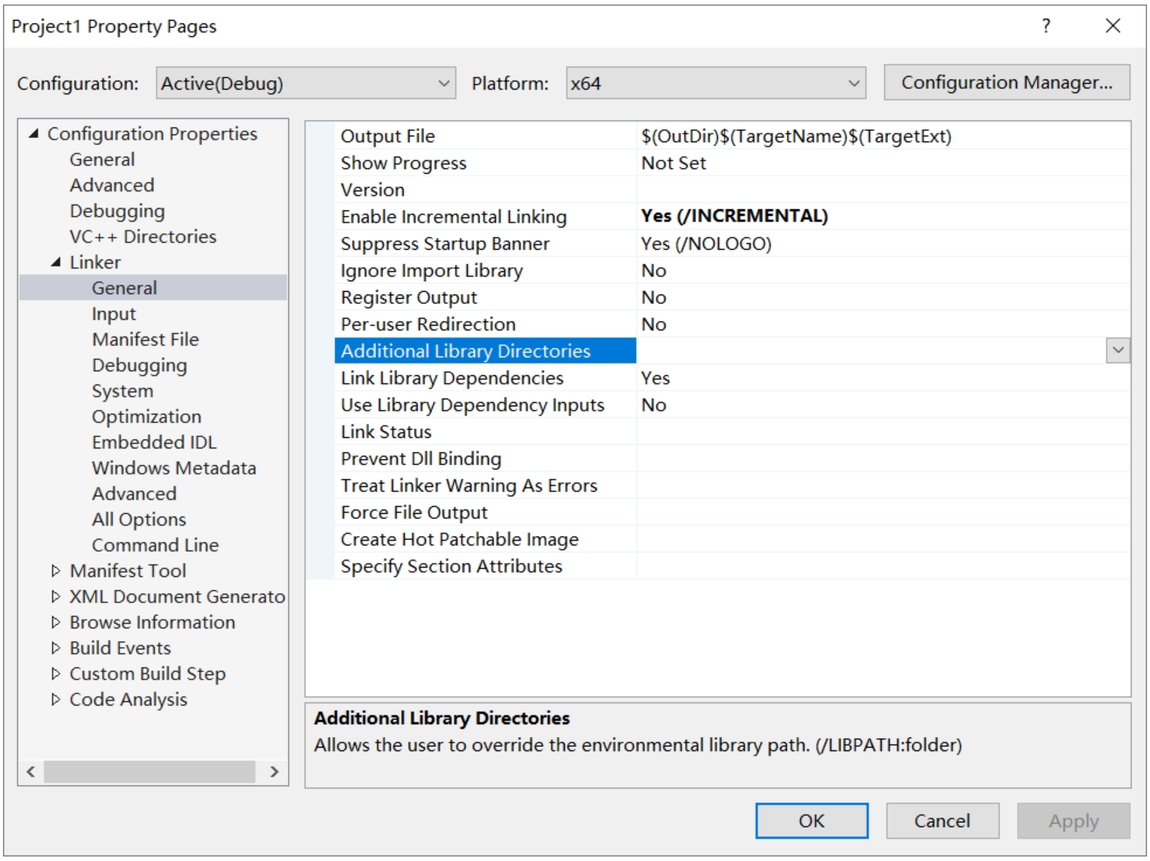
2. Run the following command to compile the code:

```
make
```

- Windows

In this example, Visual Studio is used.

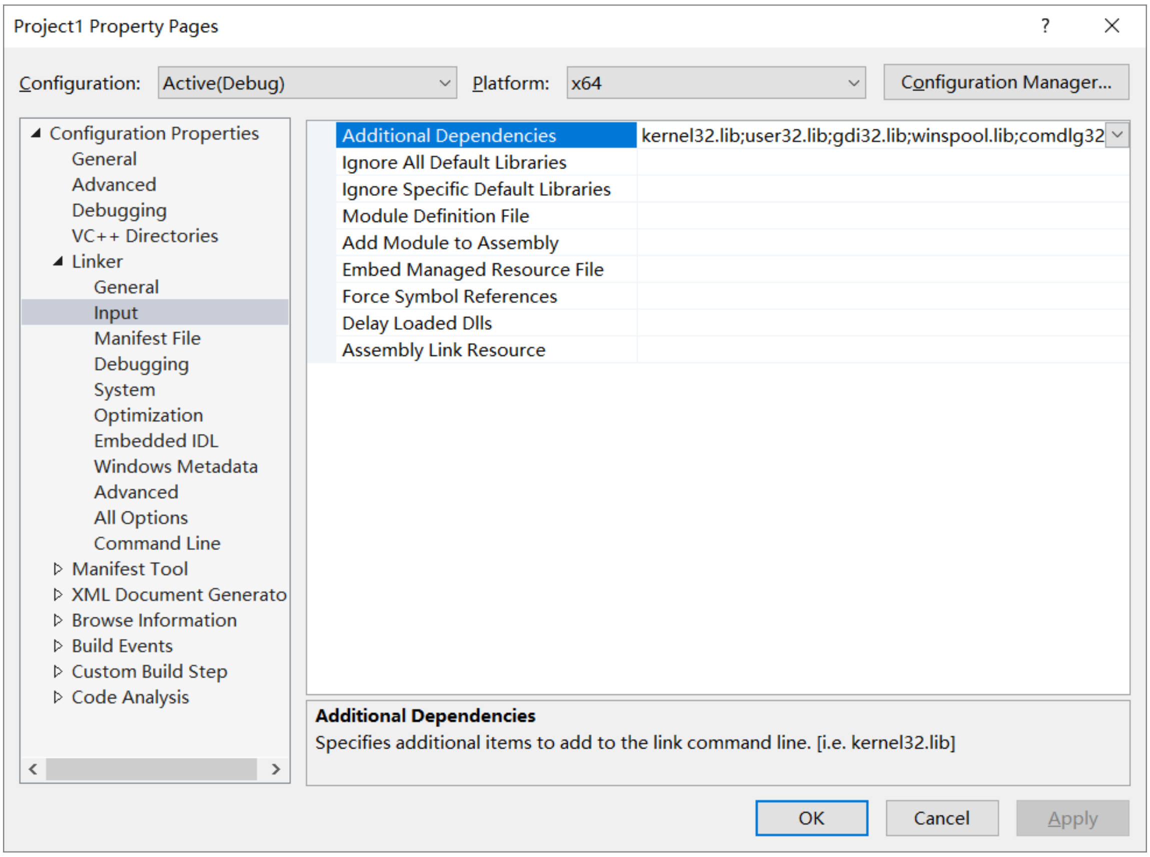
1. Add the path of the oracle oci development package in the driver directory to **Attachment Include Directories**
2. Add the paths of polardboci.dll and polardboci.lib in the driver directory to **Additional Library Directories**.



| | |
|---|---|
| Output File | $(OutDir)$(TargetName)$(TargetExt) |
| Show Progress | Not Set |
| Version | |
| Enable Incremental Linking | **Yes (/INCREMENTAL)** |
| Suppress Startup Banner | Yes (/NOLOGO) |
| Ignore Import Library | No |
| Register Output | No |
| Per-user Redirection | No |
| Additional Library Directories | |
| Link Library Dependencies | Yes |
| Use Library Dependency Inputs | No |
| Link Status | |
| Prevent Dll Binding | |
| Treat Linker Warning As Errors | |
| Force File Output | |
| Create Hot Patchable Image | |
| Specify Section Attributes | |

**Additional Library Directories**
Allows the user to override the environmental library path. (/LIBPATH:folder)

3. Enter polardboci.lib in **Additional Dependencies**.



Configuration: Active(Debug)

Platform: x64

Additional Dependencies: kernel32.lib;user32.lib;gdi32.lib;winspool.lib;comdlg32

**Additional Dependencies**

Specifies additional items to add to the link command line. [i.e. kernel32.lib]

## Example

The system generates the following executable file after polardb_demo is compiled:

```
***********************************************
Milestone  : Logged on as --> 'parallels'
***********************************************

***********************************************
MileStone : Table(s) Successfully created
***********************************************

***********************************************
MileStone : Data Sucessfully inserted
 & Committed via Transaction
***********************************************

***********************************************
MileStone : Sample Procedure 1 Successfully created
***********************************************

***********************************************
MileStone : Sample Procedure 2 Successfully created
***********************************************

***********************************************
MileStone : Sample Procedure 3 Successfully created
***********************************************

name -> [SMITH]    mgr -> [7886]
hiredate -> [2007-08-02 00:00:00]
name -> [ALLEN]    mgr -> [7110]
hiredate -> [2007-04-02 00:00:00]
name -> [KING]    mgr -> [7221]
hiredate -> [2007-03-02 00:00:00]
****************************************************

Example 1 - Using an IN OUT Parameter
****************************************************
```

```
 Field2 Before:
 size ---> 20
 length ---> 7
 value ---> Entry 3
 Field2 After:
 size ---> 20
 length ---> 10
 value ---> Successful
********************************************************
Example 2 - Using an OUT Parameter
********************************************************
 Field3 Before:
 size ---> 20
 length ---> 0
 value --->
 Field3 After:
 size ---> 20
 length ---> 10
 value ---> Successful
**************************************************
MileStone : Table(s) Successfully Dropped
**************************************************
**************************************************
MileStone : Sample Procedure 1 Successfully dropped
**************************************************
**************************************************
MileStone : Sample Procedure 2 Successfully dropped
**************************************************
**************************************************
MileStone : Sample Procedure 3 Successfully dropped
**************************************************
logged off
detached form server
free'd all handles
```

## 3.7 Use PHP to connect to a PolarDB cluster compatible with Oracle

This topic describes how to connect a PHP client to a PolarDB cluster compatible with Oracle.

### Prerequisites

- You have created an account for an ApsaraDB for PolarDB cluster. For more information about how to create an account, see #unique_6.
- You have added the IP address of the host that you want to connect to the ApsaraDB for PolarDB cluster to the whitelist. For more information, see #unique_14.

### Prepare the environment in Windows

1. Download and install WampServer. For more information, see WampServer official website.

2. Launch the PostgreSQL plug-in.
   a) Modify the php.ini file.
   b) Remove semicolons ; from the following code.

      Before you remove semicolons:

      ```
      ;extension=php_pgsql.dll
      ;extension=php_pdo_pgsql.dll
      ```

      After you remove semicolons:

      ```
      extension=php_pgsql.dll
      extension=php_pdo_pgsql.dll
      ```
3. Copy the libpq.dll file from the C:\wamp\bin\php\php5.3.5 directory to the C:\windows\system32\ directory. Note: php5.3.5 is used in this example, and the actual directory is subject to your client version.
4. Restart the Apache service.

**Prepare the environment in Linux**

1. Install the php-pgsql.x86_64 driver.

   ```
   sudo yum install php-pgsql.x86_64
   ```
2. Modify the php.ini file.

   ```
   vim /etc/php.ini
   ```
3. Add the following content to the php.ini file.

   ```
   extension=php_pgsql.so
   ```

**Connect to Apsara PolarDB**

After you prepare the environment in Windows or Linux, you can run a PHP script to connect to the Apsara PolarDB database.

The following sample code shows how to use PHP to connect to the Apsara PolarDB cluster.

```
<? php
 $host    = "host=xxxx";
 $port    = "port=xxxx";
 $dbname   = "dbname=xxxx";
 $credentials = "user=xxxx password=xxxxx";
 $db = pg_connect( "$host $port $dbname $credentials" );
 if(! $db){
  echo "Error : Unable to open database\n";
 } else {
  echo "Opened database successfully\n";
 }
```

```
 $sql =<<<EOF
  select * from pg_roles;
EOF;
 $ret = pg_query($db, $sql);
 if(! $ret){
  echo pg_last_error($db);
 } else {
  echo "Records created successfully\n";
 }
 $results = pg_fetch_all($ret);
 print_r($results);
 pg_close($db);
?>
```

In the preceding sample code, the connection information of Apsara PolarDB consists of parameters, such as host, port, dbname, and credentials, as shown in the following table.

| Parameter | Example | Description |
|---|---|---|
| host | "host=xxxxxx" | The endpoint of the Apsara PolarDB cluster. For more information about how to retrieve the endpoint, see #unique_15. |
| port | "port=1521" | The port of the Apsara PolarDB cluster. Default value: 1521. |
| dbname | "dbname=xxxx" | The name of the database to be connected. |
| credentials | "user=xxx password=xxxx" | The username and password used to log on to the Apsara PolarDB cluster. |

For more information about PHP APIs, see PHP documentation.

# 4 Basic operations

## 4.1 Create a user

This topic describes how to create a database user.

### Syntax

```
CREATE USER name [IDENTIFIED BY password]
```

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the user. |
| password | The password of the user. |

### Description

The CREATE USER statement is used to add a new user to a POLARDB cluster compatible with Oracle.

When the CREATE USER statement is executed, a schema will also be created with the same name as the new user and owned by the new user. Objects with unqualified names created by this user will be created in this schema.

**Note:**

- You must be a database superuser to use this statement.
- The maximum length allowed for the user name and password is 63 characters.

### Examples

Create a user and set the password.

```
CREATE USER user IDENTIFIED BY password;
```

## 4.2 Create a database

This topic describes how to create a database.

### Syntax

```
CREATE DATABASE name
```

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the database to be created. |

### Description

CREATE DATABASE creates a new database.

**Note:**

- To create a database, you must be a superuser or have the special CREATEDB permissions.
- Non-superusers with CREATEDB permissions can only create databases owned by them.
- CREATE DATABASE cannot be executed inside a transaction block.
- Make sure that the disk space is sufficient when you create a new database.

### Examples

```
CREATE DATABASE testdb;
```

## 4.3 Create a schema

This topic describes how to create a schema.

### Syntax

```
CREATE SCHEMA AUTHORIZATION username schema_element [ ... ];
```

### Parameters

| Parameter | Description |
|---|---|
| username | The name of the user who will own the new schema.<br>The schema will be named the same as username.<br>**Note:**<br>• Only superusers may create schemas owned by users other than themselves.<br>• In a POLARDB cluster compatible with Oracle, the role and username must already exist, but the schema may not exist.<br>• In Oracle, the user (equivalently, the schema) must exist. |
| schema_element | An SQL statement defining an object to be created within the schema.<br>CREATE TABLE, CREATE VIEW, and GRANT are accepted as clauses within CREATE SCHEMA. Other kinds of objects may be created in separate statements after the schema is created. |

### Description

This variation of the CREATE SCHEMA statement creates a new schema owned by username and populated with one or more objects. The creation of the schema and objects occur within a single transaction so either all objects are created or none of them including the schema.

A schema is essentially a namespace. It contains named objects such as tables and views whose names may duplicate those of other objects existing in other schemas. Named objects are accessed either by qualifying their names with the schema name as a prefix, or by setting a search path that includes the desired schemas. Unqualified objects are created in the current schema. The schema at the front of the search path can be determined with the CURRENT_SCHEMA function. The search path concept and the CURRENT_SCHEMA function are not compatible with Oracle databases.

CREATE SCHEMA includes sub-statements to create objects within the schema. The sub-statements are treated essentially the same as separate statements issued after creating the schema. All the created objects will be owned by the specified user.

**Note:**

To create a schema, the invoking user must have the CREATE permissions for the current database.

### Examples

```
CREATE SCHEMA AUTHORIZATION enterprisedb
   CREATE TABLE empjobs (ename VARCHAR2(10), job VARCHAR2(9))
   CREATE VIEW managers AS SELECT ename FROM empjobs WHERE job = 'MANAGER'
   GRANT SELECT ON managers TO PUBLIC;
```

## 4.4 Create a table

This topic describes how to create a table.

### Syntax

```
CREATE [ GLOBAL TEMPORARY ] TABLE table_name (
  { column_name data_type [ DEFAULT default_expr ]
  [ column_constraint [ ... ] ] | table_constraint } [, ...]
  )
  [ ON COMMIT { PRESERVE ROWS | DELETE ROWS } ]
  [ TABLESPACE tablespace ]
```

where column_constraint is one of the following values:

```
  [ CONSTRAINT constraint_name ]
  { NOT NULL |
    NULL |
    UNIQUE [ USING INDEX TABLESPACE tablespace ] |
    PRIMARY KEY [ USING INDEX TABLESPACE tablespace ] |
    CHECK (expression) |
    REFERENCES reftable [ ( refcolumn ) ]
      [ ON DELETE action ] }
  [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED |
   INITIALLY IMMEDIATE ]
```

and table_constraint is one of the following values:

```
[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ...] )
    [ USING INDEX TABLESPACE tablespace ] |
  PRIMARY KEY ( column_name [, ...] )
    [ USING INDEX TABLESPACE tablespace ] |
  CHECK ( expression ) |
  FOREIGN KEY ( column_name [, ...] )
      REFERENCES reftable [ ( refcolumn [, ...] ) ]
    [ ON DELETE action ] }
[ DEFERRABLE | NOT DEFERRABLE ]
[ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

**Parameters**

| Parameter | Description |
|---|---|
| GLOBAL TEMPORARY | If this parameter is specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction. For more information, see the ON COMMIT parameter in the following table. Existing permanent tables with the same name are not visible to the current session while the temporary table exists, unless they are referenced with schema-qualified names . In addition, temporary tables are not visible outside the session in which it was created. This aspect of global temporary tables is not compatible with Oracle databases . Any indexes created on a temporary table are automatically temporary as well. |
| table_name | The name of the table to be created.<br>**Note:**<br>The name can be schema-qualified. |
| column_name | The name of a column to be created in the new table. |
| data_type | The data type of the column. This may include array specifiers. |

| Parameter | Description |
|---|---|
| DEFAULT default_expr | The DEFAULT clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression. Subqueries and cross-references to other columns in the current table are not allowed. The data type of the default expression must match the data type of the column.<br>The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, the default is null. |
| CONSTRAINT constraint_name | An optional name for a column or table constraint. If not specified, the system generates a name. |
| NOT NULL | The column is not allowed to contain null values. |
| PRIMARY KEY - column constraint | The primary key constraint specifies that a column or columns of a table may contain unique, non-duplicate, and non-null values. Technically, PRIMARY KEY is merely a combination of UNIQUE and NOT NULL. However, identifying a set of columns as the primary key also provides metadata about the design of the schema, as a primary key implies that other tables may rely on this set of columns as a unique identifier for rows.<br>**Note:**<br>• Only one primary key can be specified for a table, whether as a column constraint or a table constraint.<br>• The primary key constraint must name a set of columns that is different from other sets of columns named by any unique constraint defined for the same table. |
| PRIMARY KEY ( column_name [, ...] ) - table constraint | |

| Parameter | Description |
|---|---|
| CHECK (expression) | The CHECK clause specifies an expression that produces a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE or unknown succeed. If any row of an insert or update operation produces a FALSE result, an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint must only reference that value of the column, while an expression appearing in a table constraint may reference multiple columns.<br>CHECK expressions cannot contain subqueries nor refer to variables other than columns of the current row. |

| Parameter | Description |
|---|---|
| REFERENCES reftable [ ( refcolumn ) ] [ ON DELETE action ] - column constraint | These clauses specify a foreign key constraint, which requires that a group of one or more columns of the new table must only contain values that match values in the referenced columns of some row of the referenced table. If refcolumn is omitted, the primary key of the reftable is used . The referenced columns must be the columns of a unique or primary key constraint in the referenced table.<br>In addition, when the data in the referenced columns is changed, certain actions are performed on the data in the columns of this table. The ON DELETE clause specifies the action to perform when a referenced row in the referenced table is being deleted. Referential actions cannot be deferred even if the constraint is deferrable. Here are the following possible actions for each clause:<br>• CASCADE: deletes any rows referencing the deleted row , or update the value of the referencing column to the new value of the referenced column.<br>• SET NULL: sets the referencing columns to NULL.<br>If the referenced columns are changed frequently, you can add an index to the foreign key column so that referentia l actions associated with the foreign key column can be performed more efficiently. |
| FOREIGN KEY ( column [, ...] ) REFERENCES reftable [ ( refcolumn [, ...] ) ] [ ON DELETE action ] - table constraint | |
| DEFERRABLE NOT | This controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediatel y after each statement is executed. Checking of constraint s that are deferrable may be postponed until the end of the transaction by using the SET CONSTRAINTS statement . NO DEFERRABLE is the default value. Only foreign key constraints accept this clause. All other constraint types are not deferrable. |
| DEFERRABLE | |

| Parameter | Description |
|---|---|
| INITIALLY IMMEDIATE | If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is INITIALLY IMMEDIATE, it is checked after each statement. This is the default. If the constraint is INITIALLY DEFERRED, it is checked only at the end of the transaction. The constraint check time can be altered by using the SET CONSTRAINTS statement. |
| INITIALLY DEFERRED | |
| ON COMMIT | The behavior of temporary tables at the end of a transaction block can be controlled using ON COMMIT. The two options are:<br>• PRESERVE ROWS: No special action is taken at the ends of transactions. This is the default behavior. Note that this aspect is not compatible with Oracle databases. The Oracle default is DELETE ROWS.<br>• DELETE ROWS: All rows in the temporary table will be deleted at the end of each transaction block. Essentially , an automatic TRUNCATE is done at each commit. |
| TABLESPACE tablespace | The tablespace is the name of the tablespace in which the new table is to be created. If not specified, the default tablespace is used. If default_tablespace is an empty string , the default tablespace of the database is used. |
| USING INDEX TABLESPACE tablespace | This clause allows selection of the tablespace in which the index associated with a UNIQUE or PRIMARY KEY constraint will be created. If not specified, the default tablespace is used. If default_tablespace is an empty string, the default tablespace of the database is used. |

**Description**

CREATE TABLE creates a new, initially empty table in the current database. The table will be owned by the user who executes the statement.

If a schema name is given, for example, CREATE TABLE myschema.mytable ..., then the table is created in the specified schema. Otherwise it is created in the current schema. Temporary tables exist in a special schema, so a schema name may not be given when you create a temporary table. The table name must be distinct from the name of any other table , sequence, index, or view in the same schema.

CREATE TABLE also automatically creates a data type that represents the composite type corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type in the same schema.

A table cannot have more than 1,600 columns. In practice, the effective limit is lower because of tuple-length constraints.

The optional constraint clauses specify constraints or tests that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table in various ways. There are two ways to define constraints:

- A column constraint is defined as part of a column definition.
- A table constraint definition is not tied to a particular column, and it can encompass more than one column.

Every column constraint can also be written as a table constraint. A column constraint is only a notational convenience if the constraint only affects one column.

**Note:**

POLARDB compatible with Oracle automatically creates an index for each unique constraint and primary key constraint to enforce the uniqueness. Thus, it is not necessary to create an explicit index for primary key columns.

### Examples

Create the table dept and table emp:

```
CREATE TABLE dept (
    deptno        NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname         VARCHAR2(14),
    loc           VARCHAR2(13)
);
CREATE TABLE emp (
    empno         NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename         VARCHAR2(10),
    job           VARCHAR2(9),
    mgr           NUMBER(4),
    hiredate      DATE,
    sal           NUMBER(7,2),
    comm          NUMBER(7,2),
    deptno        NUMBER(2) CONSTRAINT emp_ref_dept_fk
                  REFERENCES dept(deptno)
);
```

Define a unique table constraint for the table dept. Unique table constraints can be defined on one or more columns of the table.

```
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc             VARCHAR2(13)
);
```

Define a check column constraint:

```
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
```

Define a check table constraint:

```
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno),
    CONSTRAINT new_emp_ck CHECK (ename IS NOT NULL AND empno > 7000)
);
```

Define a primary key table constraint for the table jobhist. Primary key table constraints can be defined on one or more columns of the table.

```
CREATE TABLE jobhist (
    empno           NUMBER(4) NOT NULL,
    startdate       DATE NOT NULL,
    enddate         DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2),
    chgdesc         VARCHAR2(80),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate)
);
```

This assigns a literal constant default value for the job column and makes the default value of hiredate be the date at which the row is inserted.

```
CREATE TABLE emp (
    empno        NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename        VARCHAR2(10),
    job          VARCHAR2(9) DEFAULT 'SALESMAN',
    mgr          NUMBER(4),
    hiredate     DATE DEFAULT SYSDATE,
    sal          NUMBER(7,2),
    comm         NUMBER(7,2),
    deptno       NUMBER(2) CONSTRAINT emp_ref_dept_fk
                 REFERENCES dept(deptno)
);
```

Create the table dept in tablespace diskvol1:

```
CREATE TABLE dept (
    deptno       NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname        VARCHAR2(14),
    loc          VARCHAR2(13)
) TABLESPACE diskvol1;
```

## 4.5 Delete a table

This topic describes how to delete a table.

### Syntax

```
DROP TABLE name [CASCADE | RESTRICT | CASCADE CONSTRAINTS]
```

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the table to drop. The name can be schema-qualified. |

### Description

DROP TABLE removes tables from the database. Only the owner of the table can destroy a table.

The DROP TABLE statement always removes any indexes, rules, triggers, and constraints that exist for the target table.

**Note:**

Include the RESTRICT keyword to specify that the server must refuse to drop the table if any objects depend on it. This is the default behavior. The DROP TABLE statement displays an error if any objects depend on the table.

Include the CASCADE clause to drop any objects that depend on the table.

Include the CASCADE CONSTRAINTS clause to specify that POLARDB compatible with Oracle must drop any dependent constraints that exclude other object types on the specified table.

### Examples

The following statement drops a table named emp that has no dependencies:

```
DROP TABLE emp;
```

The outcome of a DROP TABLE statement varies depending on whether the table has any dependencies. You can control the outcome by specifying a drop behavior. For example, if you create two tables named orders and items, where the items table is dependent on the orders table:

```
CREATE TABLE orders
  (order_id int PRIMARY KEY, order_date date, ...) ;
CREATE TABLE items
  (order_id REFERENCES orders, quantity int, ...) ;
```

POLARDB compatible with Oracle performs one of the following actions when dropping the orders table, depending on the drop behavior that you specify:

- If you specify `DROP TABLE orders RESTRICT`, POLARDB compatible with Oracle will report an error.
- If you specify `DROP TABLE orders CASCADE`, POLARDB compatible with Oracle will drop the orders table and the items table.
- If you specify `DROP TABLE orders CASCADE CONSTRAINTS`, POLARDB compatible with Oracle will drop the orders table and remove the foreign key specification from the items table, but not drop the items table.

## 4.6 Create a view

This topic describes how to create a view.

### Syntax

```
CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ]
```

```
  AS query
```

### Parameters

| Parameter | Description |
|---|---|
| name | The name of a view to be created. The name can be schema-qualified. |
| column_name | An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query. |
| query | A SELECT statement provides the columns and rows of the view. |

### Description

CREATE VIEW defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

CREATE OR REPLACE VIEW is similar, but if a view of the same name already exists, it is replaced.

If a schema name is given by using the CREATE VIEW myschema.myview ... statement, the view is created in the specified schema. Otherwise, it is created in the current schema. The view name must be distinct from the name of any other view, table, sequence, or index in the same schema.

**Note:**

- Views are read-only. The system will not allow an insert, update, or delete operation on a view. You can get the effect of an updatable view by creating rules such as rewriting inserts on the view into appropriate actions on other tables. For information about the CREATE RULE statement, see the Postgres Plus documentation set.
- Access to tables referenced in the view is determined by permissions of the view owner. However, functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore, the user of a view must have permissions to call all functions used by the view.

### Examples

Create a view consisting of all employees in department 30:

```
CREATE VIEW dept_30 AS SELECT * FROM emp WHERE deptno = 30;
```

## 4.7 Create a materialized view

This topic describes how to create a materialized view.

### Syntax

```
CREATE MATERIALIZED VIEW name
[build clause][create mv refresh] AS subquery
```

Where build_clause is:

```
BUILD {IMMEDIATE | DEFERRED}
```

Where create_mv_refresh is:

```
REFRESH [COMPLETE] [ON DEMAND]
```

### Parameters

| Parameter | Description |
|---|---|
| name | The name of a view to be created. The name can be schema-qualified. |
| subquery | A SELECT statement that specifies the contents of the view. For more information about valid queries, see SELECT. |
| build clause | Include a build_clause to specify when the view is populated. Specify BUILD IMMEDIATE, or BUILD DEFERRED:<br>• BUILD IMMEDIATE instructs the server to populate the view immediately. This is the default behavior.<br>• BUILD DEFERRED instructs the server to populate the view at a later time during a REFRESH operation. |

| Parameter | Description |
|---|---|
| create mv refresh | Include the create_mv_refresh clause to specify when the contents of a materialized view must be updated. The clause contains the REFRESH keyword followed by COMPLETE and/or ON DEMAND, where:<br>• COMPLETE instructs the server to discard the current content and reload the materialized view by executing the defining query of the view when the materialized view is refreshed.<br>• ON DEMAND instructs the server to refresh the materialized view on demand by calling the DBMS_MVIEW package or by calling the Postgres REFRESH MATERIALIZED VIEW statement. This is the default behavior. |

### Description

CREATE MATERIALIZED VIEW defines a view of a query that is not updated each time the view is referenced in a query. By default, the view is populated when the view is created. You can include the BUILD DEFERRED keywords to delay the population of the view.

A materialized view can be schema-qualified. If you specify a schema name when executing the CREATE MATERIALIZED VIEW statement, the view will be created in the specified schema. The view name must be distinct from the name of any other view, table, sequence, or index in the same schema.

**Note:**

- Materialized views are read-only. The server will not allow an INSERT, UPDATE, or DELETE operation on a view.
- Access to tables referenced in the view is determined by permissions of the view owner . The user of a view must have permissions to call all functions used by the view.
- For more information about the Postgres REFRESH MATERIALIZED VIEW statement, see the PostgreSQL Core Documentation available at: https://www.postgresql.org/docs/11/sql-refreshmaterializedview.html.

### Examples

The following statement creates a materialized view named dept_30:

```
CREATE MATERIALIZED VIEW dept_30 BUILD IMMEDIATE AS SELECT * FROM emp WHERE
deptno = 30;
```

## 4.8 Create an index

This topic describes how to create an index.

### Syntax

```
CREATE [ UNIQUE ] INDEX name ON table
  ( { column | ( expression ) } )
  [ TABLESPACE tablespace ]
```

### Parameters

| Parameter | Description |
|---|---|
| UNIQUE | Causes the system to check for duplicate values in the table when the index is created if data already exist and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error. |
| name | The name of the index to be created. No schema name can be included here. The index is always created in the same schema as its parent table. |
| table | The name of the table to be indexed. The name can be schema-qualified. |
| column | The name of a column in the table. |
| expression | An expression based on one or more columns of the table . The expression usually must be written with surrounding parentheses, as described in the syntax. However, the parentheses may be omitted if the expression has the form of a function call. |
| tablespace | The tablespace in which to create the index. If not specified , default_tablespace is used. If default_tablespace is an empty string, the default tablespace of the database is used. |

### Description

CREATE INDEX constructs an index on the specified table. Indexes are primarily used to enhance database performance.

The key fields for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified to create multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on UPPER(col) can allow the clause `WHERE UPPER(col) = 'JIM'` to use an index.

POLARDB compatible with Oracle provides the B-tree index method. The B-tree index method is an implementation of Lehman-Yao high-concurrency B-trees.

Indexes are not used for IS NULL clauses by default.

All functions and operators used in an index definition must be immutable, that is, their results must depend only on their arguments and never on any outside influence such as the contents of another table or the current time. This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression , remember to mark the function immutable when you create it.

If you create an index on a partitioned table, the CREATE INDEX statement does not propagate indexes to the subpartitions of the table.

- If you specify the name of the partitioned root, all indexes of partitions and subpartitions of the table are created.
- If you specify the name of the partitioned backup table, all indexes of subpartitions in the partition of the table are created.
- If you specify the name of the subpartitioned backup table, only the index of the subpartition of the table is created.

**Note:**

Up to 32 fields may be specified in a multicolumn index.

### Examples

To create a B-tree index on the ename column in the table emp:

```
CREATE INDEX name_idx ON emp (ename);
```

To create the same index as the preceding index, but have it reside in the index_tblspc tablespace:

```
CREATE INDEX name_idx ON emp (ename) TABLESPACE index_tblspc;
```

## 4.9 Create and use a sequence

This topic describes how to define a new sequence generator.

### Syntax

```
CREATE SEQUENCE name [ INCREMENT BY increment ]
  [ { NOMINVALUE | MINVALUE minvalue } ]
  [ { NOMAXVALUE | MAXVALUE maxvalue } ]
  [ START WITH start ] [ CACHE cache | NOCACHE ] [ CYCLE ]
```

### Parameters

| Parameter | Description |
|---|---|
| name | The name (optionally schema-qualified) of the sequence to be created. |
| increment | The optional clause INCREMENT BY increment specifies the value to add to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1. |
| NOMINVALUE \| MINVALUE minvalue | The optional clause MINVALUE minvalue determines the minimum value a sequence can generate. If this clause is not supplied, then default values will be used. The default values are 1 and $-2^{63}$ -1 for ascending and descending sequences respectively. Note that the keyword NOMINVALUE may be used to set this behavior to the default. |

| Parameter | Description |
|---|---|
| NOMAXVALUE \| MAXVALUE maxvalue | The optional clause MAXVALUE maxvalue determines the maximum value for the sequence. If this clause is not supplied, then default values will be used. The default values are $2^{63}$ -1 and 1 for ascending and descending sequences respectively. Note that the keyword NOMAXVALUE may be used to set this behavior to the default. |
| start | The optional clause START WITH start allows the sequence to begin anywhere. The default starting value is minvalue for ascending sequences and maxvalue for descending ones. |
| cache | The optional clause CACHE cache specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time such as NOCACHE), and this is also the default. |
| CYCLE | The CYCLE option allows the sequence to wrap around when the maxvalue or minvalue has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the minvalue or maxvalue respectively. If the default value CYCLE is omitted, any calls to NEXTVAL after the sequence has reached its maximum value will return an error. Note that the keyword NO CYCLE may be used to obtain the default behavior, however, this keyword is not compatible with Oracle databases. |

**Description**

The CREATE SEQUENCE statement is used to define a new sequence generator. This involves creating and initializing a new special single-row table with the name parameter. The generator will be owned by the user issuing the statement.

If a schema name is given then the sequence is created in the specified schema, otherwise it is created in the current schema. The sequence name must be distinct from the name of any other sequence, table, index, or view in the same schema.

**Note:**

Sequences are based on big integer arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807). On some earlier platforms, there may be no compiler support for eight-byte integers, in which case sequences use regular INTEGER arithmetic (range -2147483648 to +2147483647).

Unexpected results may be obtained if a cache setting greater than one is used for a sequence object that will be used concurrently by multiple sessions. Unexpected results may be obtained if a cache setting greater than one is used for a sequence object that will be used concurrently by multiple sessions. Then, the next cache-1 uses of NEXTVAL within that session simply return the preallocated values without touching the sequence object. So, any numbers allocated but not used within a session will be lost when that session ends, resulting in "holes" in the sequence.

Furthermore, although multiple sessions are guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the sessions are considered. For example, with a cache setting of 10, session A might reserve values 1 to 10 and return NEXTVAL=1, then session B might reserve values 11 to 20 and return NEXTVAL=11 before session A has generated NEXTVAL=2. Thus, with a cache setting of one it is safe to assume that NEXTVAL values are generated sequentially. With a cache setting greater than one you only assume that the NEXTVAL values are all distinct, not that they are generated purely sequentially. Also, the last value will reflect the latest value reserved by any session, whether or not it has yet been returned by NEXTVAL.

**Examples**

Create an ascending sequence called serial, starting at 101:

```
CREATE SEQUENCE serial START WITH 101;
```

Select the next number from this sequence:

```
SELECT serial.NEXTVAL FROM DUAL;

 nextval
---------
     101
(1 row)
```

Create a sequence called supplier_seq with the NOCACHE option:

```
CREATE SEQUENCE supplier_seq
    MINVALUE 1
    START WITH 1
    INCREMENT BY 1
    NOCACHE;
```

Select the next number from this sequence:

```
SELECT supplier_seq.NEXTVAL FROM DUAL;

 nextval
---------
       1
(1 row)
```

# 4.10 Create and use a synonym

A synonym is an identifier that can be used to reference another database object in a SQL statement.

### Syntax

Use the CREATE SYNONYM statement to create a synonym. The syntax is as follows:

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema.]syn_name FOR object schema.object
name;CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema.]syn_name
       FOR object_schema.object_name[@dblink_name];
```

To delete a synonym, use the DROP SYNONYM statement. The syntax is as follows:

```
DROP [PUBLIC] SYNONYM [schema.] syn_name;
```

### Parameters

| Parameter | Description |
|---|---|
| syn_name | The name of the synonym. The name of a synonym must be unique within a schema. |
| schema | The name of the schema where the synonym resides.<br>**Note:**<br>If you do not specify a schema name, the synonym is created in the first existing schema in your search path. |
| object_name | The name of the object. |
| object_schema | The name of the schema where the object resides. |

### Description

A synonym is an identifier that can be used to reference another database object in a SQL statement.

A synonym is useful in cases where a database object normally requires full qualification by schema name to be properly referenced in a SQL statement. A synonym defined for that object simplifies the reference to a single, unqualified name.

POLARDB compatible with Oracle supports synonyms for:

- Tables
- Views
- Sequences
- Procedures
- Functions
- Types
- Other synonyms

Neither the referenced schema nor referenced object must exist at the time that you create the synonym. A synonym may refer to a non-existent object or schema. A synonym will become invalid if you drop the referenced object or schema. You must explicitly drop a synonym to remove it.

As with any other schema object, POLARDB compatible with Oracle uses the search path to resolve unqualified synonym names. If you have two synonyms that have the same name, an unqualified reference to a synonym will resolve to the first synonym that has the specified name in the search path. If public is in your search path, you can refer to a synonym in that schema without qualifying that name.

When POLARDB compatible with Oracle executes a SQL statement, the privileges of the current user are checked against the underlying database object of the synonym. If the user does not have the proper permissions for that object, the SQL statement will fail.

### Examples

- Create a synonym

  Include the REPLACE clause to replace an existing synonym definition with a new synonym definition.

  Include the PUBLIC clause to create the synonym in the public schema. Compatible with Oracle databases, the CREATE PUBLIC SYNONYM statement creates a synonym that resides in the public schema:

  ```
  CREATE [OR REPLACE] PUBLIC SYNONYM syn_name FOR object schema.object name;
  ```

  The following statement is a shorthand way to write:

  ```
  CREATE [OR REPLACE] SYNONYM public.syn_name FOR object schema.object name;
  ```

  The following example creates a synonym named personnel that refers to the enterprise db.emp table.

  ```
  CREATE SYNONYM personnel FOR enterprisedb.emp;
  ```

  Unless the synonym is schema qualified in the CREATE SYNONYM statement, the synonym will be created in the first existing schema in your search path. You can view your search path by executing the following statement:

  ```
  SHOW SEARCH_PATH;

      search_path
  -----------------------
   development,accounting
  (1 row)
  ```

  In the example, if a schema named development does not exist, the synonym will be created in the schema named accounting.

  The emp table in the enterprisedb schema can be referenced in any DDL or DML statement, by using the personnel synonym:

  ```
  INSERT INTO personnel VALUES (8142,'ANDERSON','CLERK',7902,'17-DEC-06',1300,NULL
  ,20);

  SELECT * FROM personnel;

   empno | ename  |    job    | mgr  |      hiredate      |   sal   |  comm   | deptno
  -------+--------+-----------+------+--------------------+---------+---------+--------
    7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800.00 |         |     20
    7499 | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600.00 |  300.00 |     30
    7521 | WARD   | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250.00 |  500.00 |     30
    7566 | JONES  | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975.00 |         |     20
    7654 | MARTIN | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 |     30
  ```

```
  7698 | BLAKE    | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850.00 |         |     30
  7782 | CLARK    | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450.00 |         |     10
  7788 | SCOTT    | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000.00 |         |     20
  7839 | KING     | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000.00 |         |     10
  7844 | TURNER   | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500.00 |    0.00 |     30
  7876 | ADAMS    | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100.00 |         |     20
  7900 | JAMES    | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950.00 |         |     30
  7902 | FORD     | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000.00 |         |     20
  7934 | MILLER   | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300.00 |         |     10
  8142 | ANDERSON | CLERK     | 7902 | 17-DEC-06 00:00:00 | 1300.00 |         |     20
(15 rows)
```

- Delete a synonym

  Like any other object that can be schema-qualified, you may have two synonyms with the same name in your search path. To disambiguate the name of the synonym that you are dropping, include a schema name. Unless a synonym is schema qualified in the DROP SYNONYM statement, POLARDB compatible with Oracle deletes the first instance of the synonym it finds in your search path.

  You can optionally include the PUBLIC clause to drop a synonym that resides in the public schema. Compatible with Oracle databases, the DROP PUBLIC SYNONYM statement drops a synonym that resides in the public schema:

  ```
  DROP PUBLIC SYNONYM syn_name;
  ```

  The following example drops the personnel synonym:

  ```
  DROP SYNONYM personnel;
  ```

# 5 Configuration parameters compatible with Oracle databases

## 5.1 edb_redwood_date

If DATE appears as the data type of a column in the statements and the **edb_redwood_date** configuration parameter is set to TRUE, DATE is translated to TIMESTAMP when the table definition is stored in the database. In this case, a time component is also stored in the column along with the date. This rule is consistent with the DATE data type of Oracle.

If **edb_redwood_date** is set to FALSE, the data type of the column in a CREATE TABLE or ALTER TABLE statement remains as a native PostgreSQL DATE data type and is stored in the database. PostgreSQL DATE data type stores only the date without a time component in the column.

DATE can appear as a data type in any other context such as the data type of a variable in an SPL declaration section, or the data type of a formal parameter in an SPL procedure or SPL function, or the return type of an SPL function. In this case, regardless of the setting of **edb_redwood_date**, DATE is always internally translated to a TIMESTAMP and can thus handle an existing time component.

## 5.2 edb_redwood_raw_names

If edb_redwood_raw_names is set to the default value FALSE, database object names, such as table names, column names, trigger names, program names, and user names, appear in uppercase letters when viewed from Oracle catalogs. In addition, quotation marks enclose names that are created with enclosed quotation marks.

If edb_redwood_raw_names is set to TRUE, the database object names are displayed in the way as they are stored in the PostgreSQL system catalogs when viewed from the Oracle catalogs. Thus, names created without enclosed quotation marks appear in lowercase as expected in PostgreSQL. Names created with enclosed quotation marks appear in the way as they are created, but without the quotation marks.

For example, the following user name is created and then a session is started with that user .

```
CREATE USER reduser IDENTIFIED BY password;
edb=# \c - reduser
Password for user reduser:
You are now connected to database "edb" as user "reduser".
```

When you connect to the database as reduser, the following tables are created:

```
CREATE TABLE all_lower (col INTEGER);
CREATE TABLE ALL_UPPER (COL INTEGER);
CREATE TABLE "Mixed_Case" ("Col" INTEGER);
```

When viewed from the Oracle catalog named USER_TABLES, with edb_redwood_raw_names set to the default value FALSE, the names appear in uppercase except for the Mixed_Case name. This name appears in the same way as the name is created and enclosed with quotation marks.

```
edb=> SELECT * FROM USER_TABLES;
 schema_name |  table_name
| tablespace_name | status | temporary
-------------+--------------+-----------------+--------+-----------
 REDUSER     | ALL_LOWER    |                 | VALID  | N
 REDUSER     | ALL_UPPER    |                 | VALID  | N
 REDUSER     | "Mixed_Case" |                 | VALID  | N
(3 rows)
```

When viewed with edb_redwood_raw_names set to TRUE, the names appear in lowercase except for the Mixed_Case name. This name appears in the same way as the name is created, but the name is not enclosed with quotation marks.

```
edb=> SET edb_redwood_raw_names TO true;
SET
edb=> SELECT * FROM USER_TABLES;
 schema_name | table_name |
tablespace_name | status | temporary
-------------+------------+-----------------+--------+-----------
 reduser     | all_lower  |                 | VALID  | N
 reduser     | all_upper  |                 | VALID  | N
 reduser     | Mixed_Case |                 | VALID  | N
(3 rows)
```

These names match the case when viewed from the PostgreSQL pg_tables catalog.

```
edb=> SELECT schemaname, tablename, tableowner FROM pg_tables WHERE
tableowner = 'reduser';
 schemaname | tablename  | tableowner
------------+------------+------------
 reduser    | all_lower  | reduser
 reduser    | all_upper  | reduser
 reduser    | Mixed_Case | reduser
```

```
(3 rows)
```

## 5.3 edb_redwood_strings

In Oracle, when a string is concatenated with a null variable or null column, the result is the original string. However, in PostgreSQL, concatenation of a string with a null variable or null column generates a null result. If the edb_redwood_strings parameter is set to TRUE , the preceding concatenation operation results in the original string in the same way as Oracle does. If the edb_redwood_strings parameter is set to FALSE, the native PostgreSQL behavior is maintained.

The following example illustrates the difference. The sample application introduced in the next section contains a table of employees. This table has a column named comm that is null for most employees. The following query has edb_redwood_string set to FALSE. The concatenation of a null column with non-empty strings generates a final result of null, so only employees that have a commission appear in the query result. The output line for all other employees is null.

```
SET edb_redwood_strings TO off;

SELECT RPAD(ename,10) || ' ' || TO_CHAR(sal,'99,999.99') || ' ' ||
TO_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;

    EMPLOYEE COMPENSATION
----------------------------------

 ALLEN      1,600.00    300.00
 WARD       1,250.00    500.00

 MARTIN     1,250.00  1,400.00




 TURNER     1,500.00       .00




(14 rows)
```

The following example is the same query executed when edb_redwood_strings is set to TRUE. The value of a null column is treated as an empty string. The concatenation of an empty string with a non-empty string generates the non-empty string. This result is consistent with the results generated by Oracle for the same query.

```
SET edb_redwood_strings TO on;
```

```
SELECT RPAD(ename,10) || ' ' || TO_CHAR(sal,'99,999.99') || ' ' ||
TO_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;

     EMPLOYEE COMPENSATION
----------------------------------
 SMITH         800.00
 ALLEN       1,600.00     300.00
 WARD        1,250.00     500.00
 JONES       2,975.00
 MARTIN      1,250.00   1,400.00
 BLAKE       2,850.00
 CLARK       2,450.00
 SCOTT       3,000.00
 KING        5,000.00
 TURNER      1,500.00        .00
 ADAMS       1,100.00
 JAMES         950.00
 FORD        3,000.00
 MILLER      1,300.00
(14 rows)
```

## 5.4 edb_stmt_level_tx

In Oracle, when a runtime error occurs in a SQL statement, all the updates on the database caused by that single statement are rolled back. This is called statement-level transaction isolation. For example, if a single UPDATE statement updates five rows but an attempt to update a sixth row results in an error, the updates to all six rows made by this UPDATE statement are rolled back. The effects of prior SQL statements that have not yet been committed or rolled back are pending until a COMMIT or ROLLBACK statement is executed.

In PostgreSQL, if an error occurs while executing a SQL statement, all the updates on the database since the start of the transaction are rolled back. In addition, the transaction is left in a terminated state and either a COMMIT or ROLLBACK statement must be executed before another transaction can be started.

If edb_stmt_level_tx is set to TRUE, an error does not automatically roll back prior uncommitted database updates, similar to the Oracle behavior. If edb_stmt_level_tx is set to FALSE, an error rolls back uncommitted database updates.

**Notice:**

Set edb_stmt_level_tx to TRUE only when necessary. This setting may decrease the service performance.

As shown in the following example running in PSQL, if edb_stmt_level_tx is set to FALSE, the first INSERT statement is still rolled back after the second INSERT statement is terminated. In PSQL, the statement \set AUTOCOMMIT off must be used. Otherwise every statement commits automatically. This defeats the purpose of this demonstration of the effect of edb_stmt_level_tx.

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO off;

INSERT INTO emp (empno,ename,deptno) VALUES (9001, 'JONES', 40);
INSERT INTO emp (empno,ename,deptno) VALUES (9002, 'JONES', 00);
ERROR:  insert or update on table
"emp" violates foreign key constraint "emp_ref_dept_fk"
DETAIL:  Key (deptno)=(0) is not present in table "dept".

COMMIT;
SELECT empno, ename, deptno FROM emp WHERE empno > 9000;

empno | ename | deptno
-------+-------+--------
(0 rows)
```

In the following example, edb_stmt_level_tx is set to TRUE. The first INSERT statement has not been rolled back after an error occurs in the second INSERT statement. At this point, the first INSERT statement can either be committed or rolled back.

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

INSERT INTO emp (empno,ename,deptno) VALUES (9001, 'JONES', 40);
INSERT INTO emp (empno,ename,deptno) VALUES (9002, 'JONES', 00);
ERROR:  insert or update on table
"emp" violates foreign key constraint "emp_ref_dept_fk"
DETAIL:  Key (deptno)=(0) is not present in table "dept".

SELECT empno, ename, deptno FROM emp WHERE empno > 9000;

empno | ename | deptno
-------+-------+--------
  9001 | JONES |     40
(1 row)

COMMIT;
```

A ROLLBACK statement may be executed instead of the COMMIT statement. In this case, the insert of employee number 9001 is also rolled back.

## 5.5 oracle_home

Before you create a link to an Oracle server, you must direct a PolarDB database compatible with Oracle to the correct Oracle home directory. Set the LD_LIBRARY_PATH environment variable on Linux or the PATH environment variable on Windows to the lib directory of the Oracle client installation directory.

For Windows only, you can set the value of the oracle_home configuration parameter in the postgresql.conf file. The value specified in the oracle_home configuration parameter overwrites the Windows PATH environment variable.

The LD_LIBRARY_PATH environment variable on Linux, or the PATH environment variable or oracle_home configuration parameter on Windows, must be set each time you start the PolarDB database compatible with Oracle.

When you use a Linux service script to start the PolarDB database compatible with Oracle , make sure that LD_LIBRARY_PATH has been set within the service script. This allows the environment variable to take effect when the script invokes the pg_ctl utility to start the database.

For Windows only, to set the oracle_home configuration parameter in the postgresql.conf file, edit the file by adding the following line:

```
oracle_home = 'lib_directory '
```

Substitute the name of the Windows directory that contains oci.dll for lib_directory.

After you set the oracle_home configuration parameter, you must restart the server to make the changes effective. Restart the server from the Windows Services console.

# 6 SQL tutorial

## 6.1 Get started

### 6.1.1 Overview

This topic helps you get started with the SQL language to manage relational database management systems. Basic operations such as creating, populating, querying, and updating tables are described with examples.

More advanced concepts such as views, foreign keys, and transactions are described.

A PolarDB database compatible with Oracle is a relational database management system (RDBMS). The system is used to manage data stored in relations. A relation is essentially a mathematical term for a table. Storing data in tables is a common method of database management. Databases can be organized in several ways. Files and directories on Unix-like operating systems form an example of a hierarchical database. Popular development is based on object-oriented databases.

Each table is a named collection of rows. Each row of a specified table has the same set of named columns and each column is of a specific data type. Columns have a fixed order in each row. However, SQL does not guarantee the order of the rows within the table in any way, even though the rows can be explicitly sorted for display.

Tables are grouped into databases. A collection of databases managed by a PolarDB instance compatible with Oracle constitutes a database cluster.

### 6.1.2 Install a sample database

When you install a PolarDB database compatible with Oracle, a sample database named edb is automatically created. This sample database contains the tables and programs used in this topic. This database runs the sample.sql script in the /usr/edb/as11/share directory.

This script supports the following features:

- Creates the sample tables and programs in a connected database.
- Grants all permissions on tables to the PUBLIC group.

The tables and programs are created in the first schema of the search path in which the current user is authorized to create tables and procedures. You can run the following statement to display the search path:

```
SHOW SEARCH_PATH;
```

You can use the statements in PSQL to alter the search path.

### 6.1.3 Sample database

The sample database stores the information about employees in an organization.

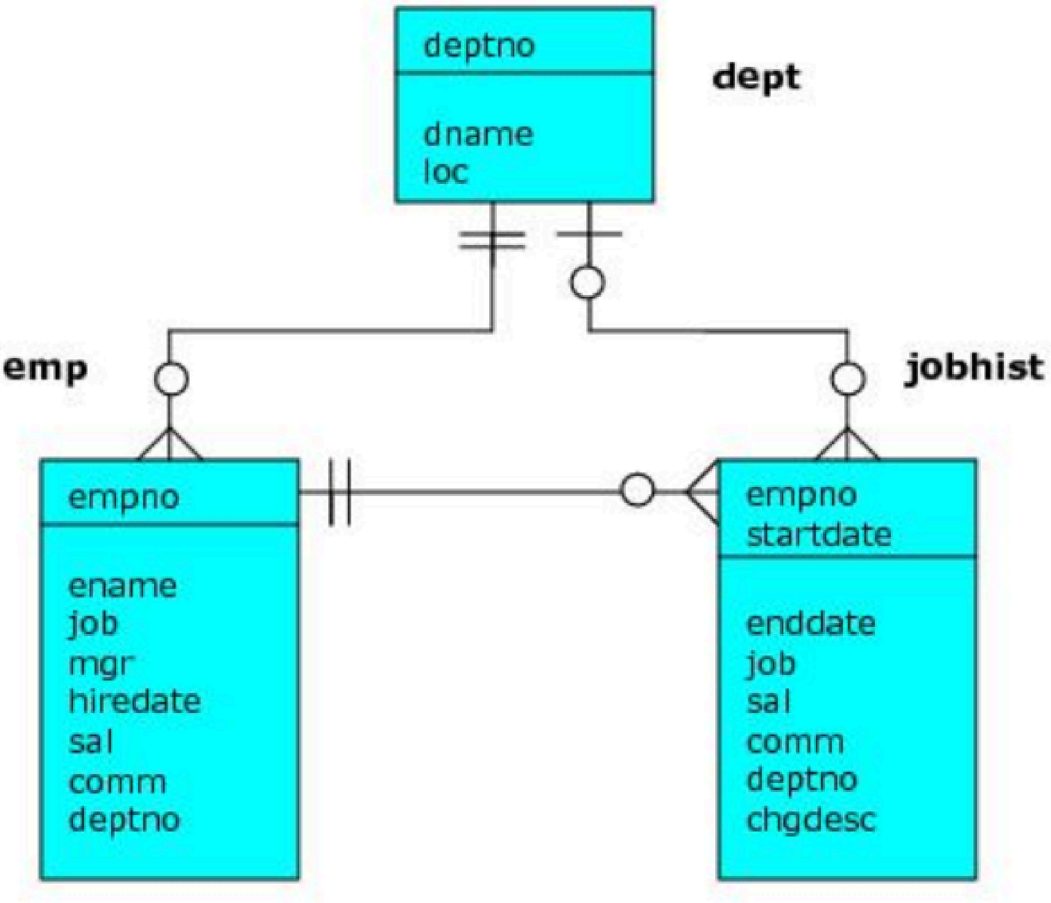
The database contains three types of records: employees, departments, and history of employees.

Each employee has an identification number, a name, a hire date, a salary, and a manager . Some employees earn commissions in addition to their salaries. All employee information is stored in the emp table.

The sample company has bases in multiple regions, so the database tracks the locations of the departments. Each employee is assigned to a department. Each department is identified by a unique department number and a short name. Each department is associated with one location. All department information is stored in the dept table.

The company also tracks the information about jobs held by the employees. Some employees have been working for the company for a long time. They may have held different positions, received raises, or switched departments. If an employee status changes, the company records the end date of the former position for this employee. A new job record is added with the start date and the new job title, department, salary, and the reason for the status change. All employee history is maintained in the jobhist table.

The following figure shows the entity relationship of the sample database tables.



The following example is the sample.sql script.

```
--
-- Script that creates the 'sample' tables, views, procedures,
-- functions, triggers, etc.
--
-- Start new transaction - commit all or nothing
--
BEGIN;
/
--
-- Create and load tables used in the documentation examples.
--
-- Create the 'dept' table
--
CREATE TABLE dept (
   deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
   dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
   loc             VARCHAR2(13)
);
--
-- Create the 'emp' table
--
CREATE TABLE emp (
   empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
   ename           VARCHAR2(10),
   job             VARCHAR2(9),
   mgr             NUMBER(4),
   hiredate        DATE,
   sal             NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
   comm            NUMBER(7,2),
   deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                       REFERENCES dept(deptno)
);
--
--  Create the 'jobhist' table
--
CREATE TABLE jobhist (
   empno           NUMBER(4) NOT NULL,
   startdate       DATE NOT NULL,
   enddate         DATE,
   job             VARCHAR2(9),
   sal             NUMBER(7,2),
   comm            NUMBER(7,2),
   deptno          NUMBER(2),
   chgdesc         VARCHAR2(80),
   CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate),
   CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno)
       REFERENCES emp(empno) ON DELETE CASCADE,
   CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno)
       REFERENCES dept (deptno) ON DELETE SET NULL,
   CONSTRAINT jobhist_date_chk CHECK (startdate <= enddate)
);
--
--  Create the 'salesemp' view
--
CREATE OR REPLACE VIEW salesemp AS
   SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';
--
--  Sequence to generate values for function 'new_empno'.
--
CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;
--
--  Issue PUBLIC grants
--
GRANT ALL ON emp TO PUBLIC;
GRANT ALL ON dept TO PUBLIC;
GRANT ALL ON jobhist TO PUBLIC;
GRANT ALL ON salesemp TO PUBLIC;
GRANT ALL ON next_empno TO PUBLIC;
--
--  Load the 'dept' table
--
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
INSERT INTO dept VALUES (30,'SALES','CHICAGO');
INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');
--
--  Load the 'emp' table
--
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'02-APR-81',2975,NULL,20);
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'28-SEP-81',1250,1400,30);
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,'01-MAY-81',2850,NULL,30);
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,'09-JUN-81',2450,NULL,10);
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,'19-APR-87',3000,NULL,20);
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,'17-NOV-81',5000,NULL,10);
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,'08-SEP-81',1500,0,30);
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1100,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'03-DEC-81',950,NULL,30);
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,'03-DEC-81',3000,NULL,20);
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NULL,10);
--
--  Load the 'jobhist' table
--
INSERT INTO jobhist VALUES (7369,'17-DEC-80',NULL,'CLERK',800,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7499,'20-FEB-81',NULL,'SALESMAN',1600,300,30,'New Hire');
INSERT INTO jobhist VALUES (7521,'22-FEB-81',NULL,'SALESMAN',1250,500,30,'New Hire');
INSERT INTO jobhist VALUES (7566,'02-APR-81',NULL,'MANAGER',2975,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7654,'28-SEP-81',NULL,'SALESMAN',1250,1400,30,'New Hire');
INSERT INTO jobhist VALUES (7698,'01-MAY-81',NULL,'MANAGER',2850,NULL,30,'New Hire
');
INSERT INTO jobhist VALUES (7782,'09-JUN-81',NULL,'MANAGER',2450,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7788,'19-APR-87','12-APR-88','CLERK',1000,NULL,20,'New
Hire');
INSERT INTO jobhist VALUES (7788,'13-APR-88','04-MAY-89','CLERK',1040,NULL,20,'Raise');
INSERT INTO jobhist VALUES (7788,'05-MAY-90',NULL,'ANALYST',3000,NULL,20,'Promoted
to Analyst');
INSERT INTO jobhist VALUES (7839,'17-NOV-81',NULL,'PRESIDENT',5000,NULL,10,'New Hire
');
INSERT INTO jobhist VALUES (7844,'08-SEP-81',NULL,'SALESMAN',1500,0,30,'New Hire');
INSERT INTO jobhist VALUES (7876,'23-MAY-87',NULL,'CLERK',1100,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7900,'03-DEC-81','14-JAN-83','CLERK',950,NULL,10,'New Hire
');
INSERT INTO jobhist VALUES (7900,'15-JAN-83',NULL,'CLERK',950,NULL,30,'Changed to Dept
 30');
INSERT INTO jobhist VALUES (7902,'03-DEC-81',NULL,'ANALYST',3000,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7934,'23-JAN-82',NULL,'CLERK',1300,NULL,10,'New Hire');
--
--  Populate statistics table and view (pg_statistic/pg_stats)
--
ANALYZE dept;
ANALYZE emp;
ANALYZE jobhist;
--
--  Procedure that lists all employees' numbers and names
--  from the 'emp' table using a cursor.
--
CREATE OR REPLACE PROCEDURE list_emp
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur IS
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    --------');
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_cur;
END;
/
--
--  Procedure that selects an employee row given the employee
--  number and displays certain columns.
--
CREATE OR REPLACE PROCEDURE select_emp (
    p_empno         IN  NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_dname         dept.dname%TYPE;
    v_disp_date     VARCHAR2(10);
BEGIN
    SELECT ename, hiredate, sal, NVL(comm, 0), dname
        INTO v_ename, v_hiredate, v_sal, v_comm, v_dname
        FROM emp e, dept d
        WHERE empno = p_empno
          AND e.deptno = d.deptno;
    v_disp_date := TO_CHAR(v_hiredate, 'MM/DD/YYYY');
    DBMS_OUTPUT.PUT_LINE('Number    : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name      : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
    DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
    DBMS_OUTPUT.PUT_LINE('Department: ' || v_dname);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
/
--
--  Procedure that queries the 'emp' table based on
--  department number and employee number or name.  Returns
--  employee number and name as IN OUT parameters and job,
--  hire date, and salary as OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query (
    p_deptno        IN      NUMBER,
    p_empno         IN OUT NUMBER,
    p_ename         IN OUT VARCHAR2,
    p_job           OUT    VARCHAR2,
    p_hiredate      OUT    DATE,
    p_sal           OUT    NUMBER
)
IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
        INTO p_empno, p_ename, p_job, p_hiredate, p_sal
        FROM emp
        WHERE deptno = p_deptno
          AND (empno = p_empno
           OR  ename = UPPER(p_ename));
END;
/
--
--  Procedure to call 'emp_query_caller' with IN and IN OUT
--  parameters.  Displays the results received from IN OUT and
--  OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query_caller
IS
    v_deptno        NUMBER(2);
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER;
BEGIN
    v_deptno := 30;
    v_empno  := 0;
    v_ename  := 'Martin';
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('More than one employee was selected');
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No employees were selected');
END;
/
--
--  Function to compute yearly compensation based on semimonthly
--  salary.
--
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal           NUMBER,
    p_comm          NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;
/
--
--  Function that gets the next number from sequence, 'next_empno',
--  and ensures it is not already in use as an employee number.
--
CREATE OR REPLACE FUNCTION new_empno RETURN NUMBER
IS
    v_cnt           INTEGER := 1;
    v_new_empno     NUMBER;
BEGIN
    WHILE v_cnt > 0 LOOP
        SELECT next_empno.nextval INTO v_new_empno FROM dual;
        SELECT COUNT(*) INTO v_cnt FROM emp WHERE empno = v_new_empno;
    END LOOP;
    RETURN v_new_empno;
END;
/
--
--  EDB-SPL function that adds a new clerk to table 'emp'.  This function
--  uses package 'emp_admin'.
--
CREATE OR REPLACE FUNCTION hire_clerk (
    p_ename         VARCHAR2,
    p_deptno        NUMBER
) RETURN NUMBER
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_mgr           NUMBER(4);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_deptno        NUMBER(2);
BEGIN
    v_empno := new_empno;
    INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
        TRUNC(SYSDATE), 950.00, NULL, p_deptno);
    SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
        FROM emp WHERE empno = v_empno;
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Manager    : ' || v_mgr);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission : ' || v_comm);
    RETURN v_empno;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
        RETURN -1;
END;
/
--
--  PostgreSQL PL/pgSQL function that adds a new salesman
--  to table 'emp'.
--
CREATE OR REPLACE FUNCTION hire_salesman (
    p_ename         VARCHAR,
    p_sal           NUMERIC,
    p_comm          NUMERIC
) RETURNS NUMERIC
AS $$
DECLARE
    v_empno         NUMERIC(4);
    v_ename         VARCHAR(10);
    v_job           VARCHAR(9);
    v_mgr           NUMERIC(4);
    v_hiredate      DATE;
    v_sal           NUMERIC(7,2);
    v_comm          NUMERIC(7,2);
    v_deptno        NUMERIC(2);
BEGIN
    v_empno := new_empno();
    INSERT INTO emp VALUES (v_empno, p_ename, 'SALESMAN', 7698,
        CURRENT_DATE, p_sal, p_comm, 30);
    SELECT INTO
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
        empno, ename, job, mgr, hiredate, sal, comm, deptno
        FROM emp WHERE empno = v_empno;
    RAISE INFO 'Department : %', v_deptno;
    RAISE INFO 'Employee No: %', v_empno;
    RAISE INFO 'Name       : %', v_ename;
    RAISE INFO 'Job        : %', v_job;
    RAISE INFO 'Manager    : %', v_mgr;
    RAISE INFO 'Hire Date  : %', v_hiredate;
    RAISE INFO 'Salary     : %', v_sal;
    RAISE INFO 'Commission : %', v_comm;
    RETURN v_empno;
EXCEPTION
    WHEN OTHERS THEN
```

```
        RAISE INFO 'The following is SQLERRM:';
        RAISE INFO '%', SQLERRM;
        RAISE INFO 'The following is SQLSTATE:';
        RAISE INFO '%', SQLSTATE;
        RETURN -1;
END;
$$ LANGUAGE 'plpgsql';
/
--
--  Rule to INSERT into view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_i AS ON INSERT TO salesemp
DO INSTEAD
    INSERT INTO emp VALUES (NEW.empno, NEW.ename, 'SALESMAN', 7698,
        NEW.hiredate, NEW.sal, NEW.comm, 30);
--
--  Rule to UPDATE view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_u AS ON UPDATE TO salesemp
DO INSTEAD
    UPDATE emp SET empno    = NEW.empno,
                   ename    = NEW.ename,
                   hiredate = NEW.hiredate,
                   sal      = NEW.sal,
                   comm     = NEW.comm
        WHERE empno = OLD.empno;
--
--  Rule to DELETE from view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_d AS ON DELETE TO salesemp
DO INSTEAD
    DELETE FROM emp WHERE empno = OLD.empno;
--
--  After statement-level trigger that displays a message after
--  an insert, update, or deletion to the 'emp' table.  One message
--  per SQL statement is displayed.
--
CREATE OR REPLACE TRIGGER user_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    IF INSERTING THEN
        v_action := ' added employee(s) on ';
    ELSIF UPDATING THEN
        v_action := ' updated employee(s) on ';
    ELSIF DELETING THEN
        v_action := ' deleted employee(s) on ';
    END IF;
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action || TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
--
--  Before row-level trigger that displays employee number and
--  salary of an employee that is about to be added, updated,
--  or deleted in the 'emp' table.
--
CREATE OR REPLACE TRIGGER emp_sal_trig
    BEFORE DELETE OR INSERT OR UPDATE ON emp
    FOR EACH ROW
DECLARE
    sal_diff        NUMBER;
BEGIN
    IF INSERTING THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Inserting employee ' ‖ :NEW.empno);
        DBMS_OUTPUT.PUT_LINE('..New salary: ' ‖ :NEW.sal);
    END IF;
    IF UPDATING THEN
        sal_diff := :NEW.sal - :OLD.sal;
        DBMS_OUTPUT.PUT_LINE('Updating employee ' ‖ :OLD.empno);
        DBMS_OUTPUT.PUT_LINE('..Old salary: ' ‖ :OLD.sal);
        DBMS_OUTPUT.PUT_LINE('..New salary: ' ‖ :NEW.sal);
        DBMS_OUTPUT.PUT_LINE('..Raise     : ' ‖ sal_diff);
    END IF;
    IF DELETING THEN
        DBMS_OUTPUT.PUT_LINE('Deleting employee ' ‖ :OLD.empno);
        DBMS_OUTPUT.PUT_LINE('..Old salary: ' ‖ :OLD.sal);
    END IF;
END;
/
--
--  Package specification for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE emp_admin
IS
    FUNCTION get_dept_name (
        p_deptno        NUMBER
    ) RETURN VARCHAR2;
    FUNCTION update_emp_sal (
        p_empno         NUMBER,
        p_raise         NUMBER
    ) RETURN NUMBER;
    PROCEDURE hire_emp (
        p_empno         NUMBER,
        p_ename         VARCHAR2,
        p_job           VARCHAR2,
        p_sal           NUMBER,
        p_hiredate      DATE,
        p_comm          NUMBER,
        p_mgr           NUMBER,
        p_deptno        NUMBER
    );
    PROCEDURE fire_emp (
        p_empno         NUMBER
    );
END emp_admin;
/
--
--  Package body for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
    --
    --  Function that queries the 'dept' table based on the department
    --  number and returns the corresponding department name.
    --
    FUNCTION get_dept_name (
        p_deptno        IN NUMBER
    ) RETURN VARCHAR2
    IS
        v_dname         VARCHAR2(14);
    BEGIN
        SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
        RETURN v_dname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Invalid department number ' ‖ p_deptno);
            RETURN '';
    END;
    --
    --  Function that updates an employee's salary based on the
    --  employee number and salary increment/decrement passed
    --  as IN parameters.  Upon successful completion the function
    --  returns the new updated salary.
    --
    FUNCTION update_emp_sal (
        p_empno         IN NUMBER,
        p_raise         IN NUMBER
    ) RETURN NUMBER
    IS
        v_sal           NUMBER := 0;
    BEGIN
        SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
        v_sal := v_sal + p_raise;
        UPDATE emp SET sal = v_sal WHERE empno = p_empno;
        RETURN v_sal;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
            RETURN -1;
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
            DBMS_OUTPUT.PUT_LINE(SQLERRM);
            DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
            DBMS_OUTPUT.PUT_LINE(SQLCODE);
            RETURN -1;
    END;
    --
    --  Procedure that inserts a new employee record into the 'emp' table.
    --
    PROCEDURE hire_emp (
        p_empno         NUMBER,
        p_ename         VARCHAR2,
        p_job           VARCHAR2,
        p_sal           NUMBER,
        p_hiredate      OUT    DATE,
        p_comm          NUMBER,
        p_mgr           NUMBER,
        p_deptno        NUMBER
    )
    AS
    BEGIN
        INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
            VALUES(p_empno, p_ename, p_job, p_sal,
                   p_hiredate, p_comm, p_mgr, p_deptno);
    END;
    --
    --  Procedure that deletes an employee record from the 'emp' table based
    --  on the employee number.
    --
    PROCEDURE fire_emp (
        p_empno         NUMBER
    )
    AS
    BEGIN
        DELETE FROM emp WHERE empno = p_empno;
    END;
END;
/
```

```
COMMIT;
```

## 6.1.4 Sample database

## 6.1.5 Create a table

You can create a new table by specifying the table name and the names and types of all columns in the table.

The following table is a simplified version of the emp sample table. Only the basic information is provided to define a table.

```
CREATE TABLE emp (
    empno          NUMBER(4),
    ename          VARCHAR2(10),
    job            VARCHAR2(9),
    mgr            NUMBER(4),
    hiredate       DATE,
    sal            NUMBER(7,2),
    comm           NUMBER(7,2),
    deptno         NUMBER(2)
);
```

You can enter the sample code into PSQL with line breaks. PSQL recognizes that the statement is not terminated until the semicolon.

Whitespace characters such as blanks, tabs, and newlines may be used in SQL statements. Therefore, you can type the statement aligned differently from the preceding example. You can also type the statement on one line. Two dashes (--) introduce comments. Whatever follows the dashes is ignored up to the end of the line. Keywords and identifiers are case insensitive in SQL, except when identifiers are double-quoted to preserve the case. No double-quoted identifiers are used in the preceding example.

VARCHAR2(10) specifies a data type that can store arbitrary character strings with up to 10 characters in length. NUMBER(7,2) is a fixed point number with precision 7 and scale 2. NUMBER(4) is an integer number with precision 4 and scale 0.

PolarDB databases compatible with Oracle support common SQL data types including INTEGER, SMALLINT, NUMBER, REAL, DOUBLE PRECISION, CHAR, VARCHAR2, DATE, and TIMESTAMP, and also support various synonyms for these types.

If you do not need a table or you want to create a new table to replace the table, you can remove the table by running the following statement:

```
DROP TABLE tablename;
```

## 6.1.6 Populate a table with rows

The following INSERT statement is used to populate a table with rows:

```
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
```

**Note:**

All data types use obvious input formats. Constants that are not simple numeric values must be enclosed by single quotation marks ('). The DATE type supports a wide range of content. In this tutorial, the unambiguous format in this example is recommended.

The syntax requires you to remember the order of the columns. An alternative syntax allows you to list the columns:

```
INSERT INTO emp(empno,ename,job,mgr,hiredate,sal,comm,deptno)
    VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);
```

You can list the columns in a different order or omit some columns in some cases, for example, if the commission is unknown. The following example shows this type of query:

```
INSERT INTO emp(empno,ename,job,mgr,hiredate,sal,deptno)
    VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,20);
```

Many developers prefer an explicit list of columns to relying on implicit sorting.

## 6.1.7 Query a table

To retrieve data from a table, you must query the table. A SQL SELECT statement is used in the query. The statement is divided into a select list, a table list, and an optional qualification. The select list displays the columns to be returned. The table list displays the tables from which data is retrieved. The optional qualification specifies relevant restrictions. The following query lists all columns of all employees in the table in no particular order.

```
SELECT * FROM emp;
```

The asterisk (*) in the select list specifies all columns. The following example shows the output from this query.

```
 empno | ename  |    job    | mgr  |      hiredate      |   sal   |  comm   | deptno
-------+--------+-----------+------+--------------------+---------+---------+--------
```

```
  7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800.00 |         |     20
  7499 | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600.00 |  300.00 |     30
  7521 | WARD   | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250.00 |  500.00 |     30
  7566 | JONES  | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975.00 |         |     20
  7654 | MARTIN | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 |     30
  7698 | BLAKE  | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850.00 |         |     30
  7782 | CLARK  | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450.00 |         |     10
  7788 | SCOTT  | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000.00 |         |     20
  7839 | KING   | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000.00 |         |     10
  7844 | TURNER | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500.00 |    0.00 |     30
  7876 | ADAMS  | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100.00 |         |     20
  7900 | JAMES  | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950.00 |         |     30
  7902 | FORD   | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000.00 |         |     20
  7934 | MILLER | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300.00 |         |     10
(14 rows)
```

You may specify any arbitrary expression in the select list. For example, you can run the following query:

```
SELECT ename, sal, sal * 24 AS yearly_salary, deptno FROM emp;

 ename  |   sal   | yearly_salary | deptno
--------+---------+---------------+--------
 SMITH  |  800.00 |      19200.00 |     20
 ALLEN  | 1600.00 |      38400.00 |     30
 WARD   | 1250.00 |      30000.00 |     30
 JONES  | 2975.00 |      71400.00 |     20
 MARTIN | 1250.00 |      30000.00 |     30
 BLAKE  | 2850.00 |      68400.00 |     30
 CLARK  | 2450.00 |      58800.00 |     10
 SCOTT  | 3000.00 |      72000.00 |     20
 KING   | 5000.00 |     120000.00 |     10
 TURNER | 1500.00 |      36000.00 |     30
 ADAMS  | 1100.00 |      26400.00 |     20
 JAMES  |  950.00 |      22800.00 |     30
 FORD   | 3000.00 |      72000.00 |     20
 MILLER | 1300.00 |      31200.00 |     10
(14 rows)
```

The AS clause can be used to relabel the output column. The AS clause is optional.

You can add a WHERE clause to qualify a query. This clause specifies the required rows. The WHERE clause contains a Boolean expression with a truth value. Only the rows for which the Boolean expression is true are returned. The usual Boolean operators including AND, OR, and NOT are allowed in the qualification. For example, the following query retrieves the employees in department 20 with salaries over USD 1000.00:

```
SELECT ename, sal, deptno FROM emp WHERE deptno = 20 AND sal > 1000;

 ename |   sal   | deptno
-------+---------+--------
 JONES | 2975.00 |     20
 SCOTT | 3000.00 |     20
 ADAMS | 1100.00 |     20
 FORD  | 3000.00 |     20
```

```
(4 rows)
```

You can specify that the results of a query are returned in sorted order by using the following query:

```
SELECT ename, sal, deptno FROM emp ORDER BY ename;

 ename  |   sal   | deptno
--------+---------+--------
 ADAMS  | 1100.00 |     20
 ALLEN  | 1600.00 |     30
 BLAKE  | 2850.00 |     30
 CLARK  | 2450.00 |     10
 FORD   | 3000.00 |     20
 JAMES  |  950.00 |     30
 JONES  | 2975.00 |     20
 KING   | 5000.00 |     10
 MARTIN | 1250.00 |     30
 MILLER | 1300.00 |     10
 SCOTT  | 3000.00 |     20
 SMITH  |  800.00 |     20
 TURNER | 1500.00 |     30
 WARD   | 1250.00 |     30
(14 rows)
```

You can specify that duplicate rows are removed from the result by using the following query:

```
SELECT DISTINCT job FROM emp;

   job
-----------
 ANALYST
 CLERK
 MANAGER
 PRESIDENT
 SALESMAN
(5 rows)
```

The next topic describes how to retrieve rows from more than one table in a single query.

## 6.1.8 Joins between tables

You can access one or more tables in each query. You can also process multiple rows from one or more tables concurrently in each query. This query is called a join query. For example, if you want to list the information about all employees and the names and addresses of relevant departments, you must compare the deptno column of each row of the emp table with the deptno column of all rows in the dept table, and select the pairs of rows where these values match. You can use the following query to achieve this purpose:

```
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM emp, dept
WHERE emp.deptno = dept.deptno;

 ename  |   sal   | deptno |   dname    |   loc
```

```
--------+---------+--------+------------+----------
 MILLER | 1300.00 |     10 | ACCOUNTING | NEW YORK
 CLARK  | 2450.00 |     10 | ACCOUNTING | NEW YORK
 KING   | 5000.00 |     10 | ACCOUNTING | NEW YORK
 SCOTT  | 3000.00 |     20 | RESEARCH   | DALLAS
 JONES  | 2975.00 |     20 | RESEARCH   | DALLAS
 SMITH  |  800.00 |     20 | RESEARCH   | DALLAS
 ADAMS  | 1100.00 |     20 | RESEARCH   | DALLAS
 FORD   | 3000.00 |     20 | RESEARCH   | DALLAS
 WARD   | 1250.00 |     30 | SALES      | CHICAGO
 TURNER | 1500.00 |     30 | SALES      | CHICAGO
 ALLEN  | 1600.00 |     30 | SALES      | CHICAGO
 BLAKE  | 2850.00 |     30 | SALES      | CHICAGO
 MARTIN | 1250.00 |     30 | SALES      | CHICAGO
 JAMES  |  950.00 |     30 | SALES      | CHICAGO
(14 rows)
```

You must understand the following comments on this result set:

- No result row corresponds to department 40. No entry in the emp table matches department 40, so the join ignores the unmatched rows in the dept table. The following sections describe how to fix this issue.
- We recommend that you use the following query to list the output columns qualified by table name instead of using asterisks (*) or leaving out the qualification:

  ```
  SELECT ename, sal, dept.deptno, dname, loc FROM emp, dept WHERE emp.deptno =
  dept.deptno;
  ```

The deptno column must be qualified. All other columns have unique names. The parser automatically locates the table that these columns belong to. We recommend that you fully qualify column names in join queries.

You can also write join queries by following this syntax:

```
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM emp INNER JOIN
dept ON emp.deptno = dept.deptno;
```

This syntax helps you understand the following scenario.

In the preceding results for joins, no employees that belong to department 40 are returned and no entries for department 40 are generated. To retrieve the entries for department 40 from the results where no matched employees exist, you can use the query to scan the dept table to find the matched emp row. If no matched row is found, you can use the NULL values to replace the columns in the emp table. This type of query is called an outer join. Most joins are inner joins. The following example shows an outer join:

```
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM dept LEFT OUTER
 JOIN emp ON emp.deptno = dept.deptno;

 ename  |   sal   | deptno |   dname    |   loc
--------+---------+--------+------------+----------
```

```
 MILLER | 1300.00 |     10 | ACCOUNTING | NEW YORK
 CLARK  | 2450.00 |     10 | ACCOUNTING | NEW YORK
 KING   | 5000.00 |     10 | ACCOUNTING | NEW YORK
 SCOTT  | 3000.00 |     20 | RESEARCH   | DALLAS
 JONES  | 2975.00 |     20 | RESEARCH   | DALLAS
 SMITH  |  800.00 |     20 | RESEARCH   | DALLAS
 ADAMS  | 1100.00 |     20 | RESEARCH   | DALLAS
 FORD   | 3000.00 |     20 | RESEARCH   | DALLAS
 WARD   | 1250.00 |     30 | SALES      | CHICAGO
 TURNER | 1500.00 |     30 | SALES      | CHICAGO
 ALLEN  | 1600.00 |     30 | SALES      | CHICAGO
 BLAKE  | 2850.00 |     30 | SALES      | CHICAGO
 MARTIN | 1250.00 |     30 | SALES      | CHICAGO
 JAMES  |  950.00 |     30 | SALES      | CHICAGO
        |         |     40 | OPERATIONS | BOSTON
(15 rows)
```

This query is called a left outer join. The table mentioned on the left of the join operator has each row from the table appearing in the output at least once. The table on the right only has the rows that match some rows of the left table displayed in the output. If a left-table row does not match any rows of the right table, NULL values are used to replace the right-table columns.

As an alternative syntax for an outer join, you can use the outer join operator "(+)" in the join condition within the WHERE clause. The outer join operator is placed after the column name of the table where the NULL values are used to replace unmatched rows. For all the rows in the dept table that have no matched rows in the emp table, the PolarDB database compatible with Oracle returns NULL for any select list expressions that contain columns of emp. Therefore, you can rewrite the query in the following way:

```
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM dept, emp
WHERE emp.deptno(+) = dept.deptno;

 ename  |   sal   | deptno |   dname    |   loc
--------+---------+--------+------------+----------
 MILLER | 1300.00 |     10 | ACCOUNTING | NEW YORK
 CLARK  | 2450.00 |     10 | ACCOUNTING | NEW YORK
 KING   | 5000.00 |     10 | ACCOUNTING | NEW YORK
 SCOTT  | 3000.00 |     20 | RESEARCH   | DALLAS
 JONES  | 2975.00 |     20 | RESEARCH   | DALLAS
 SMITH  |  800.00 |     20 | RESEARCH   | DALLAS
 ADAMS  | 1100.00 |     20 | RESEARCH   | DALLAS
 FORD   | 3000.00 |     20 | RESEARCH   | DALLAS
 WARD   | 1250.00 |     30 | SALES      | CHICAGO
 TURNER | 1500.00 |     30 | SALES      | CHICAGO
 ALLEN  | 1600.00 |     30 | SALES      | CHICAGO
 BLAKE  | 2850.00 |     30 | SALES      | CHICAGO
 MARTIN | 1250.00 |     30 | SALES      | CHICAGO
 JAMES  |  950.00 |     30 | SALES      | CHICAGO
        |         |     40 | OPERATIONS | BOSTON
```

```
(15 rows)
```

We can also join a table with itself. This join is a self join. For example, if you want to find the names of employees along with the names of their managers, you can compare the mgr column of each emp row to the empno column of all other emp rows.

```
SELECT e1.ename || ' works for ' || e2.ename AS "Employees and their Managers" FROM
emp e1, emp e2 WHERE e1.mgr = e2.empno;

 Employees and their Managers
------------------------------
 FORD works for JONES
 SCOTT works for JONES
 WARD works for BLAKE
 TURNER works for BLAKE
 MARTIN works for BLAKE
 JAMES works for BLAKE
 ALLEN works for BLAKE
 MILLER works for CLARK
 ADAMS works for SCOTT
 CLARK works for KING
 BLAKE works for KING
 JONES works for KING
 SMITH works for FORD
(13 rows)
```

In this example, the emp table has been relabeled as e1 to represent the employee row in the select list and in the join condition, and as e2 to represent the matched manager row in the select list and in the join condition. These types of aliases can be used in other queries to reduce input. The following example uses these types of aliases:

```
SELECT e.ename, e.mgr, d.deptno, d.dname, d.loc FROM emp e, dept d WHERE e.deptno
 = d.deptno;

 ename  | mgr  | deptno |   dname    |   loc
--------+------+--------+------------+----------
 MILLER | 7782 |     10 | ACCOUNTING | NEW YORK
 CLARK  | 7839 |     10 | ACCOUNTING | NEW YORK
 KING   |      |     10 | ACCOUNTING | NEW YORK
 SCOTT  | 7566 |     20 | RESEARCH   | DALLAS
 JONES  | 7839 |     20 | RESEARCH   | DALLAS
 SMITH  | 7902 |     20 | RESEARCH   | DALLAS
 ADAMS  | 7788 |     20 | RESEARCH   | DALLAS
 FORD   | 7566 |     20 | RESEARCH   | DALLAS
 WARD   | 7698 |     30 | SALES      | CHICAGO
 TURNER | 7698 |     30 | SALES      | CHICAGO
 ALLEN  | 7698 |     30 | SALES      | CHICAGO
 BLAKE  | 7839 |     30 | SALES      | CHICAGO
 MARTIN | 7698 |     30 | SALES      | CHICAGO
 JAMES  | 7698 |     30 | SALES      | CHICAGO
(14 rows)
```

This is a common abbreviation style.

## 6.1.9 Aggregate functions

Similar to most other relational database services, PolarDB databases compatible with Oracle support aggregate functions. An aggregate function computes a single result from multiple input rows. For example, you can use aggregates to compute the COUNT, SUM, AVG (average), MAX (maximum), and MIN (minimum) over a set of rows.

The following example shows how the highest and lowest salaries are found in a query:

```
SELECT MAX(sal) highest_salary, MIN(sal) lowest_salary FROM emp;

 highest_salary | lowest_salary
----------------+---------------
        5000.00 |        800.00
(1 row)
```

If you want to find the employee with the largest salary, the following query is invalid:

```
SELECT ename FROM emp WHERE sal = MAX(sal);

ERROR:  aggregates not allowed in WHERE clause
```

The MAX aggregate function cannot be used in a WHERE clause. The WHERE clause determines the rows that can be aggregated. The clause must be evaluated before aggregate functions are computed. However, you can use a subquery to restate the query to obtain the expected result:

```
SELECT ename FROM emp WHERE sal = (SELECT MAX(sal) FROM emp);

 ename
-------
 KING
(1 row)
```

The subquery is an independent computation that obtains its own result separately from the outer query.

Aggregates are also very useful in combination with the GROUP BY clause. For example, the following query retrieves the highest salary in each department.

```
SELECT deptno, MAX(sal) FROM emp GROUP BY deptno;

 deptno |   max
--------+---------
     10 | 5000.00
     20 | 3000.00
     30 | 2850.00
```

```
(3 rows)
```

This query produces one output row per department. Each aggregate result is computed over the rows matching that department. You can use the HAVING clause to filter these grouped rows.

```
SELECT deptno, MAX(sal) FROM emp GROUP BY deptno HAVING AVG(sal) > 2000;

 deptno |   max
--------+---------
     10 | 5000.00
     20 | 3000.00
(2 rows)
```

This query retrieves the same results for only those departments that have an average salary greater than 2000.

The following query takes into account only the highest paid employees who are analysts in each department.

```
SELECT deptno, MAX(sal) FROM emp WHERE job = 'ANALYST' GROUP BY deptno HAVING
AVG(sal) > 2000;

 deptno |   max
--------+---------
     20 | 3000.00
(1 row)
```

A subtle distinction exists between the WHERE and HAVING clauses. Before grouping occurs and aggregate functions are applied, the WHERE clause filters out rows. After rows are grouped and aggregate functions are computed for each group, the HAVING clause applies filters on the results.

Therefore, in the previous example, only employees who are analysts are considered. From this subset, the employees are grouped by department and only those groups where the average salary of analysts in the group is greater than 2000 are in the final result. Only the group for department 20 meets the criteria and the maximum analyst salary in department 20 is 3000.00.

## 6.1.10 Updates

You can use the UPDATE statement to change the column values of existing rows.

For example, the following example shows how to offer anyone who is a manager a 10% raise:

```
    SELECT ename, sal FROM emp WHERE job = 'MANAGER';

 ename |   sal
```

```
-------+---------
 JONES | 2975.00
 BLAKE | 2850.00
 CLARK | 2450.00
(3 rows)

UPDATE emp SET sal = sal * 1.1 WHERE job = 'MANAGER';

SELECT ename, sal FROM emp WHERE job = 'MANAGER';

 ename |   sal
-------+---------
 JONES | 3272.50
 BLAKE | 3135.00
 CLARK | 2695.00
(3 rows)
```

## 6.1.11 Deletions

You can use the DELETE statement to remove rows from a table.

For example, the following example shows how all employees in department 20 are deleted.

```
SELECT ename, deptno FROM emp;

 ename  | deptno
--------+--------
 SMITH  |     20
 ALLEN  |     30
 WARD   |     30
 JONES  |     20
 MARTIN |     30
 BLAKE  |     30
 CLARK  |     10
 SCOTT  |     20
 KING   |     10
 TURNER |     30
 ADAMS  |     20
 JAMES  |     30
 FORD   |     20
 MILLER |     10
(14 rows)

DELETE FROM emp WHERE deptno = 20;

SELECT ename, deptno FROM emp;
 ename  | deptno
--------+--------
 ALLEN  |     30
 WARD   |     30
 MARTIN |     30
 BLAKE  |     30
 CLARK  |     10
 KING   |     10
 TURNER |     30
 JAMES  |     30
 MILLER |     10
```

```
(9 rows)
```

Be cautious when you execute a DELETE statement without a WHERE clause. The following example shows this type of statement:

```
DELETE FROM tablename;
```

This statement removes all rows from the specified table and leaves the table empty. The system does not request confirmation before this deletion.

# 6.2 Advanced concepts

## 6.2.1 Views

The following example shows the SELECT statement.

```
SELECT ename, sal, sal * 24 AS yearly_salary, deptno FROM emp;

 ename  |   sal   | yearly_salary | deptno
--------+---------+---------------+--------
 SMITH  |  800.00 |      19200.00 |     20
 ALLEN  | 1600.00 |      38400.00 |     30
 WARD   | 1250.00 |      30000.00 |     30
 JONES  | 2975.00 |      71400.00 |     20
 MARTIN | 1250.00 |      30000.00 |     30
 BLAKE  | 2850.00 |      68400.00 |     30
 CLARK  | 2450.00 |      58800.00 |     10
 SCOTT  | 3000.00 |      72000.00 |     20
 KING   | 5000.00 |     120000.00 |     10
 TURNER | 1500.00 |      36000.00 |     30
 ADAMS | 1100.00 | 26400.00 | 20
 JAMES  |  950.00 |      22800.00 |     30
 FORD   | 3000.00 |      72000.00 |     20
 MILLER | 1300.00 |      31200.00 |     10
(14 rows)
```

If this query is used repeatedly, you can create a view to reuse this query without re-typing the entire SELECT statement each time. The following example shows how to create a view:

```
CREATE VIEW employee_pay AS SELECT ename, sal, sal * 24 AS yearly_salary, deptno
FROM emp;
```

The employee_pay view name can be used as an ordinary table name in a query.

```
SELECT * FROM employee_pay;

 ename  |   sal   | yearly_salary | deptno
--------+---------+---------------+--------
 SMITH  |  800.00 |      19200.00 |     20
 ALLEN  | 1600.00 |      38400.00 |     30
 WARD   | 1250.00 |      30000.00 |     30
 JONES  | 2975.00 |      71400.00 |     20
 MARTIN | 1250.00 |      30000.00 |     30
 BLAKE  | 2850.00 |     68400.00 |     30
 CLARK  | 2450.00 |     58800.00 |     10
 SCOTT  | 3000.00 |     72000.00 |     20
 KING   | 5000.00 |    120000.00 |     10
 TURNER | 1500.00 |     36000.00 |     30
 ADAMS  | 1100.00 |     26400.00 |     20
 JAMES  |  950.00 |     22800.00 |     30
 FORD   | 3000.00 |     72000.00 |     20
 MILLER | 1300.00 |     31200.00 |     10
(14 rows)
```

The liberal use of views is important to create a good SQL database design. Views provide a consistent interface that encapsulates details of the structure of your tables. The tables may change as your application evolves.

Views can be used in almost any place where a real table can be used. Views can be built based on other views.

## 6.2.2 Foreign keys

If you want to make sure that all employees belong to a valid department, you must maintain referential integrity of the data. To maintain referential integrity for simplistic database systems, check whether the dept table contains a matched record and insert or reject a new employee record. This approach causes a number of problems and is not easy to use. PolarDB databases compatible with Oracle can simplify your data management.

A modified version of the emp table presented in section 2.1.2 is shown in this section. A foreign key constraint is added to the version. The following example shows the modified emp table:

```
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                    REFERENCES dept(deptno)
);
```

If an attempt is made to execute the following INSERT statement in the sample emp table , the foreign key constraint emp_ref_dept_fk makes sure that department 50 exists in the dept table. This department does not exist, so the statement is rejected.

```
INSERT INTO emp VALUES (8000,'JONES','CLERK',7902,'17-AUG-07',1200,NULL,50);

ERROR:  insert or update on table "emp" violates foreign key constraint "emp_ref_dept_fk
"
```

```
DETAIL:  Key (deptno)=(50) is not present in table "dept".
```

The behavior of foreign keys can be finely tuned based on your application. The correct use of foreign keys improves the performance of your database applications. We recommend that you improve the use of foreign keys.

## 6.2.3 Pseudo column ROWNUM

ROWNUM is a pseudo column that is assigned an incremental and unique integer value for each row based on the order the rows were retrieved from a query. Therefore, the first row retrieved has ROWNUM of 1. The second row has ROWNUM of 2. The other rows follow similar rules.

This feature can be used to limit the number of rows retrieved by a query. The following example shows how this feature works:

```
SELECT empno, ename, job FROM emp WHERE ROWNUM < 5;

 empno | ename |   job
-------+-------+----------
  7369 | SMITH | CLERK
  7499 | ALLEN | SALESMAN
  7521 | WARD  | SALESMAN
  7566 | JONES | MANAGER
(4 rows)
```

The ROWNUM value is assigned to each row before the result set is sorted. The result set is returned in the order specified by the ORDER BY clause, but the ROWNUM values may not be sorted in ascending order. The following example shows how the result set and ROWNUM values are returned:

```
SELECT ROWNUM, empno, ename, job FROM emp WHERE ROWNUM < 5 ORDER BY ename;

 rownum | empno | ename |   job
--------+-------+-------+----------
      2 |  7499 | ALLEN | SALESMAN
      4 |  7566 | JONES | MANAGER
      1 |  7369 | SMITH | CLERK
      3 |  7521 | WARD  | SALESMAN
(4 rows)
```

The following example shows how a sequence number can be added to each row in the jobhist table. A new column named seqno is added to the table and then the seqno column is set to ROWNUM in the UPDATE statement.

```
ALTER TABLE jobhist ADD seqno NUMBER(3);
```

```
UPDATE jobhist SET seqno = ROWNUM;
```

The following SELECT statement shows the new values of the seqno column.

```
SELECT seqno, empno, TO_CHAR(startdate,'DD-MON-YY') AS start, job FROM jobhist;

 seqno | empno |   start   |    job
-------+-------+-----------+-----------
     1 |  7369 | 17-DEC-80 | CLERK
     2 |  7499 | 20-FEB-81 | SALESMAN
     3 |  7521 | 22-FEB-81 | SALESMAN
     4 |  7566 | 02-APR-81 | MANAGER
     5 |  7654 | 28-SEP-81 | SALESMAN
     6 |  7698 | 01-MAY-81 | MANAGER
     7 |  7782 | 09-JUN-81 | MANAGER
     8 |  7788 | 19-APR-87 | CLERK
     9 |  7788 | 13-APR-88 | CLERK
    10 |  7788 | 05-MAY-90 | ANALYST
    11 |  7839 | 17-NOV-81 | PRESIDENT
    12 |  7844 | 08-SEP-81 | SALESMAN
    13 |  7876 | 23-MAY-87 | CLERK
    14 |  7900 | 03-DEC-81 | CLERK
    15 |  7900 | 15-JAN-83 | CLERK
    16 |  7902 | 03-DEC-81 | ANALYST
    17 |  7934 | 23-JAN-82 | CLERK
(17 rows)
```

## 6.2.4 Synonyms

A synonym is an identifier that can be used to reference another database object in a SQL statement. A synonym is useful in the scenarios where a database object requires full qualification by schema name to be correctly referenced in a SQL statement. A synonym defined for that object simplifies the reference to a single and unqualified name.

PolarDB databases compatible with Oracle support synonyms for:

- Tables
- Views
- Materialized views
- Sequences
- Procedures
- Functions
- Types
- Objects that are accessible through a database link
- Other synonyms

The referenced schema or the referenced object may exist at the time when you create the synonym. A synonym may reference a non-existent object or schema. A synonym is invalid if you drop the referenced object or schema. You must explicitly drop a synonym to remove the synonym.

Similar to any other schema object, PolarDB databases compatible Oracle use the search path to resolve unqualified synonym names. If you have two synonyms with the same name, an unqualified reference to a synonym resolves to the first synonym with the specified name in the search path. If public is in your search path, you can reference a synonym in the schema without qualifying that name.

When a PolarDB database compatible Oracle executes a SQL statement, the permissions of the current user are checked based on the underlying database object of the synonym. If the user does not have the proper permissions for that object, the SQL statement fails.

### Create a synonym

Use the CREATE SYNONYM statement to create a synonym. The statement has the following syntax:

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema.]syn_name
      FOR object_schema.object_name[@dblink_name];
```

**Table 6-1: Parameters**

| Parameter | Description |
|---|---|
| syn_name | The name of the synonym. A synonym name must be unique within a schema. |
| schema | The name of the schema where the synonym is located. If you do not specify a schema name, the synonym is created in the first existing schema in your search path. |
| object_name | The name of the object. |
| object_schema | The name of the schema where the object is located. |
| dblink_name | The name of the database link through which a target object may be accessed. |

You must include the REPLACE clause to replace an existing synonym definition with a new synonym definition.

You must include the PUBLIC clause to create the synonym in the public schema. Compatible with Oracle databases, the CREATE PUBLIC SYNONYM statement creates a synonym that is located in the public schema:

```
CREATE [OR REPLACE] PUBLIC SYNONYM syn_name FOR object_schema.object_name;
```

The following example is a shorthand version:

```
CREATE [OR REPLACE] SYNONYM public.syn_name FOR object_schema.object_name;
```

The following example is used to create a synonym named personnel that references the enterprisedb.emp table.

```
CREATE SYNONYM personnel FOR enterprisedb.emp;
```

Unless the synonym is schema qualified in the CREATE SYNONYM statement, the synonym is created in the first existing schema in your search path. You can view your search path by executing the following statement:

```
SHOW SEARCH_PATH;

      search_path
-----------------------
 development,accounting
(1 row)
```

In this example, if a schema named development does not exist, the synonym is created in the schema named accounting.

The emp table in the enterprisedb schema can be referenced in any DDL or DML SQL statement, by using the personnel synonym:

```
INSERT INTO personnel VALUES (8142,'ANDERSON','CLERK',7902,'17-DEC-06',1300,NULL,20
);

SELECT * FROM personnel;

 empno |  ename   |    job    | mgr  |      hiredate      |   sal   |  comm   | deptno
-------+----------+-----------+------+--------------------+---------+---------+--------
  7369 | SMITH    | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800.00 |         |     20
  7499 | ALLEN    | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600.00 |  300.00 |     30
  7521 | WARD     | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250.00 |  500.00 |     30
  7566 | JONES    | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975.00 |         |     20
  7654 | MARTIN   | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 |     30
  7698 | BLAKE    | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850.00 |         |     30
  7782 | CLARK    | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450.00 |         |     10
  7788 | SCOTT    | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000.00 |         |     20
  7839 | KING     | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000.00 |         |     10
  7844 | TURNER   | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500.00 |    0.00 |     30
  7876 | ADAMS    | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100.00 |         |     20
  7900 | JAMES    | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950.00 |         |     30
  7902 | FORD     | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000.00 |         |     20
  7934 | MILLER   | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300.00 |         |     10
  8142 | ANDERSON | CLERK    | 7902 | 17-DEC-06 00:00:00 | 1300.00 |       |    20
(15 rows)
```

**Delete a synonym**

To delete a synonym, use the DROP SYNONYM statement. The statement has the following syntax:

```
DROP [PUBLIC] SYNONYM [schema.] syn_name
```

**Table 6-2: Parameters**

| Parameter | Description |
|---|---|
| syn_name | The name of the synonym. A synonym name must be unique within a schema. |
| schema | The name of the schema where the synonym is located. |

Similar to any other object that can be schema qualified, you may have two synonyms with the same name in your search path. To clarify the name of the synonym that you want to drop, you must include a schema name. Unless a synonym is schema qualified in the DROP SYNONYM statement, a PolarDB database compatible with Oracle deletes the first instance of the synonym found in your search path.

You can include the PUBLIC clause to drop a synonym that is located in the public schema. Compatible with Oracle databases, the DROP PUBLIC SYNONYM statement drops a synonym that is located in the public schema by using the following syntax:

```
DROP PUBLIC SYNONYM syn_name;
```

The following example shows how the personnel synonym is dropped:

```
DROP SYNONYM personnel;
```

# 6.3 Hierarchical queries

## 6.3.1 Overview

A hierarchical query is a type of query that returns the rows of the result set in a hierarchical order based on data forming a parent-child relationship.

A hierarchy is typically represented by an inverted tree structure. The tree contains interconnected nodes. Each node may be connected to none, one, or multiple child nodes . Each node is connected to one parent node except for the top node which has no parent

. This node is the root node. Each tree has only one root node. Nodes that do not have any child nodes are called leaf nodes. A tree always has at least one leaf node. For example, a tree contains only a single node. In this case, this node is both the root and the leaf.

In a hierarchical query, the rows of the result set represent the nodes of one or more trees.

**Note:**

A specified single row may appear in more than one tree and thus appear more than once in the result set.

The hierarchical relationship in a query is described by the CONNECT BY clause. This clause forms the basis of the order in which rows in the result set are returned. The following example shows how the CONNECT BY clause and its associated optional clauses are used in the SELECT statement.

```
SELECT select_list FROM table_expression [ WHERE ...]
  [ START WITH start_expression ]
    CONNECT BY { PRIOR parent_expr = child_expr |
 child_expr = PRIOR parent_expr }
  [ ORDER SIBLINGS BY column1 [ ASC | DESC ]
      [, column2 [ ASC | DESC ] ] ...
  [ GROUP BY ...]
  [ HAVING ...]
  [ other ...]
```

select_list is one or more expressions that comprise the fields of the result set. table_expression is one or more tables or views from which the rows of the result set originate. other is any additional valid SELECT statement. The following sections describe the clauses pertinent to hierarchical queries, including START WITH, CONNECT BY, and ORDER SIBLINGS BY.

**Note:**

PolarDB databases compatible with Oracle do not support AND or other operators in the CONNECT BY clause.

## 6.3.2 Define parent-child relationships

For any specified row, its parent node and its child nodes are determined by the CONNECT BY clause. The CONNECT BY clause must consist of two expressions compared with the equals (=) operator. One of these two expressions must be preceded by the keyword PRIOR.

To determine the child nodes of any specified row, follow these steps:

1. Evaluate parent_expr on the specified row.

2. Evaluate child_expr on any other row resulting from the evaluation of table_expression.
3. If parent_expr = child_expr, this row is a child node of the specified parent row.
4. Repeat the process for all remaining rows in table_expression. All rows that satisfy the equation in step 3 are the child nodes of the specified parent row.

**Note:**

The evaluation process checks whether a row is a child node occurs on every row returned by table_expression. Then, the WHERE clause is used in table_expression.

By repeating this process, you can regard each child node found in the preceding steps as a parent and build an inverted tree of nodes. The process is completed when the final set of child nodes has no child nodes. These nodes are the leaf nodes.

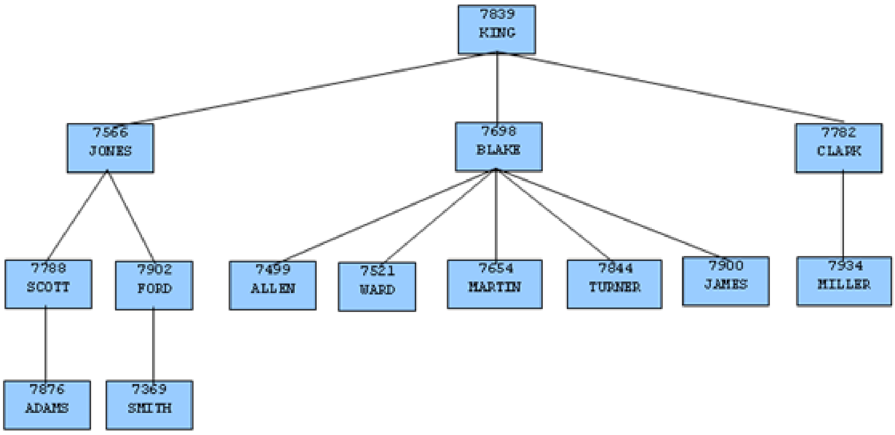
A SELECT statement that includes a CONNECT BY clause includes the START WITH clause. The START WITH clause determines the rows that are the root nodes. For example, the rows are the initial parent nodes on which the preceding algorithm is used. For more information, see the next topic.

## 6.3.3 Select root nodes

The START WITH clause is used to determine the rows selected by table_expression. These rows are used as the root nodes. All rows selected by table_expression where start_expression evaluates to true are regarded as a root node of a tree. The number of potential trees in the result set is equal to the number of root nodes. If the START WITH clause is omitted, every row returned by table_expression is a root of its own tree.

## 6.3.4 Organization tree in the sample application

The following example shows the emp table of the sample application. The rows of the emp table form a hierarchy based on the mgr column. This column contains the employee number of the manager of the employee. Each employee has up to one manager. KING is the president of the company so that he has no manager. The mgr column of KING is null. An employee may act as a manager for more than one employee. This relationship forms a typical, tree-structured, hierarchical organization chart. The following figure shows this relationship.



To form a hierarchical query based on this relationship, the SELECT statement includes this clause: CONNECT BY PRIOR empno = mgr. For example, if the company president KING has the employee number 7839, any employee whose mgr column is 7839 reports to KING. In this case, JONES, BLAKE, and CLARK are the qualified employees, because they are the child nodes of KING. Similarly, for the employee JONES, any other employee with the mgr column that matches 7566 is a child node of JONES. The qualified employees are SCOTT and FORD in this example.

The top of the organization chart is KING so that there is one root node in this tree. The START WITH mgr IS NULL clause only selects KING as the initial root node.

The following example shows the complete SELECT statement:

```
SELECT ename, empno, mgr
FROM emp
START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr;
```

The rows in the query output traverse each branch from the root to leaf moving from top to bottom and from left to right. The following example shows the output of this query:

```
 ename  | empno | mgr
--------+-------+------
 KING   |  7839 |
 JONES  |  7566 | 7839
 SCOTT  |  7788 | 7566
 ADAMS  |  7876 | 7788
 FORD   |  7902 | 7566
 SMITH  |  7369 | 7902
 BLAKE  |  7698 | 7839
 ALLEN  |  7499 | 7698
 WARD   |  7521 | 7698
 MARTIN |  7654 | 7698
 TURNER |  7844 | 7698
 JAMES  |  7900 | 7698
 CLARK  |  7782 | 7839
 MILLER |  7934 | 7782
```

```
(14 rows)
```

## 6.3.5 Node level

LEVEL is a pseudo column that can be used wherever a column can appear in the SELECT statement. For each row in the result set, LEVEL returns a non-zero integer value designating the depth in the hierarchy of the node represented by this row. The LEVEL value for root nodes is 1. The LEVEL value for direct child nodes of root nodes is 2. The LEVEL values for other nodes are calculated in a similar way.

The following query is modified based on the previous query. The LEVEL pseudo column is added to the following query. Based on the LEVEL value, the employee names are indented to emphasize the depth in the hierarchy of each row.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr
FROM emp START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr;
```

The following example shows the output of this query:

```
 level |  employee   | empno | mgr
-------+-------------+-------+------
     1 | KING        |  7839 |
     2 |   JONES     |  7566 | 7839
     3 |     SCOTT   |  7788 | 7566
     4 |       ADAMS |  7876 | 7788
     3 |     FORD    |  7902 | 7566
     4 |       SMITH |  7369 | 7902
     2 |   BLAKE     |  7698 | 7839
     3 |     ALLEN   |  7499 | 7698
     3 |     WARD    |  7521 | 7698
     3 |     MARTIN  |  7654 | 7698
     3 |     TURNER  |  7844 | 7698
     3 |     JAMES   |  7900 | 7698
     2 |   CLARK     |  7782 | 7839
     3 |     MILLER  |  7934 | 7782
(14 rows)
```

Nodes that share a common parent and are at the same level are called siblings. For example, in the preceding output, the employees including ALLEN, WARD, MARTIN, TURNER, and JAMES are siblings, because they are all at level 3 for parent BLAKE. JONES, BLAKE, and CLARK are siblings, because they are at level 2 and KING is their common parent.

## 6.3.6 Order siblings

You can use the ORDER SIBLINGS BY clause to sort the result set by selected column values to order the siblings in ascending or descending order. This special case of the ORDER BY clause can be used only in hierarchical queries.

The previous query is further modified with the addition of ORDER SIBLINGS BY ename ASC.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr
FROM emp START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

The output from the last query is modified so that the siblings appear in ascending order by name. Siblings BLAKE, CLARK, and JONES are alphabetically arranged for KING. Siblings ALLEN, JAMES, MARTIN, TURNER, and WARD are alphabetically arranged for BLAKE. Other column values are ordered in a similar way.

```
 level |  employee   | empno | mgr
-------+-------------+-------+------
     1 | KING        |  7839 |
     2 |   BLAKE     |  7698 | 7839
     3 |     ALLEN   |  7499 | 7698
     3 |     JAMES   |  7900 | 7698
     3 |     MARTIN  |  7654 | 7698
     3 |     TURNER  |  7844 | 7698
     3 |     WARD    |  7521 | 7698
     2 |   CLARK     |  7782 | 7839
     3 |     MILLER  |  7934 | 7782
     2 |   JONES     |  7566 | 7839
     3 |     FORD    |  7902 | 7566
     4 |       SMITH |  7369 | 7902
     3 |     SCOTT   |  7788 | 7566
     4 |       ADAMS |  7876 | 7788
(14 rows)
```

In this final example, the query uses the WHERE clause and starts with three root nodes. After the node tree is constructed, the WHERE clause filters out rows in the tree to form the result set.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr
FROM emp WHERE mgr IN (7839, 7782, 7902, 7788)
START WITH ename IN ('BLAKE','CLARK','JONES')
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

The output from the query shows three level-1 root nodes, including BLAKE, CLARK, and JONES. In addition, rows that do not meet the criteria specified by the WHERE clause have been eliminated from the output.

```
 level | employee  | empno | mgr
-------+-----------+-------+------
```

```
    1 | BLAKE     | 7698 | 7839
    1 | CLARK     | 7782 | 7839
    2 |   MILLER  | 7934 | 7782
    1 | JONES     | 7566 | 7839
    3 |     SMITH | 7369 | 7902
    3 |     ADAMS | 7876 | 7788
(6 rows)
```

## 6.3.7 Use CONNECT_BY_ROOT to retrieve a root node

CONNECT_BY_ROOT is a unary operator that qualifies a column to return a value in this column. The value in the row that is regarded as the root node in relation to the current row.

A unary operator operates on a single operand. In the case of CONNECT_BY_ROOT, the single operand is the column name following the CONNECT_BY_ROOT keyword.

The following example shows the CONNECT_BY_ROOT operator in the context of the SELECT list:

```
SELECT [... ,] CONNECT_BY_ROOT column [, ...]
  FROM table_expression ...
```

When you use the CONNECT_BY_ROOT operator, follow these rules:

- The CONNECT_BY_ROOT operator can be used in the SELECT list, the WHERE clause, the GROUP BY clause, the HAVING clause, the ORDER BY clause, and the ORDER SIBLINGS BY clause if the SELECT statement is used for a hierarchical query.
- The CONNECT_BY_ROOT operator cannot be used in the CONNECT BY clause or the START WITH clause of a hierarchical query.
- The CONNECT_BY_ROOT operator can be used in an expression involving a column. The expression must be enclosed within parentheses.

The following query shows how to use the CONNECT_BY_ROOT operator to return the result set based on trees starting with employees BLAKE, CLARK, and JONES. The result set includes the employee number and employee name of the root node for each employee listed.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
CONNECT_BY_ROOT empno "mgr empno",
CONNECT_BY_ROOT ename "mgr ename"
FROM emp
START WITH ename IN ('BLAKE','CLARK','JONES')
CONNECT BY PRIOR empno = mgr
```

```
ORDER SIBLINGS BY ename ASC;
```

The output from the query shows that all of the root nodes in the columns including mgr empno and mgr ename are one of the employees, including BLAKE, CLARK, and JONES, listed in the START WITH clause.

```
 level | employee  | empno | mgr  | mgr empno | mgr ename
-------+-----------+-------+------+-----------+-----------
     1 | BLAKE     |  7698 | 7839 |      7698 | BLAKE
     2 |   ALLEN   |  7499 | 7698 |      7698 | BLAKE
     2 |   JAMES   |  7900 | 7698 |      7698 | BLAKE
     2 |   MARTIN  |  7654 | 7698 |      7698 | BLAKE
     2 |   TURNER  |  7844 | 7698 |      7698 | BLAKE
     2 |   WARD    |  7521 | 7698 |      7698 | BLAKE
     1 | CLARK     |  7782 | 7839 |      7782 | CLARK
     2 |   MILLER  |  7934 | 7782 |      7782 | CLARK
     1 | JONES     |  7566 | 7839 |      7566 | JONES
     2 |   FORD    |  7902 | 7566 |      7566 | JONES
     3 |     SMITH |  7369 | 7902 |      7566 | JONES
     2 |   SCOTT   |  7788 | 7566 |      7566 | JONES
     3 |     ADAMS |  7876 | 7788 |      7566 | JONES
(13 rows)
```

The following example shows a similar query. In this query, only one tree starting with the single top-level employee is generated. The mgr column must be null.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
CONNECT_BY_ROOT empno "mgr empno",
CONNECT_BY_ROOT ename "mgr ename"
FROM emp START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

In the following output, all of the root nodes in the mgr empno and mgr ename columns indicate KING as the root for this particular query.

```
 level |  employee   | empno | mgr  | mgr empno | mgr ename
-------+-------------+-------+------+-----------+-----------
     1 | KING        |  7839 |      |      7839 | KING
     2 |   BLAKE     |  7698 | 7839 |      7839 | KING
     3 |     ALLEN   |  7499 | 7698 |      7839 | KING
     3 |     JAMES   |  7900 | 7698 |      7839 | KING
     3 |     MARTIN  |  7654 | 7698 |      7839 | KING
     3 |     TURNER  |  7844 | 7698 |      7839 | KING
     3 |     WARD    |  7521 | 7698 |      7839 | KING
     2 |   CLARK     |  7782 | 7839 |      7839 | KING
     3 |     MILLER  |  7934 | 7782 |      7839 | KING
     2 |   JONES     |  7566 | 7839 |      7839 | KING
     3 |     FORD    |  7902 | 7566 |      7839 | KING
     4 |       SMITH |  7369 | 7902 |      7839 | KING
     3 |     SCOTT   |  7788 | 7566 |      7839 | KING
     4 |       ADAMS |  7876 | 7788 |      7839 | KING
(14 rows)
```

By contrast, the following example omits the START WITH clause and generates 14 trees.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
CONNECT_BY_ROOT empno "mgr empno",
CONNECT_BY_ROOT ename "mgr ename"
FROM emp
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

The following example shows the output of this query. Each node appears at least once as a root node for the mgr empno and mgr ename columns. Even the leaf nodes form the top of their own trees.

```
 level |  employee   | empno | mgr  | mgr empno | mgr ename
-------+-------------+-------+------+-----------+-----------
     1 | ADAMS       |  7876 | 7788 |      7876 | ADAMS
     1 | ALLEN       |  7499 | 7698 |      7499 | ALLEN
     1 | BLAKE       |  7698 | 7839 |      7698 | BLAKE
     2 |   ALLEN     |  7499 | 7698 |      7698 | BLAKE
     2 |   JAMES     |  7900 | 7698 |      7698 | BLAKE
     2 |   MARTIN    |  7654 | 7698 |      7698 | BLAKE
     2 |   TURNER    |  7844 | 7698 |      7698 | BLAKE
     2 |   WARD      |  7521 | 7698 |      7698 | BLAKE
     1 | CLARK       |  7782 | 7839 |      7782 | CLARK
     2 |   MILLER    |  7934 | 7782 |      7782 | CLARK
     1 | FORD        |  7902 | 7566 |      7902 | FORD
     2 |   SMITH     |  7369 | 7902 |      7902 | FORD
     1 | JAMES       |  7900 | 7698 |      7900 | JAMES
     1 | JONES       |  7566 | 7839 |      7566 | JONES
     2 |   FORD      |  7902 | 7566 |      7566 | JONES
     3 |     SMITH   |  7369 | 7902 |      7566 | JONES
     2 |   SCOTT     |  7788 | 7566 |      7566 | JONES
     3 |     ADAMS   |  7876 | 7788 |      7566 | JONES
     1 | KING        |  7839 |      |      7839 | KING
     2 |   BLAKE     |  7698 | 7839 |      7839 | KING
     3 |     ALLEN   |  7499 | 7698 |      7839 | KING
     3 |     JAMES   |  7900 | 7698 |      7839 | KING
     3 |     MARTIN  |  7654 | 7698 |      7839 | KING
     3 |     TURNER  |  7844 | 7698 |      7839 | KING
     3 |     WARD    |  7521 | 7698 |      7839 | KING
     2 |   CLARK     |  7782 | 7839 |      7839 | KING
     3 |     MILLER  |  7934 | 7782 |      7839 | KING
     2 |   JONES     |  7566 | 7839 |      7839 | KING
     3 |     FORD    |  7902 | 7566 |      7839 | KING
     4 |       SMITH |  7369 | 7902 |      7839 | KING
     3 |     SCOTT   |  7788 | 7566 |      7839 | KING
     4 |       ADAMS |  7876 | 7788 |      7839 | KING
     1 | MARTIN      |  7654 | 7698 |      7654 | MARTIN
     1 | MILLER      |  7934 | 7782 |      7934 | MILLER
     1 | SCOTT       |  7788 | 7566 |      7788 | SCOTT
     2 |   ADAMS     |  7876 | 7788 |      7788 | SCOTT
     1 | SMITH       |  7369 | 7902 |      7369 | SMITH
     1 | TURNER      |  7844 | 7698 |      7844 | TURNER
     1 | WARD        |  7521 | 7698 |      7521 | WARD
(39 rows)
```

The following example illustrates the unary operator effect of CONNECT_BY_ROOT. When used in an expression that is not enclosed in parentheses, the CONNECT_BY_ROOT operator affects only the ename term that immediately follows the operator. The subsequent concatenation of ‖ ' manages ' ‖ ename is not part of the CONNECT_BY_ROOT operation.

Therefore, the second occurrence of ename results in the value of the current row. The first occurrence of ename results in the value from the root node.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
CONNECT_BY_ROOT ename || ' manages ' || ename "top mgr/employee"
FROM emp
START WITH ename IN ('BLAKE','CLARK','JONES')
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

The following example shows the output of this query. The values are generated for the top mgr/employee column.

```
 level | employee  | empno | mgr  |   top mgr/employee
-------+-----------+-------+------+---------------------
     1 | BLAKE     |  7698 | 7839 | BLAKE manages BLAKE
     2 |   ALLEN   |  7499 | 7698 | BLAKE manages ALLEN
     2 |   JAMES   |  7900 | 7698 | BLAKE manages JAMES
     2 |   MARTIN  |  7654 | 7698 | BLAKE manages MARTIN
     2 |   TURNER  |  7844 | 7698 | BLAKE manages TURNER
     2 |   WARD    |  7521 | 7698 | BLAKE manages WARD
     1 | CLARK     |  7782 | 7839 | CLARK manages CLARK
     2 |   MILLER  |  7934 | 7782 | CLARK manages MILLER
     1 | JONES     |  7566 | 7839 | JONES manages JONES
     2 |   FORD    |  7902 | 7566 | JONES manages FORD
     3 |     SMITH |  7369 | 7902 | JONES manages SMITH
     2 |   SCOTT   |  7788 | 7566 | JONES manages SCOTT
     3 |     ADAMS |  7876 | 7788 | JONES manages ADAMS
(13 rows)
```

In the following example, the CONNECT_BY_ROOT operator is used in an expression that is enclosed in parentheses.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
CONNECT_BY_ROOT ('Manager ' || ename || ' is emp # ' || empno)
"top mgr/empno"
FROM emp
START WITH ename IN ('BLAKE','CLARK','JONES')
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

The following example shows the output of this query. The values of both ename and empno are affected by the CONNECT_BY_ROOT operator. The top mgr/empno column shows the values returned from the root node.

```
 level | employee  | empno | mgr  |       top mgr/empno
-------+-----------+-------+------+-----------------------------
     1 | BLAKE     |  7698 | 7839 | Manager BLAKE is emp # 7698
     2 |   ALLEN   |  7499 | 7698 | Manager BLAKE is emp # 7698
     2 |   JAMES   |  7900 | 7698 | Manager BLAKE is emp # 7698
     2 |   MARTIN  |  7654 | 7698 | Manager BLAKE is emp # 7698
     2 |   TURNER  |  7844 | 7698 | Manager BLAKE is emp # 7698
     2 |   WARD    |  7521 | 7698 | Manager BLAKE is emp # 7698
     1 | CLARK     |  7782 | 7839 | Manager CLARK is emp # 7782
     2 |   MILLER  |  7934 | 7782 | Manager CLARK is emp # 7782
     1 | JONES     |  7566 | 7839 | Manager JONES is emp # 7566
     2 |   FORD    |  7902 | 7566 | Manager JONES is emp # 7566
     3 |     SMITH |  7369 | 7902 | Manager JONES is emp # 7566
     2 |   SCOTT   |  7788 | 7566 | Manager JONES is emp # 7566
     3 |     ADAMS |  7876 | 7788 | Manager JONES is emp # 7566
(13 rows)
```

## 6.3.8 Use SYS_CONNECT_BY_PATH to retrieve a path

SYS_CONNECT_BY_PATH is a function that works within a hierarchical query to retrieve the column values of a specified column that occur between the current node and the root node. The function has the following signature:

```
SYS_CONNECT_BY_PATH (column, delimiter)
```

The function provides two parameters:

- column specifies the name of a column that is located within a table. This table is specified in the hierarchical query that calls the function.
- delimiter specifies the varchar value that separates each entry in the specified column.

The following example returns a list of names of employees and their managers. If a manager reports to a superior manager, the superior manager name is appended to the result:

```
edb=# SELECT level, ename , SYS_CONNECT_BY_PATH(ename, '/') managers
      FROM emp
      CONNECT BY PRIOR empno = mgr
      START WITH mgr IS NULL
      ORDER BY level, ename, managers;
 level | ename  |        managers
-------+--------+-------------------------
     1 | KING   | /KING
     2 | BLAKE  | /KING/BLAKE
     2 | CLARK  | /KING/CLARK
     2 | JONES  | /KING/JONES
     3 | ALLEN  | /KING/BLAKE/ALLEN
     3 | FORD   | /KING/JONES/FORD
     3 | JAMES  | /KING/BLAKE/JAMES
     3 | MARTIN | /KING/BLAKE/MARTIN
     3 | MILLER | /KING/CLARK/MILLER
     3 | SCOTT  | /KING/JONES/SCOTT
     3 | TURNER | /KING/BLAKE/TURNER
     3 | WARD   | /KING/BLAKE/WARD
     4 | ADAMS  | /KING/JONES/SCOTT/ADAMS
     4 | SMITH  | /KING/JONES/FORD/SMITH
(14 rows)
```

Where:

- The level column displays the number of levels that the query returns.
- The ename column displays the employee names.
- The managers column displays the hierarchical list of managers.

The implementation of SYS_CONNECT_BY_PATH used in PolarDB databases for Oracle does not support use of:

- SYS_CONNECT_BY_PATH inside CONNECT_BY_PATH
- SYS_CONNECT_BY_PATH inside SYS_CONNECT_BY_PATH

# 6.4 Multidimensional analysis

## 6.4.1 Overview

Multidimensional analysis is a common process used in data warehousing applications . This process helps you examine data by using various combinations of dimensions. Dimensions are categories used to classify data such as time, geography, departments, and product lines. The results associated with a particular set of dimensions are called facts. Facts are typically figures associated with dimensions such as product sales, profits, volumes, and counts.

You can use SQL aggregation to obtain these facts based on a set of dimensions in a relational database system. During SQL aggregation, data is grouped by certain criteria or dimensions. The result set consists of aggregates of facts, such as counts, sums, and averages of the data in each group.

The GROUP BY clause of the SQL SELECT statement supports the following extensions that simplify the process of generating aggregate results.

- ROLLUP extension
- CUBE extension
- GROUPING SETS extension

In addition, the GROUPING function and the GROUPING_ID function can be used in the SELECT list or the HAVING clause to interpret the results when these extensions are used.

This topic describes how to use these extensions by taking the dept and emp tables for example. The following changes are used to these tables to provide more informative results.

```
UPDATE dept SET loc = 'BOSTON' WHERE deptno = 20;
INSERT INTO emp (empno,ename,job,deptno) VALUES (9001,'SMITH','CLERK',40);
INSERT INTO emp (empno,ename,job,deptno) VALUES (9002,'JONES','ANALYST',40);
INSERT INTO emp (empno,ename,job,deptno) VALUES (9003,'ROGERS','MANAGER',40);
```

The following rows from a join of the emp and dept tables are used:

```
SELECT loc, dname, job, empno FROM emp e, dept d
WHERE e.deptno = d.deptno
ORDER BY 1, 2, 3, 4;

   loc    |   dname    |    job    | empno
----------+------------+-----------+-------
 BOSTON   | OPERATIONS | ANALYST   |  9002
 BOSTON   | OPERATIONS | CLERK     |  9001
 BOSTON   | OPERATIONS | MANAGER   |  9003
 BOSTON   | RESEARCH   | ANALYST   |  7788
 BOSTON   | RESEARCH   | ANALYST   |  7902
 BOSTON   | RESEARCH   | CLERK     |  7369
 BOSTON   | RESEARCH   | CLERK     |  7876
 BOSTON   | RESEARCH   | MANAGER   |  7566
 CHICAGO  | SALES      | CLERK     |  7900
 CHICAGO  | SALES      | MANAGER   |  7698
 CHICAGO  | SALES      | SALESMAN  |  7499
 CHICAGO  | SALES      | SALESMAN  |  7521
 CHICAGO  | SALES      | SALESMAN  |  7654
 CHICAGO  | SALES      | SALESMAN  |  7844
 NEW YORK | ACCOUNTING | CLERK     |  7934
 NEW YORK | ACCOUNTING | MANAGER   |  7782
 NEW YORK | ACCOUNTING | PRESIDENT |  7839
(17 rows)
```

The loc, dname, and job columns are used for the dimensions of the SQL aggregations used in the examples. The COUNT(*) function is used to retrieve the number of employees as the resulting facts of the aggregations.

The following example shows a basic query where the loc, dname, and job columns are grouped.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc, dname, job
ORDER BY 1, 2, 3;
```

The rows of this result set that uses the basic GROUP BY clause without extensions are called the base aggregate rows.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 NEW YORK | ACCOUNTING | CLERK     |         1
 NEW YORK | ACCOUNTING | MANAGER   |         1
 NEW YORK | ACCOUNTING | PRESIDENT |        1
(12 rows)
```

The ROLLUP and CUBE extensions are added to the base aggregate rows and provide additional levels of subtotals to the result set.

The GROUPING SETS extension can be used to combine different types of groups into a single result set.

The GROUPING and GROUPING_ID functions are used to interpret the result set.

For more information about the additions provided by these extensions, see subsequent topics.

## 6.4.2 ROLLUP extension

A ROLLUP extension generates a hierarchical set of groups with subtotals for each hierarchical group and a grand total. The order of the hierarchy is determined by the order of the expressions specified in the ROLLUP expression list. The top of the hierarchy is the leftmost item in the list. Each successive item proceeding to the right side moves down the hierarchy. The rightmost item is at the lowest level.

A single ROLLUP extension has the following syntax:

```
ROLLUP ( { expr_1 | ( expr_1a [, expr_1b ] ...) }
  [, expr_2 | ( expr_2a [, expr_2b ] ...) ] ...)
```

Each expr is an expression that determines the grouping of the result set. If enclosed within parentheses as ( expr_1a, expr_1b, ...), the combination of values returned by expr_1a and expr_1b defines a single grouping level of the hierarchy.

The base level of aggregates returned in the result set corresponds to each unique combination of values returned by the expression list.

A subtotal of each unique value is returned by the first item in the list. This item can be expr_1 or the combination of ( expr_1a, expr_1b, ...). A subtotal of each unique value is returned by the second item in the list. This item can be expr_2 or the combination of ( expr_2a, expr_2b, ...). Similar rules are used within each grouping of the first item and other items. Finally, a grand total is returned for the entire result set.

For the subtotal rows, null is returned for the items across which the subtotal is taken.

The following example shows the ROLLUP extension specified within the context of the GROUP BY clause:

```
SELECT select_list FROM ...
```

```
GROUP BY [... ,] ROLLUP ( expression_list ) [, ...]
```

The items specified in select_list must appear in the ROLLUP expression_list, be aggregate functions such as COUNT, SUM, AVG, MIN, or MAX, or be constants or functions such as the SYSDATE function whose returned values are independent of the individual rows in the group.

The GROUP BY clause may specify multiple ROLLUP extensions and multiple occurrences of other GROUP BY extensions and individual expressions.

You must use the ORDER BY clause if you want to display the output in a hierarchical or meaningful structure. The order of the result set is not determined if no ORDER BY clause is specified.

The number of grouping levels or totals is n + 1, where n represents the number of items in the ROLLUP expression list. A parenthesized list counts as one item.

The following query generates a rollup based on a hierarchy of columns loc, dname, and job.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, dname, job)
ORDER BY 1, 2, 3;
```

The following example shows the result of this query. The system calculates the number of employees for each unique combination of loc, dname, and job, and also calculates the subtotals for each unique combination of loc and dname, for each unique value of loc, and a grand total displayed on the last line.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | OPERATIONS |           |         3
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 BOSTON   | RESEARCH   |           |         5
 BOSTON   |            |           |         8
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 CHICAGO  | SALES      |           |         6
 CHICAGO  |            |           |         6
 NEW YORK | ACCOUNTING | CLERK     |         1
 NEW YORK | ACCOUNTING | MANAGER   |         1
 NEW YORK | ACCOUNTING | PRESIDENT |         1
 NEW YORK | ACCOUNTING |           |         3
 NEW YORK |            |           |         3
          |            |           |        17
```

```
(20 rows)
```

The following query shows how to combine the items in the ROLLUP list within parentheses:

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, (dname, job))
ORDER BY 1, 2, 3;
```

In the following output, different from the last output, no subtotals are generated for loc and dname combinations.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 BOSTON   |            |           |         8
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 CHICAGO  |            |           |         6
 NEW YORK | ACCOUNTING | CLERK     |         1
 NEW YORK | ACCOUNTING | MANAGER   |         1
 NEW YORK | ACCOUNTING | PRESIDENT |         1
 NEW YORK |            |           |         3
          |            |           |        17
(16 rows)
```

If the first two columns in the ROLLUP list are enclosed in parentheses, the subtotal levels are different.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP ((loc, dname), job)
ORDER BY 1, 2, 3;
```

A subtotal is generated for each unique loc and dname combination, but none for unique values of loc.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | OPERATIONS |           |         3
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 BOSTON   | RESEARCH   |           |         5
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 CHICAGO  | SALES      |           |         6
 NEW YORK | ACCOUNTING | CLERK     |        1
 NEW YORK | ACCOUNTING | MANAGER   |        1
 NEW YORK | ACCOUNTING | PRESIDENT |        1
 NEW YORK | ACCOUNTING |           |        3
          |            |           |       17
(17 rows)
```

## 6.4.3 CUBE extension

A CUBE extension is similar to the ROLLUP extension. However, a ROLLUP extension generates groupings and results in a hierarchy based on a left-to-right listing of items in the ROLLUP expression list. The CUBE extension generates groupings and subtotals based on every permutation of all items in the CUBE expression list. The result set contains more rows than a ROLLUP extension used in the same expression list.

A single CUBE expression has the following syntax:

```
CUBE ( { expr_1 | ( expr_1a [, expr_1b ] ...) }
  [, expr_2 | ( expr_2a [, expr_2b ] ...) ] ...)
```

Each expr is an expression that determines the grouping of the result set. If enclosed within parentheses as ( expr_1a, expr_1b, ...), the combination of values returned by expr_1a and expr_1b defines a single group.

The base level of aggregates returned in the result set corresponds to each unique combination of values returned by the expression list.

A subtotal of each unique value is returned by the first item in the list. This item can be expr_1 or the combination of ( expr_1a, expr_1b, ...). A subtotal of each unique value is returned by the second item in the list. This item can be expr_2 or the combination of ( expr_2a, expr_2b, ...). A subtotal of each unique combination is also returned by the first item and the second item. Similarly, if a third item exists, a subtotal of each unique value is returned by the third item, a subtotal of each unique combination is returned by the third item and first item, a subtotal of each unique combination is returned by the third item and second item, and a subtotal of each unique combination is returned by the third item, second item, and first item. Finally, a grand total is returned for the entire result set.

For the subtotal rows, null is returned for the items across which the subtotal is taken.

The following example shows the CUBE extension specified within the context of the GROUP BY clause:

```
SELECT select_list FROM ...
```

```
GROUP BY [... ,] CUBE ( expression_list ) [, ...]
```

The items specified in select_list must appear in the CUBE expression_list, be aggregate functions such as COUNT, SUM, AVG, MIN, or MAX, or be constants or functions such as the SYSDATE function whose returned values are independent of the individual rows in the group.

The GROUP BY clause may specify multiple CUBE extensions and multiple occurrences of other GROUP BY extensions and individual expressions.

You must use the ORDER BY clause if you want to display the output in a meaningful structure. The order of the result set is not determined if no ORDER BY clause is specified.

The number of grouping levels or totals is 2 raised to the power of n, where n represents the number of items in the CUBE expression list. A parenthesized list counts as one item.

The following query generates a cube based on permutations of the loc, dname, and job columns.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (loc, dname, job)
ORDER BY 1, 2, 3;
```

The following example shows the result of this query. The system calculates the number of employees for each combination of loc, dname, and job. The system also calculates the subtotals for each combination of loc and dname, for each combination of loc and job , for each combination of dname and job, for each unique value of loc, for each unique value of dname, and for each unique value of job. Then, the system generates a grand total displayed on the last line.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | OPERATIONS |           |         3
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 BOSTON   | RESEARCH   |           |         5
 BOSTON   |            | ANALYST   |         3
 BOSTON   |            | CLERK     |         3
 BOSTON   |            | MANAGER   |         2
 BOSTON   |            |           |         8
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 CHICAGO  | SALES      |           |         6
 CHICAGO  |            | CLERK     |         1
 CHICAGO  |            | MANAGER   |         1
 CHICAGO  |            | SALESMAN  |         4
 CHICAGO  |            |           |         6
 NEW YORK | ACCOUNTING | CLERK     |         1
 NEW YORK | ACCOUNTING | MANAGER   |         1
 NEW YORK | ACCOUNTING | PRESIDENT |         1
 NEW YORK | ACCOUNTING |           |         3
 NEW YORK |            | CLERK     |         1
 NEW YORK |            | MANAGER   |         1
 NEW YORK |            | PRESIDENT |         1
 NEW YORK |            |           |         3
          | ACCOUNTING | CLERK     |         1
          | ACCOUNTING | MANAGER   |         1
          | ACCOUNTING | PRESIDENT |         1
          | ACCOUNTING |           |         3
          | OPERATIONS | ANALYST   |         1
          | OPERATIONS | CLERK     |         1
          | OPERATIONS | MANAGER   |         1
          | OPERATIONS |           |         3
          | RESEARCH   | ANALYST   |         2
          | RESEARCH   | CLERK     |         2
          | RESEARCH   | MANAGER   |         1
          | RESEARCH   |           |         5
          | SALES      | CLERK     |         1
          | SALES      | MANAGER   |         1
          | SALES      | SALESMAN  |         4
          | SALES      |           |         6
          |            | ANALYST   |         3
          |            | CLERK     |         5
          |            | MANAGER   |         4
          |            | PRESIDENT |         1
          |            | SALESMAN  |         4
          |            |           |        17
(50 rows)
```

The following query shows how to combine the items in the CUBE list within parentheses:

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (loc, (dname, job))
ORDER BY 1, 2, 3;
```

The following output shows that no subtotals are generated for permutations involving the combinations of loc and dname and the combinations of loc and job, or for dname or job.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 BOSTON   |            |           |         8
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 CHICAGO  |            |           |         6
 NEW YORK | ACCOUNTING | CLERK     |         1
 NEW YORK | ACCOUNTING | MANAGER   |         1
 NEW YORK | ACCOUNTING | PRESIDENT |         1
 NEW YORK |            |           |         3
```

```
          | ACCOUNTING | CLERK     |         1
          | ACCOUNTING | MANAGER   |         1
          | ACCOUNTING | PRESIDENT |         1
          | OPERATIONS | ANALYST   |         1
          | OPERATIONS | CLERK     |         1
          | OPERATIONS | MANAGER   |         1
          | RESEARCH   | ANALYST   |         2
          | RESEARCH   | CLERK     |         2
          | RESEARCH   | MANAGER   |         1
          | SALES      | CLERK     |         1
          | SALES      | MANAGER   |         1
          | SALES      | SALESMAN  |         4
          |            |           |        17
(28 rows)
```

The following query shows another variation whereby the first expression is specified outside of the CUBE extension.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc, CUBE (dname, job)
ORDER BY 1, 2, 3;
```

In the following output, the permutations are performed for dname and job within each grouping of loc.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   | OPERATIONS | ANALYST   |         1
 BOSTON   | OPERATIONS | CLERK     |         1
 BOSTON   | OPERATIONS | MANAGER   |         1
 BOSTON   | OPERATIONS |           |         3
 BOSTON   | RESEARCH   | ANALYST   |         2
 BOSTON   | RESEARCH   | CLERK     |         2
 BOSTON   | RESEARCH   | MANAGER   |         1
 BOSTON   | RESEARCH   |           |         5
 BOSTON   |            | ANALYST   |         3
 BOSTON   |            | CLERK     |         3
 BOSTON   |            | MANAGER   |         2
 BOSTON   |            |           |         8
 CHICAGO  | SALES      | CLERK     |         1
 CHICAGO  | SALES      | MANAGER   |         1
 CHICAGO  | SALES      | SALESMAN  |         4
 CHICAGO  | SALES      |           |         6
 CHICAGO  |            | CLERK     |         1
 CHICAGO  |            | MANAGER   |         1
 CHICAGO  |            | SALESMAN  |         4
 CHICAGO  |            |           |         6
 NEW YORK | ACCOUNTING | CLERK     |         1
 NEW YORK | ACCOUNTING | MANAGER   |         1
 NEW YORK | ACCOUNTING | PRESIDENT |         1
 NEW YORK | ACCOUNTING |           |         3
 NEW YORK |            | CLERK     |         1
 NEW YORK |            | MANAGER   |         1
 NEW YORK |            | PRESIDENT |         1
 NEW YORK |            |           |         3
```

```
(28 rows)
```

## 6.4.4 GROUPING SETS extension

A GROUPING SETS extension within the GROUP BY clause is used to generate one result set that is the concatenation of multiple results sets based on different groupings. The UNION ALL operator is used to combine the result sets of multiple groupings into one result set.

The UNION ALL operator and the GROUPING SETS extension do not remove duplicate rows from the combined result sets.

A single GROUPING SETS extension has the following syntax:

```
GROUPING SETS (
  { expr_1 | ( expr_1a [, expr_1b ] ...) |
    ROLLUP ( expr_list ) | CUBE ( expr_list )
  } [, ...] )
```

A GROUPING SETS extension can contain any combination of one or more comma-separated expressions, lists of expressions enclosed within parentheses, ROLLUP extensions, and CUBE extensions.

The GROUPING SETS extension is specified within the context of the GROUP BY clause. The following example shows this extension:

```
SELECT select_list FROM ...
GROUP BY [... ,] GROUPING SETS ( expression_list ) [, ...]
```

The items specified in select_list must appear in the GROUPING SETS expression_list, be aggregate functions such as COUNT, SUM, AVG, MIN, or MAX, or be constants or functions such as the SYSDATE function whose returned values are independent of the individual rows in the group.

The GROUP BY clause may specify multiple GROUPING SETS extensions and multiple occurrences of other GROUP BY extensions and individual expressions.

You must use the ORDER BY clause if you want to display the output in a meaningful structure. The order of the result set is not determined if no ORDER BY clause is specified.

The following query generates a union of groups specified by columns loc, dname, and job.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY GROUPING SETS (loc, dname, job)
```

```
ORDER BY 1, 2, 3;
```

The following example shows the result of this query:

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   |            |           |         8
 CHICAGO  |            |           |         6
 NEW YORK |            |           |         3
          | ACCOUNTING |           |         3
          | OPERATIONS |           |         3
          | RESEARCH   |           |         5
          | SALES      |           |         6
          |            | ANALYST   |         3
          |            | CLERK     |         5
          |            | MANAGER   |         4
          |            | PRESIDENT |         1
          |            | SALESMAN  |         4
(12 rows)
```

To retrieve the same result, you can also use the UNION ALL operator in the following query :

```
SELECT loc AS "loc", NULL AS "dname", NULL AS "job", COUNT(*) AS "employees" FROM
emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc
  UNION ALL
SELECT NULL, dname, NULL, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY dname
  UNION ALL
SELECT NULL, NULL, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY job
ORDER BY 1, 2, 3;
```

The output from the UNION ALL query is the same as the GROUPING SETS output.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   |            |           |         8
 CHICAGO  |            |           |         6
 NEW YORK |            |           |         3
          | ACCOUNTING |           |         3
          | OPERATIONS |           |         3
          | RESEARCH   |           |         5
          | SALES      |           |         6
          |            | ANALYST   |         3
          |            | CLERK     |         5
          |            | MANAGER   |         4
          |            | PRESIDENT |         1
          |            | SALESMAN  |         4
```

```
(12 rows)
```

The following example shows how various types of GROUP BY extensions can be used together within a GROUPING SETS expression list:

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY GROUPING SETS (loc, ROLLUP (dname, job), CUBE (job, loc))
ORDER BY 1, 2, 3;
```

The following example shows the output from the preceding query:

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   |            | ANALYST   |         3
 BOSTON   |            | CLERK     |         3
 BOSTON   |            | MANAGER   |         2
 BOSTON   |            |           |         8
 BOSTON   |            |           |         8
 CHICAGO  |            | CLERK     |         1
 CHICAGO  |            | MANAGER   |         1
 CHICAGO  |            | SALESMAN  |         4
 CHICAGO  |            |           |         6
 CHICAGO  |            |           |         6
 NEW YORK |            | CLERK     |         1
 NEW YORK |            | MANAGER   |         1
 NEW YORK |            | PRESIDENT |         1
 NEW YORK |            |           |         3
 NEW YORK |            |           |         3
          | ACCOUNTING | CLERK     |         1
          | ACCOUNTING | MANAGER   |         1
          | ACCOUNTING | PRESIDENT |         1
          | ACCOUNTING |           |         3
          | OPERATIONS | ANALYST   |         1
          | OPERATIONS | CLERK     |         1
          | OPERATIONS | MANAGER   |         1
          | OPERATIONS |           |         3
          | RESEARCH   | ANALYST   |         2
          | RESEARCH   | CLERK     |         2
          | RESEARCH   | MANAGER   |         1
          | RESEARCH   |           |         5
          | SALES      | CLERK     |         1
          | SALES      | MANAGER   |         1
          | SALES      | SALESMAN  |         4
          | SALES      |           |         6
          |            | ANALYST   |         3
          |            | CLERK     |         5
          |            | MANAGER   |         4
          |            | PRESIDENT |         1
          |            | SALESMAN  |         4
          |            |           |        17
          |            |           |        17
(38 rows)
```

The output is a concatenation of the result sets of GROUP BY loc, GROUP BY ROLLUP (dname, job), and GROUP BY CUBE (job, loc). The following example shows these queries:

```
SELECT loc, NULL AS "dname", NULL AS "job", COUNT(*) AS "employees"
FROM emp e, dept d
```

```
WHERE e.deptno = d.deptno
GROUP BY loc
ORDER BY 1;
```

The following example shows the result set of the GROUP BY loc clause.

```
   loc    | dname | job | employees
----------+-------+-----+-----------
 BOSTON   |       |     |         8
 CHICAGO  |       |     |         6
 NEW YORK |       |     |         3
(3 rows)
```

The following query uses the GROUP BY ROLLUP (dname, job) clause:

```
SELECT NULL AS "loc", dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (dname, job)
ORDER BY 2, 3;
```

The following query is the result set of the GROUP BY ROLLUP (dname, job) clause.

```
 loc |   dname    |    job    | employees
-----+------------+-----------+-----------
     | ACCOUNTING | CLERK     |         1
     | ACCOUNTING | MANAGER   |         1
     | ACCOUNTING | PRESIDENT |         1
     | ACCOUNTING |           |         3
     | OPERATIONS | ANALYST   |         1
     | OPERATIONS | CLERK     |         1
     | OPERATIONS | MANAGER   |         1
     | OPERATIONS |           |         3
     | RESEARCH   | ANALYST   |         2
     | RESEARCH   | CLERK     |         2
     | RESEARCH   | MANAGER   |         1
     | RESEARCH   |           |         5
     | SALES      | CLERK     |         1
     | SALES      | MANAGER   |         1
     | SALES      | SALESMAN  |         4
     | SALES      |           |         6
     |            |           |        17
(17 rows)
```

The following query uses the GROUP BY CUBE (job, loc) clause:

```
SELECT loc, NULL AS "dname", job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (job, loc)
ORDER BY 1, 3;
```

The following example shows the result set of the GROUP BY CUBE (job, loc) clause:

```
   loc    | dname |    job    | employees
----------+-------+-----------+-----------
 BOSTON   |       | ANALYST   |         3
 BOSTON   |       | CLERK     |         3
 BOSTON   |       | MANAGER   |         2
 BOSTON   |       |           |         8
 CHICAGO  |       | CLERK     |         1
 CHICAGO  |      | MANAGER   |         1
 CHICAGO  |      | SALESMAN  |         4
 CHICAGO  |      |           |         6
 NEW YORK |      | CLERK     |         1
 NEW YORK |      | MANAGER   |         1
 NEW YORK |      | PRESIDENT |         1
 NEW YORK |      |           |         3
          |      | ANALYST   |         3
          |      | CLERK     |         5
          |      | MANAGER   |         4
          |      | PRESIDENT |         1
          |      | SALESMAN  |         4
          |      |           |        17
(18 rows)
```

If you combine the preceding three queries by using the UNION ALL operator, a concatenation of the three results sets is generated.

```
SELECT loc AS "loc", NULL AS "dname", NULL AS "job", COUNT(*) AS "employees" FROM
emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc
  UNION ALL
SELECT NULL, dname, job, count(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (dname, job)
  UNION ALL
SELECT loc, NULL, job, count(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (job, loc)
ORDER BY 1, 2, 3;
```

The following example shows the same output as when the GROUP BY GROUPING SETS (loc, ROLLUP (dname, job), CUBE (job, loc)) clause is used.

```
   loc    |   dname    |    job    | employees
----------+------------+-----------+-----------
 BOSTON   |            | ANALYST   |         3
 BOSTON   |            | CLERK     |         3
 BOSTON   |            | MANAGER   |         2
 BOSTON   |            |           |         8
 BOSTON   |            |           |         8
 CHICAGO  |            | CLERK     |         1
 CHICAGO  |            | MANAGER   |         1
 CHICAGO  |            | SALESMAN  |         4
 CHICAGO  |            |           |         6
 CHICAGO  |            |           |         6
 NEW YORK |            | CLERK     |         1
 NEW YORK |            | MANAGER   |         1
 NEW YORK |            | PRESIDENT |         1
 NEW YORK |            |           |         3
 NEW YORK |            |           |         3
          | ACCOUNTING | CLERK     |         1
          | ACCOUNTING | MANAGER   |         1
          | ACCOUNTING | PRESIDENT |         1
          | ACCOUNTING |           |         3
          | OPERATIONS | ANALYST   |         1
          | OPERATIONS | CLERK     |         1
          | OPERATIONS | MANAGER   |         1
          | OPERATIONS |           |         3
```

```
         | RESEARCH   | ANALYST   |         2
         | RESEARCH   | CLERK     |         2
         | RESEARCH   | MANAGER   |         1
         | RESEARCH   |           |         5
         | SALES      | CLERK     |         1
         | SALES      | MANAGER   |         1
         | SALES      | SALESMAN  |         4
         | SALES      |           |         6
         |            | ANALYST   |         3
         |            | CLERK     |         5
         |            | MANAGER   |         4
         |            | PRESIDENT |         1
         |            | SALESMAN  |         4
         |            |           |        17
         |            |           |        17
(38 rows)
```

## 6.4.5 GROUPING function

When you use the ROLLUP, CUBE, or GROUPING SETS extensions to the GROUP BY clause, the various levels of subtotals generated by the extensions may not be distinguished from the base aggregate rows in the result set. The GROUPING function allows you to distinguish them.

The GROUPING function has the following general syntax:

```
SELECT [ expr ...,] GROUPING( col_expr ) [, expr ] ...
FROM ...
GROUP BY [...,]
  { ROLLUP | CUBE | GROUPING SETS }( [...,] col_expr
  [, ...] ) [, ...]
```

The GROUPING function uses a single parameter that must be an expression of a dimension column specified in the expression list of a ROLLUP, CUBE, or GROUPING SETS extension of the GROUP BY clause.

The value returned by the GROUPING function is either 0 or 1. In the result set of a query , if the column expression specified in the GROUPING function is null because the row represents a subtotal over multiple values of that column, the GROUPING function returns a value of 1. If the row returns results based on a particular value of the column specified in the GROUPING function, the GROUPING function returns a value of 0. In the latter case, the column can be a null or non-null values. In both cases, it is for a particular value of that column, not a subtotal across multiple values.

The following query shows how the values returned by the GROUPING function correspond to the subtotal rows.

```
SELECT loc, dname, job, COUNT(*) AS "employees",
  GROUPING(loc) AS "gf_loc",
  GROUPING(dname) AS "gf_dname",
  GROUPING(job) AS "gf_job"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, dname, job)
ORDER BY 1, 2, 3;
```

In the three right-most columns returned by the GROUPING function, a value of 1 appears on a subtotal row wherever a subtotal is taken across values of the corresponding columns.

```
   loc    |   dname    |    job    | employees | gf_loc | gf_dname | gf_job
----------+------------+-----------+-----------+--------+----------+--------
 BOSTON   | OPERATIONS | ANALYST   |         1 |      0 |        0 |      0
 BOSTON   | OPERATIONS | CLERK     |         1 |      0 |        0 |      0
 BOSTON   | OPERATIONS | MANAGER   |         1 |      0 |        0 |      0
 BOSTON   | OPERATIONS |           |         3 |      0 |        0 |      1
 BOSTON   | RESEARCH   | ANALYST   |         2 |      0 |        0 |      0
 BOSTON   | RESEARCH   | CLERK     |         2 |      0 |        0 |      0
 BOSTON   | RESEARCH   | MANAGER   |         1 |      0 |        0 |      0
 BOSTON   | RESEARCH   |           |         5 |      0 |        0 |      1
 BOSTON   |            |           |         8 |      0 |        1 |      1
 CHICAGO  | SALES      | CLERK     |         1 |      0 |        0 |      0
 CHICAGO  | SALES      | MANAGER   |         1 |      0 |        0 |      0
 CHICAGO  | SALES      | SALESMAN  |         4 |      0 |        0 |      0
 CHICAGO  | SALES      |           |         6 |      0 |        0 |      1
 CHICAGO  |            |           |         6 |      0 |        1 |      1
 NEW YORK | ACCOUNTING | CLERK     |         1 |      0 |        0 |      0
 NEW YORK | ACCOUNTING | MANAGER   |         1 |      0 |        0 |      0
 NEW YORK | ACCOUNTING | PRESIDENT |         1 |      0 |        0 |      0
 NEW YORK | ACCOUNTING |           |         3 |      0 |        0 |      1
 NEW YORK |            |           |         3 |      0 |        1 |      1
          |            |           |        17 |      1 |        1 |      1
(20 rows)
```

These indicators can be used as the criteria to filter particular subtotals. For example, in the previous query, you can display only those subtotals for the combinations of loc and dname by using the GROUPING function in a HAVING clause.

```
SELECT loc, dname, job, COUNT(*) AS "employees",
  GROUPING(loc) AS "gf_loc",
  GROUPING(dname) AS "gf_dname",
  GROUPING(job) AS "gf_job"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, dname, job)
HAVING GROUPING(loc) = 0
  AND  GROUPING(dname) = 0
  AND  GROUPING(job) = 1
ORDER BY 1, 2;
```

The following example shows the result of this query:

```
   loc    |   dname    | job | employees | gf_loc | gf_dname | gf_job
----------+------------+-----+-----------+--------+----------+--------
 BOSTON   | OPERATIONS |     |         3 |      0 |        0 |      1
 BOSTON   | RESEARCH   |     |         5 |      0 |        0 |      1
 CHICAGO  | SALES      |     |         6 |      0 |        0 |      1
 NEW YORK | ACCOUNTING |     |         3 |      0 |        0 |      1
```

```
(4 rows)
```

The GROUPING function can be used to distinguish a subtotal row from a base aggregate row or from certain subtotal rows. In these rows, one of the items in the expression list returns null due to the null column on which the expression is based. The null column corresponds to one or more rows in the table. The item does not represent a subtotal over the column.

For example, add the following row to the emp table. As a result, a row with a null value is created for the job column.

```
INSERT INTO emp (empno,ename,deptno) VALUES (9004,'PETERS',40);
```

In the following query, the number of rows is reduced for clarity.

```
SELECT loc, job, COUNT(*) AS "employees",
  GROUPING(loc) AS "gf_loc",
  GROUPING(job) AS "gf_job"
FROM emp e, dept d
WHERE e.deptno = d.deptno AND loc = 'BOSTON'
GROUP BY CUBE (loc, job)
ORDER BY 1, 2;
```

In the following output, two rows contains BOSTON in the loc column and spaces in the job column. The fourth and fifth entries in the table show these two rows.

```
  loc   |   job   | employees | gf_loc | gf_job
--------+---------+-----------+--------+--------
 BOSTON | ANALYST |         3 |      0 |      0
 BOSTON | CLERK   |         3 |      0 |      0
 BOSTON | MANAGER |         2 |      0 |      0
 BOSTON |         |         1 |      0 |      0
 BOSTON |         |         9 |      0 |      1
        | ANALYST |         3 |      1 |      0
        | CLERK   |         3 |      1 |      0
        | MANAGER |         2 |      1 |      0
        |         |         1 |      1 |      0
        |         |         9 |      1 |      1
(10 rows)
```

The GROUPING function on the job column (gf_job) returns 1 in the fifth row to indicate that this value is a subtotal over all jobs. The row contains a subtotal value of 9 in the employees column.

The GROUPING function on the job column and on the loc column returns 0 in the fourth row to indicate that this value is a base aggregate of all rows where loc is BOSTON and job is null. The fourth row is inserted for this example. The employees column contains 1, which indicates the number of null job rows.

In the ninth row next to the last row, the GROUPING function on the job column returns 0 and the GROUPING function on the loc column returns 1. These values are a subtotal over all locations where the job column is null. The employees column indicates the number of null job rows.

## 6.4.6 GROUPING_ID function

The GROUPING_ID function simplifies the implementation of the GROUPING function to determine the subtotal level of a row in the result set from a ROLLBACK, CUBE, or GROUPING SETS extension.

The GROUPING function takes only one column expression and returns a value to indicate whether a row is a subtotal over all values of the specified column. Multiple GROUPING functions may be required to interpret the level of subtotals for queries with multiple grouping columns.

The GROUPING_ID function supports one or more column expressions that have been used in the ROLLBACK, CUBE, or GROUPING SETS extensions and returns a single integer that indicates the column on which a subtotal has been aggregated.

The GROUPING_ID function has the following general syntax:

```
SELECT [ expr ...,]
  GROUPING_ID( col_expr_1 [, col_expr_2 ] ... )
  [, expr ] ...
FROM ...
GROUP BY [...,]
  { ROLLUP | CUBE | GROUPING SETS }( [...,] col_expr_1
  [, col_expr_2 ] [, ...] ) [, ...]
```

The GROUPING_ID function uses one or more parameters that must be expressions of dimension columns specified in the expression list of a ROLLUP, CUBE, or GROUPING SETS extension of the GROUP BY clause.

The GROUPING_ID function returns an integer value. This value corresponds to the base-10 interpretation of a bit vector that consists of concatenated 1s and 0s. This bit vector is returned by a series of GROUPING functions specified in the same left-to-right order as the ordering of the parameters specified in the GROUPING_ID function.

The following query shows how the values in column gid returned by the GROUPING_ID function correspond to the values in columns loc and dname returned by two GROUPING functions.

```
SELECT loc, dname, COUNT(*) AS "employees",
  GROUPING(loc) AS "gf_loc", GROUPING(dname) AS "gf_dname",
  GROUPING_ID(loc, dname) AS "gid"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (loc, dname)
ORDER BY 6, 1, 2;
```

The following output shows the relationship between a bit vector and an integer specified in gid. The bit vector consists of the gf_loc value and the gf_dname value.

```
   loc    |   dname    | employees | gf_loc | gf_dname | gid
----------+------------+-----------+--------+----------+-----
 BOSTON   | OPERATIONS |         3 |      0 |        0 |   0
 BOSTON   | RESEARCH   |         5 |      0 |        0 |   0
 CHICAGO  | SALES      |         6 |      0 |        0 |   0
 NEW YORK | ACCOUNTING |         3 |      0 |        0 |   0
 BOSTON   |            |         8 |      0 |        1 |   1
 CHICAGO  |            |         6 |      0 |        1 |   1
 NEW YORK |            |         3 |      0 |        1 |   1
          | ACCOUNTING |         3 |      1 |        0 |   2
          | OPERATIONS |         3 |      1 |        0 |   2
          | RESEARCH   |         5 |      1 |        0 |   2
          | SALES      |         6 |      1 |        0 |   2
          |            |        17 |      1 |        1 |   3
(12 rows)
```

The following table provides specific examples of the GROUPING_ID function calculations. These calculations are based on four row values returned by the GROUPING function in the output.

| loc | dname | Bit Vector<br>gf_loc gf_dname | GROUPING_ID<br>gid |
|---|---|---|---|
| BOSTON | OPERATIONS | $0 * 2^{1} + 0 * 2^{0}$ | 0 |
| BOSTON | null | $0 * 2^{1} + 1 * 2^{0}$ | 1 |
| null | ACCOUNTING | $1 * 2^{1} + 0 * 2^{0}$ | 2 |
| null | null | $1 * 2^{1} + 1 * 2^{0}$ | 3 |

The following table summarizes how the values returned by the GROUPING_ID function correspond to the grouping columns to be aggregated.

| Aggregation by column | Bit vector<br>gf_loc gf_dname | GROUPING_ID<br>gid |
|---|---|---|
| loc, dname | 0 0 | 0 |
| loc | 0 1 | 1 |
| dname | 1 0 | 2 |

| Aggregation by column | Bit vector<br>gf_loc gf_dname | GROUPING_ID<br>gid |
|---|---|---|
| Grand Total | 1 1 | 3 |

To display only those subtotals by dname, the following simplified query can be used with a HAVING clause based on the GROUPING_ID function.

```
SELECT loc, dname, COUNT(*) AS "employees",
  GROUPING(loc) AS "gf_loc", GROUPING(dname) AS "gf_dname",
  GROUPING_ID(loc, dname) AS "gid"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (loc, dname)
HAVING GROUPING_ID(loc, dname) = 2
ORDER BY 6, 1, 2;
```

The following example shows the result of this query:

```
loc |   dname    | employees | gf_loc | gf_dname | gid
-----+------------+-----------+--------+----------+-----
     | ACCOUNTING |         3 |      1 |        0 |   2
     | OPERATIONS |         3 |      1 |        0 |   2
     | RESEARCH   |         5 |      1 |        0 |   2
     | SALES      |         6 |      1 |        0 |   2
(4 rows)
```

# 6.5 Profiles

## 6.5.1 Overview

PolarDB databases compatible with Oracle allow a database superuser to create named profiles. Each profile defines rules for password management that enhances the password and md5 authentication. The rules in a profile support these features:

- Count failed logon attempts.
- Lock an account due to excessive failed logon attempts.
- Mark a password for expiration.
- Define a grace period after a password expires.
- Define rules for password complexity.
- Define rules of reusing a password.

A profile is a named set of password attributes that allow you to easily manage a group of roles. These roles share comparable authentication rules. If the password requirements change, you can modify the profile to create new rules for each user that is associated with that profile.

After you create a profile, you can associate the profile with one or more users. When a user connects to the server, the server enforces the profile that is associated with the logon role . Profiles are shared by all databases within a cluster, but each cluster may have multiple profiles. A single user that has access to multiple databases use the same profile to connect to each database within the cluster.

A PolarDB database compatible with Oracle creates a profile named default that is associated with a new role when the role is created. If an alternative profile is specified, the new role is associated with the specified profile. If you upgrade the server to a PolarDB database compatible with Oracle, existing roles are automatically assigned to the default profile. You cannot delete the default profile.

The default profile specifies the following attributes:

```
FAILED_LOGIN_ATTEMPTS UNLIMITED
PASSWORD_LOCK_TIME UNLIMITED
PASSWORD_LIFE_TIME  UNLIMITED
PASSWORD_GRACE_TIME  UNLIMITED
PASSWORD_REUSE_TIME  UNLIMITED
PASSWORD_REUSE_MAX  UNLIMITED
PASSWORD_VERIFY_FUNCTION NULL
PASSWORD_ALLOW_HASHED  TRUE
```

## 6.5.2 Create a new profile

You can use the CREATE PROFILE statement to create a new profile. The statement has the following syntax:

```
CREATE PROFILE profile_name
 [LIMIT {parameter value} ... ];
```

You can use the LIMIT clause and one or more space-delimited parameter-value pairs to specify the rules enforced by PolarDB databases compatible with Oracle.

**Parameters**

| Parameter | Description |
|---|---|
| profile_name | Specifies the name of a profile. |
| parameter | Specifies the attribute limited by the profile. |
| value | Specifies the parameter limit. |

PolarDB databases compatible with Oracle support the following values for each parameter :

FAILED_LOGIN_ATTEMPTS specifies the number of failed logon attempts that a user has made before the server locks the account of the user. PASSWORD_LOCK_TIME specifies the period in which the account is locked. Valid values:

- An INTEGER value greater than 0.
- DEFAULT: the value of FAILED_LOGIN_ATTEMPTS specified in the DEFAULT profile.
- UNLIMITED: specifies that the system allows an unlimited number of failed logon attempts.

PASSWORD_LOCK_TIME specifies the period in which an account is locked before the server unlocks the account. This account is locked due to the failed logon attempts more than the value specified by FAILED_LOGIN_ATTEMPTS. Valid values:

- A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days and 12 hours.
- DEFAULT: the value of PASSWORD_LOCK_TIME specified in the DEFAULT profile.
- UNLIMITED: the account is locked until it is manually unlocked by a database superuser.

PASSWORD_LIFE_TIME specifies the number of days that the current password are used before the user is prompted to provide a new password. If you use the PASSWORD_LIFE_TIME clause, you can use the PASSWORD_GRACE_TIME clause to specify the period between the time when a password expires and the time when the connection request of the role that uses the password is rejected. If PASSWORD_GRACE_TIME is not specified, the password expires on the day specified by the default value of PASSWORD_GRACE_TIME. Then, the user is not allowed to execute any statement before a new password is provided. Valid values:

- A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days and 12 hours.
- DEFAULT: the value of PASSWORD_LIFE_TIME specified in the DEFAULT profile.
- UNLIMITED: specifies that the password never expires.

PASSWORD_GRACE_TIME specifies the grace period between the time when a password expires and the time when the user is forced to change the password. After the grace period, a user is allowed to connect to the service, but cannot execute any statement before the user updates the expired password. Valid values:

- A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days and 12 hours.
- DEFAULT: the value of PASSWORD_GRACE_TIME specified in the DEFAULT profile.
- UNLIMITED: specifies that the grace period is infinite.

PASSWORD_REUSE_TIME specifies the number of days a user must wait before the user can reuse a password.

The PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX parameters are used together. If you specify a finite value for either of the parameters and the other parameter is set to UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED, passwords can be reused without restrictions. Valid values:

- A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days and 12 hours.
- DEFAULT: the value of PASSWORD_REUSE_TIME specified in the DEFAULT profile.
- UNLIMITED: specifies that the password can be reused without restrictions.

PASSWORD_REUSE_MAX specifies the number of password changes that must occur before a password can be reused.

The PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX parameters are used together. If you specify a finite value for either of the parameters and the other parameter is set to UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED, passwords can be reused without restrictions. Valid values:

- An INTEGER value greater than or equal to 0.
- DEFAULT: the value of PASSWORD_REUSE_MAX specified in the DEFAULT profile.
- UNLIMITED: specifies that the password can be reused without restrictions.

PASSWORD_VERIFY_FUNCTION specifies password complexity. Valid values:

- The name of a PL/SQL function.
- DEFAULT: the value of PASSWORD_VERIFY_FUNCTION specified in the DEFAULT profile.
- NULL

PASSWORD_ALLOW_HASHED specifies whether an encrypted password can be used. If you specify TRUE, the system allows a user to change the password by specifying a hash computed encrypted password on the client side. However, if you specify FALSE, a valid password must be in a plain-text form. Otherwise, an error message is returned if a server receives an encrypted password. Valid values:

- A BOOLEAN value: TRUE, ON, YES, 1, FALSE, OFF, NO, and 0.
- DEFAULT: the value of PASSWORD_ALLOW_HASHED specified in the DEFAULT profile.

**Note:**

The PASSWORD_ALLOW_HASHED parameter is not compatible with Oracle.

**Notes**

You can run the DROP PROFILE statement to remove the profile.

**Examples**

You can run the following statement to create a profile named acctg. The profile specifies that an account is locked for one day if the user has not been authenticated with the correct password during five attempts.

```
CREATE PROFILE acctg LIMIT
       FAILED_LOGIN_ATTEMPTS 5
       PASSWORD_LOCK_TIME 1;
```

You can run the following statement to create a profile named sales. The profile specifies that a user must change their password every 90 days.

```
CREATE PROFILE sales LIMIT
       PASSWORD_LIFE_TIME 90
       PASSWORD_GRACE_TIME 3;
```

If the user has not changed their password during the 90 days specified in the profile, an error message is returned when the user tries to log on to the service. After a grace period of three days, the account is not be allowed to execute any statements before the user change the password.

You can run the following statement to create a profile named accts. The profile specifies that a user cannot reuse a password within 180 days after the password is used, and must change the password at least five times before the password is reused.

```
CREATE PROFILE accts LIMIT
       PASSWORD_REUSE_TIME 180
       PASSWORD_REUSE_MAX 5;
```

You can run the following statement to create a profile named resources. The profile calls a user-defined function named password_rules. This function verifies that the provided password meets the complexity requirements:

```
CREATE PROFILE resources LIMIT
```

```
    PASSWORD_VERIFY_FUNCTION password_rules;
```

## 6.5.3 Alter a profile

Use the ALTER PROFILE statement to modify a user-defined profile. PolarDB databases compatible with Oracle support the following statements:

```
ALTER PROFILE profile_name RENAME TO new_name;

ALTER PROFILE profile_name
    LIMIT {parameter value}[...] ;
```

You can use the LIMIT clause and one or more space-delimited parameter-value pairs to specify the rules enforced by PolarDB databases compatible with Oracle. You can also use ALTER PROFILE...RENAME TO to change the name of a profile.

**Parameters**

| Parameter | Description |
|---|---|
| profile_name | Specifies the name of a profile. |
| new_name | Specifies the new name of the profile. |
| parameter | Specifies the attribute limited by the profile. |
| value | Specifies the parameter limit. |

**Examples**

The following example shows how to modify a profile named acctg_profile:

```
ALTER PROFILE acctg_profile
    LIMIT FAILED_LOGIN_ATTEMPTS 3 PASSWORD_LOCK_TIME 1;
```

The profile is used to calculate the number of failed attempts that a logon role has made to connect to the server. The profile specifies that the account is locked for one day if the role has not been authenticated with the correct password during three attempts.

The following example changes the name of acctg_profile to payables_profile:

```
ALTER PROFILE acctg_profile RENAME TO payables_profile;
```

## 6.5.4 Drop a profile

You can use the DROP PROFILE statement to drop a profile. The statement has the following syntax:

```
DROP PROFILE [IF EXISTS] profile_name [CASCADE|RESTRICT];
```

The IF EXISTS clause specifies that the server does not return an error if the specified profile does not exist. The server generates a notification if the profile does not exist.

You can use the optional CASCADE clause to reassign any users that are associated with the profile to the default profile and then drop the profile. The optional RESTRICT clause specifies that the server does not drop any profile that is associated with a role. This is the default behavior.

**Parameters**

| Parameter | Description |
|---|---|
| profile_name | The name of the profile to be dropped. |

**Examples**

The following example drops a profile named acctg_profile:

```
DROP PROFILE acctg_profile CASCADE;
```

The statement associates any roles associated with the acctg_profile profile with the default profile again and then drops the acctg_profile profile.

The following example drops a profile named acctg_profile:

```
DROP PROFILE acctg_profile RESTRICT;
```

The RESTRICT clause in the statement specify that the server does not drop acctg_profile if any roles are associated with the profile.

## 6.5.5 Back up profile management functions

A profile may include the PASSWORD_VERIFY_FUNCTION clause that references a user-defined function. This function specifies the behavior enforced by PolarDB databases compatible with Oracle. Profiles are global objects. These objects are shared by all databases within a cluster. Different from profiles, user-defined functions are database objects.

By invoking pg_dumpall with the -g or -r option, you can create a script that recreates the definition of any existing profiles. However, the script does not recreate the user-defined functions that are referenced by the PASSWORD_VERIFY_FUNCTION clause. You must use the pg_dump utility to explicitly dump the database where those functions are located and then restore the database.

The script created by pg_dump contains the following statement that includes the clause and function name:

```
ALTER PROFILE... LIMIT PASSWORD_VERIFY_FUNCTION function_name
```

This statement helps to associate the restored function with the profile with which the function was previously associated.

If the PASSWORD_VERIFY_FUNCTION clause is set to DEFAULT or NULL, the behavior is replicated by the script generated by the pg_dumpall -g or pg_dumpall -r statement.

# 6.6 Optimizer hints

## 6.6.1 Overview

When you invoke the DELETE, INSERT, SELECT or UPDATE statement, the server generates a set of execution plans. After analyzing those execution plans, the server selects a plan that returns a result set within the least amount of time. The server selects a plan based on several factors:

- The estimated execution cost of data handling operations.
- Parameter values assigned to parameters in the Query Tuning section of the postgresql.conf file.
- Column statistics that have been gathered by the ANALYZE statement.

The query planner selects the most cost-effective plan. You can use an optimizer hint to set the mode in which the server selects a query plan. An optimizer hint includes one or more directives embedded in a syntax similar to a comment. The syntax immediately follows the DELETE, INSERT, SELECT or UPDATE statement. When the server generates a result set, the server employs or avoids a specific plan based on keywords in the comment.

```
{ DELETE | INSERT | SELECT | UPDATE } /*+ { hint [ comment ] } [...] */
```

```
 statement_body
{ DELETE | INSERT | SELECT | UPDATE } --+ { hint [ comment ] } [...]
 statement_body
```

Optimizer hints may be included in either of the preceding forms. In both forms, a plus sign (+) must immediately follow the /* or -- opening comment symbols, with no spaces between the signs. Otherwise, the server cannot interpret the following tokens as hints.

If you use the first form, the hint and optional comment may span multiple lines. The second form requires all hints and comments to occupy a single line. The remaining parts of the statement must start on a new line.

**Note:**

- The database server always tries to use the specified hints.
- If a planner method parameter is set to disable a certain plan type, this plan is not be used even if the plan is specified in a hint, unless no other options are available to the planner. Examples of planner method parameters are enable_indexscan, enable_seqscan, enable_hashjoin, enable_mergejoin, and enable_nestloop. All these parameters are Boolean parameters.
- The hint is embedded within a comment. If the hint is misspelled, or if any parameter of the hint such as the view, table, or column name is misspelled or does not exist in the SQL statement, the system does not indicate that any type of error has occurred. No syntax error is specified and the entire hint is ignored.
- If an alias is used for a table or view name in the SQL statement, the alias name rather than the original object name must be used in the hint. For example, in the statement, SELECT /*+ FULL(acct) */ * FROM accounts acct ..., the alias of acct for accounts rather than the table name accounts must be specified in the FULL hint.

Use the EXPLAIN statement to make sure that the hint is correctly formed and the planner uses the hint. For more information about the EXPLAIN statement, see the documentation of PolarDB databases compatible with Oracle.

Optimizer hints cannot be used in production applications where table data changes throughout the life of the application. To make sure that dynamic columns are frequently analyzed with the ANALYZE statement, the column statistics is updated to reflect value changes, and the planner uses the statistics to generate the most cost-effective plan for any specified statement execution. However, optimizer hints generate in the same plan, regardless of how the table data changes.

**Parameters**

| Parameter | Description |
|---|---|
| hint | An optimizer hint directive. |
| comment | A string with additional information. The characters that can be included in a comment are restricted. A comment can only contain letters, digits, underscores (_), dollar signs ($), number signs (#), and space characters. These characters must conform to the syntax of an identifier. Any subsequent hint is ignored if the comment is not in this form. |
| statement_body | The remaining part of the DELETE, INSERT, SELECT, or UPDATE statement. |

For more information about the optimizer hint directives, see the following topics.

## 6.6.2 Default optimization mode

Multiple optimization modes are available. You can select one optimization mode as as the default mode for a PolarDB database cluster compatible with Oracle. You can also change this setting on a per-session basis by running the ALTER SESSION statement and or by running the DELETE, SELECT, or UPDATE statement within an optimizer hint. The configuration parameter that specifies the default mode is named OPTIMIZER_MODE. The following table shows the valid values of this parameter.

| Hint | Description |
|---|---|
| ALL_ROWS | Optimizes retrieval of all rows of the result set. |
| CHOOSE | Does not implement the default optimization based on the assumed number of rows to be retrieved from the result set. This is the default value. |
| FIRST_ROWS | Optimizes retrieval of only the first row of the result set. |
| FIRST_ROWS_10 | Optimizes retrieval of the first 10 rows of the results set. |
| FIRST_ROWS_100 | Optimizes retrieval of the first 100 rows of the result set. |
| FIRST_ROWS_1000 | Optimizes retrieval of the first 1,000 rows of the result set. |
| FIRST_ROWS(n) | Optimizes retrieval of the first n rows of the result set. This form cannot be used as the object of the ALTER SESSION SET OPTIMIZER_MODE statement. This form can only be used as a hint in a SQL statement. |

If you submit the SQL statement to use these optimization modes, you can only view the first n rows of the result set and abandon the other rows of the result set. The system allocates resources to the query based on this rule.

**Examples**

Modify the current session to optimize retrieval of the first 10 rows of the result set.

```
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS_10;
```

You can run the SHOW statement to show the current value of the OPTIMIZER_MODE parameter. This statement is a utility dependent statement. In PSQL, the SHOW statement is used as follows:

```
SHOW OPTIMIZER_MODE;

optimizer_mode
----------------
 first_rows_10
(1 row)
```

The SHOW statement is compatible with Oracle databases and supports the following syntax:

```
SHOW PARAMETER OPTIMIZER_MODE;

NAME
--------------------------------------------------
VALUE
--------------------------------------------------
optimizer_mode
first_rows_10
```

The following example shows an optimization mode used as a hint in a SELECT statement:

```
SELECT /*+ FIRST_ROWS(7) */ * FROM emp;

 empno | ename  |    job    | mgr  |      hiredate      |   sal   |  comm   | deptno
-------+--------+-----------+------+--------------------+---------+---------+--------
  7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800.00 |         |     20
  7499 | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600.00 |  300.00 |     30
  7521 | WARD   | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250.00 |  500.00 |     30
  7566 | JONES  | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975.00 |         |     20
  7654 | MARTIN | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 |     30
  7698 | BLAKE  | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850.00 |         |     30
  7782 | CLARK  | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450.00 |         |     10
  7788 | SCOTT  | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000.00 |         |     20
  7839 | KING   | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000.00 |         |     10
  7844 | TURNER | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500.00 |    0.00 |     30
  7876 | ADAMS  | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100.00 |         |     20
  7900 | JAMES  | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950.00 |         |     30
  7902 | FORD   | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000.00 |         |     20
  7934 | MILLER | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300.00 |         |     10
```

```
(14 rows)
```

## 6.6.3 Access method hints

The following hints determine how the optimizer accesses relations to create a result set.

| Hint | Description |
|---|---|
| FULL(table) | Performs a full sequential scan on the table. |
| INDEX(table [ index ] [...]) | Uses the index on the table to access a relation. |
| NO_INDEX(table [ index ] [...]) | Does not use the index on table to access a relation. |

In addition, the ALL_ROWS, FIRST_ROWS, and FIRST_ROWS(n) hints in this table can be used.

### Examples

The sample application does not have enough data to describe the effect of optimizer hints . Therefore, the remaining examples in this section use the bank database created by the pgbench application. This application is located in the bin subdirectory of the PolarDB database compatible with Oracle.

The following example shows how to create a database named bank. The database is populated by the tables including pgbench_accounts, pgbench_branches, pgbench_tellers , and pgbench_history. The -s 20 option specifies a scaling factor of 20. This factor allows you to create 20 branches. Each branch has 100,000 accounts. Therefore, a total of 2,000, 000 rows are generated in the pgbench_accounts table and 20 rows are generated in the pgbench_branches table. Ten tellers are assigned to each branch. As a result, a total of 200 rows are generated in the pgbench_tellers table.

The following example shows how to initialize the pgbench application in the bank database.

```
createdb -U enterprisedb bank
CREATE DATABASE

pgbench -i -s 20 -U enterprisedb bank

NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
creating tables...
100000 of 2000000 tuples (5%) done (elapsed 0.11 s, remaining 2.10 s)
200000 of 2000000 tuples (10%) done (elapsed 0.22 s, remaining 1.98 s)
300000 of 2000000 tuples (15%) done (elapsed 0.33 s, remaining 1.84 s)
400000 of 2000000 tuples (20%) done (elapsed 0.42 s, remaining 1.67 s)
```

```
500000 of 2000000 tuples (25%) done (elapsed 0.52 s, remaining 1.57 s)
600000 of 2000000 tuples (30%) done (elapsed 0.62 s, remaining 1.45 s)
700000 of 2000000 tuples (35%) done (elapsed 0.73 s, remaining 1.35 s)
800000 of 2000000 tuples (40%) done (elapsed 0.87 s, remaining 1.31 s)
900000 of 2000000 tuples (45%) done (elapsed 0.98 s, remaining 1.19 s)
1000000 of 2000000 tuples (50%) done (elapsed 1.09 s, remaining 1.09 s)
1100000 of 2000000 tuples (55%) done (elapsed 1.22 s, remaining 1.00 s)
1200000 of 2000000 tuples (60%) done (elapsed 1.36 s, remaining 0.91 s)
1300000 of 2000000 tuples (65%) done (elapsed 1.51 s, remaining 0.82 s)
1400000 of 2000000 tuples (70%) done (elapsed 1.65 s, remaining 0.71 s)
1500000 of 2000000 tuples (75%) done (elapsed 1.78 s, remaining 0.59 s)
1600000 of 2000000 tuples (80%) done (elapsed 1.93 s, remaining 0.48 s)
1700000 of 2000000 tuples (85%) done (elapsed 2.10 s, remaining 0.37 s)
1800000 of 2000000 tuples (90%) done (elapsed 2.23 s, remaining 0.25 s)
1900000 of 2000000 tuples (95%) done (elapsed 2.37 s, remaining 0.12 s)
2000000 of 2000000 tuples (100%) done (elapsed 2.48 s, remaining 0.00 s)
vacuum...
set primary keys...
done.
```

A total of 500,000 transactions are processed. Therefore, the pgbench_history table is populated with 500,000 rows.

```
pgbench -U enterprisedb -t 500000 bank

starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 20
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 500000
number of transactions actually processed: 500000/500000
latency average: 0.000 ms
tps = 1464.338375 (including connections establishing)
tps = 1464.350357 (excluding connections establishing)
```

The following example shows the table definitions:

```
\d pgbench_accounts

   Table "public.pgbench_accounts"
  Column  |     Type      | Modifiers
----------+---------------+-----------
 aid      | integer       | not null
 bid      | integer       |
 abalance | integer       |
 filler   | character(84) |
Indexes:
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)

\d pgbench_branches

   Table "public.pgbench_branches"
  Column  |     Type      | Modifiers
----------+---------------+-----------
 bid      | integer       | not null
 bbalance | integer       |
 filler   | character(88) |
Indexes:
    "pgbench_branches_pkey" PRIMARY KEY, btree (bid)
```

```
\d pgbench_tellers

    Table "public.pgbench_tellers"
  Column  |     Type      | Modifiers
----------+---------------+-----------
 tid      | integer       | not null
 bid      | integer       |
 tbalance | integer       |
 filler   | character(84) |
Indexes:
    "pgbench_tellers_pkey" PRIMARY KEY, btree (tid)

\d pgbench_history

          Table "public.pgbench_history"
 Column |            Type             | Modifiers
--------+-----------------------------+-----------
 tid    | integer                     |
 bid    | integer                     |
 aid    | integer                     |
 delta  | integer                     |
 mtime  | timestamp without time zone |
 filler | character(22)               |
```

The EXPLAIN statement shows the plan selected by the query planner. In the following example, aid is the primary key column. An indexed search is used on the pgbench_accounts_pkey index.

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE aid = 100;

                                        QUERY PLAN
-----------------------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.43..8.45 rows=
1 width=97)
   Index Cond: (aid = 100)
(2 rows)
```

In the following example, the FULL hint is used to force a full sequential scan. No index is used.

```
EXPLAIN SELECT /*+ FULL(pgbench_accounts) */ * FROM pgbench_accounts WHERE aid =
100;

                            QUERY PLAN
---------------------------------------------------------------------
 Seq Scan on pgbench_accounts  (cost=0.00..58781.69 rows=1 width=97)
   Filter: (aid = 100)
(2 rows)
```

In the following example, NO_INDEX hint forces a parallel sequential scan. No index is used .

```
EXPLAIN SELECT /*+ NO_INDEX(pgbench_accounts pgbench_accounts_pkey) */ * FROM
pgbench_accounts WHERE aid = 100;

                                   QUERY PLAN
-----------------------------------------------------------------------------------
```

```
 Gather  (cost=1000.00..45094.80 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..44094.70 rows=1 width=97)
         Filter: (aid = 100)
(4 rows)
```

In addition to the EXPLAIN statement in the prior examples, you can set the trace_hints configuration parameter to retrieve more detailed information regarding whether a hint is used by the planner.

```
SET trace_hints TO on;
```

In the following example, the SELECT statement with the NO_INDEX hint is repeated to illustrate the additional information that is generated after you set the trace_hints configuration parameters.

```
EXPLAIN SELECT /*+ NO_INDEX(pgbench_accounts pgbench_accounts_pkey) */ * FROM
pgbench_accounts WHERE aid = 100;

INFO:  [HINTS] Index Scan of [pgbench_accounts].[pgbench_accounts_pkey] rejected due
to NO_INDEX hint.
                                   QUERY PLAN
------------------------------------------------------------------------------------
 Gather  (cost=1000.00..45094.80 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..44094.70 rows=1 width=97)
         Filter: (aid = 100)
(4 rows)
```

If a hint is ignored, the INFO: [HINTS] line does not appear. This may indicate that some syntax errors or spelling errors exist in the hint. The following example shows that the index name is misspelled.

```
EXPLAIN SELECT /*+ NO_INDEX(pgbench_accounts pgbench_accounts_xxx) */ * FROM
pgbench_accounts WHERE aid = 100;

                                        QUERY PLAN
-----------------------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.43..8.45 rows=
1 width=97)
   Index Cond: (aid = 100)
```

```
(2 rows)
```

## 6.6.4 Specify a join order

You can use the ORDERED directive to instruct the query optimizer to join tables in the order in which they are listed in the FROM clause. If you do not include the ORDERED keyword, the query optimizer uses the order in which the tables are joined.

For example, the following statement allows the optimizer to choose the order in which the tables listed in the FROM clause to join these tables:

```
SELECT e.ename, d.dname, h.startdate
  FROM emp e, dept d, jobhist h
  WHERE d.deptno = e.deptno
  AND h.empno = e.empno;
```

The following statement instructs the optimizer to join the tables in specified order:

```
SELECT /*+ ORDERED */ e.ename, d.dname, h.startdate
  FROM emp e, dept d, jobhist h
  WHERE d.deptno = e.deptno
  AND h.empno = e.empno;
```

In the ORDERED version of the statement, a PolarDB database compatible with Oracle joins emp e with dept d and then joins the result of the previous join with jobhist h. Without the ORDERED directive, the query optimizer specifies the join order.

**Note:**

The ORDERED directive does not work for Oracle-style outer joins. These outer joins contain a plus sign (+).

## 6.6.5 Join relations hints

Three possible plans are available for you to join two tables:

- Nested loop join: A table is scanned once for every row in the other joined table.
- Merge sort join: Each table is sorted on the join attributes before the join starts. Then, these two tables are scanned in parallel and the matched rows are combined to form the join rows.
- Hash join: A table is scanned and its join attributes are loaded into a hash table. The join attributes of the table are used as hash keys. Then, the other joined table is scanned and its join attributes are used as hash keys to locate the matched rows from the first table.

The following table lists the optimizer hints that can be used to enable the planner to use a specified type of join plan.

**Table 6-3: Join hints**

| Hint | Description |
|---|---|
| USE_HASH(table [...]) | Uses a hash join for the table. |
| NO_USE_HASH(table [...]) | Does not use a hash join for the table. |
| USE_MERGE(table [...]) | Uses a merge sort join for the table. |
| NO_USE_MERGE(table [...]) | Does not use a merge sort join for the table. |
| USE_NL(table [...]) | Uses a nested loop join for the table. |
| NO_USE_NL(table [...]) | Does not use a nested loop join for the table . |

**Examples**

In the following example, the USE_HASH hint is used for a join on the pgbench_branches and pgbench_accounts tables. The query plan shows that a hash table is created from the join attribute of the pgbench_branches table to enable a hash join.

```
EXPLAIN SELECT /*+ USE_HASH(b) */ b.bid, a.aid, abalance FROM pgbench_branches b,
pgbench_accounts a WHERE b.bid = a.bid;

                               QUERY PLAN
--------------------------------------------------------------------------------------
 Hash Join  (cost=21.45..81463.06 rows=2014215 width=12)
   Hash Cond: (a.bid = b.bid)
   ->  Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215 width=12)
   ->  Hash  (cost=21.20..21.20 rows=20 width=4)
         ->  Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20 width=4)
(5 rows)
```

Afterward, the NO_USE_HASH(a b) hint forces the planner to use an approach other than hash tables. The result is a merge join.

```
EXPLAIN SELECT /*+ NO_USE_HASH(a b) */ b.bid, a.aid, abalance FROM pgbench_br
anches b, pgbench_accounts a WHERE b.bid = a.bid;

                                    QUERY PLAN
--------------------------------------------------------------------------------------------------
 Merge Join  (cost=333526.08..368774.94 rows=2014215 width=12)
   Merge Cond: (b.bid = a.bid)
   ->  Sort  (cost=21.63..21.68 rows=20 width=4)
         Sort Key: b.bid
         ->  Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20 width=4)
   ->  Materialize  (cost=333504.45..343575.53 rows=2014215 width=12)
         ->  Sort  (cost=333504.45..338539.99 rows=2014215 width=12)
               Sort Key: a.bid
```

```
                 ->  Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215 width=
12)
(9 rows)
```

Finally, the USE_MERGE hint forces the planner to use a merge join.

```
EXPLAIN SELECT /*+ USE_MERGE(a) */ b.bid, a.aid, abalance FROM pgbench_branches b,
pgbench_accounts a WHERE b.bid = a.bid;

                                   QUERY PLAN
------------------------------------------------------------------------------------------
 Merge Join  (cost=333526.08..368774.94 rows=2014215 width=12)
   Merge Cond: (b.bid = a.bid)
   ->  Sort  (cost=21.63..21.68 rows=20 width=4)
         Sort Key: b.bid
         ->  Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20 width=4)
   ->  Materialize  (cost=333504.45..343575.53 rows=2014215 width=12)
         ->  Sort  (cost=333504.45..338539.99 rows=2014215 width=12)
               Sort Key: a.bid
               ->  Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215 width=
12)
(9 rows)
```

In this three-table join example, the planner performs a hash join on the pgbench_branches and pgbench_history tables, and then performs a hash join of the result of the previous join with the pgbench_accounts table.

```
EXPLAIN SELECT h.mtime, h.delta, b.bid, a.aid FROM pgbench_history h, pgbench_br
anches b, pgbench_accounts a WHERE h.bid = b.bid AND h.aid = a.aid;

                                QUERY PLAN
-------------------------------------------------------------------------------------
 Hash Join  (cost=86814.29..123103.29 rows=500000 width=20)
   Hash Cond: (h.aid = a.aid)
   ->  Hash Join  (cost=21.45..15081.45 rows=500000 width=20)
         Hash Cond: (h.bid = b.bid)
         ->  Seq Scan on pgbench_history h  (cost=0.00..8185.00 rows=500000 width=20)
         ->  Hash  (cost=21.20..21.20 rows=20 width=4)
               ->  Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20 width=4)
   ->  Hash  (cost=53746.15..53746.15 rows=2014215 width=4)
         ->  Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215 width=4)
(9 rows)
```

You can use the hints to force a combination of a merge sort join and a hash join and modify the plan.

```
EXPLAIN SELECT /*+ USE_MERGE(h b) USE_HASH(a) */ h.mtime, h.delta, b.bid, a.aid FROM
 pgbench_history h, pgbench_branches b, pgbench_accounts a WHERE h.bid = b.bid AND
 h.aid = a.aid;

                                    QUERY PLAN
---------------------------------------------------------------------------------------------
 Hash Join  (cost=152583.39..182562.49 rows=500000 width=20)
   Hash Cond: (h.aid = a.aid)
   ->  Merge Join  (cost=65790.55..74540.65 rows=500000 width=20)
         Merge Cond: (b.bid = h.bid)
         ->  Sort  (cost=21.63..21.68 rows=20 width=4)
               Sort Key: b.bid
               ->  Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20 width=4)
         ->  Materialize  (cost=65768.92..68268.92 rows=500000 width=20)
               ->  Sort  (cost=65768.92..67018.92 rows=500000 width=20)
                     Sort Key: h.bid
                     ->  Seq Scan on pgbench_history h  (cost=0.00..8185.00 rows=500000 width=
20)
   ->  Hash  (cost=53746.15..53746.15 rows=2014215 width=4)
         ->  Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215 width=4)
(13 rows)
```

## 6.6.6 Global hints

Hints have been used in tables that are referenced in a SQL statement. Hints can also be used in tables that appear in a view if the view is referenced in a SQL statement. The hint does not appear in the view. Instead, the hint appears in the SQL statement that references the view.

If you want to specify a hint in a table within a view, provide the view and table names in dot notation within the hint argument list.

### Synopsis

```
hint(view.table)
```

### Parameters

| Parameter | Description |
|---|---|
| hint | Any of the hints in Table 1 or Table 2. |
| view | The name of the view that includes the table. |
| table | The table in which the hint is used. |

### Examples

A view named tx is created from the three-table join of pgbench_history, pgbench_branches, and pgbench_accounts. The final example in Join relations hints shows this view.

```
CREATE VIEW tx AS SELECT h.mtime, h.delta, b.bid, a.aid FROM pgbench_history h,
pgbench_branches b, pgbench_accounts a WHERE h.bid = b.bid AND h.aid = a.aid;
```

The following example shows the query plan generated by this view:

```
EXPLAIN SELECT * FROM tx;

                                   QUERY PLAN
------------------------------------------------------------------------------------
 Hash Join  (cost=86814.29..123103.29 rows=500000 width=20)
   Hash Cond: (h.aid = a.aid)
   ->  Hash Join  (cost=21.45..15081.45 rows=500000 width=20)
         Hash Cond: (h.bid = b.bid)
         ->  Seq Scan on pgbench_history h  (cost=0.00..8185.00 rows=500000 width=20)
         ->  Hash  (cost=21.20..21.20 rows=20 width=4)
               ->  Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20 width=4)
   ->  Hash  (cost=53746.15..53746.15 rows=2014215 width=4)
         ->  Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215 width=4)
(9 rows)
```

The hints used in this join at the end of Join relations hints can be used in the view. The following example shows this usage:

```
EXPLAIN SELECT /*+ USE_MERGE(tx.h tx.b) USE_HASH(tx.a) */ * FROM tx;

                                         QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Hash Join  (cost=152583.39..182562.49 rows=500000 width=20)
   Hash Cond: (h.aid = a.aid)
   ->  Merge Join  (cost=65790.55..74540.65 rows=500000 width=20)
         Merge Cond: (b.bid = h.bid)
         ->  Sort  (cost=21.63..21.68 rows=20 width=4)
               Sort Key: b.bid
               ->  Seq Scan on pgbench_branches b  (cost=0.00..21.20 rows=20 width=4)
         ->  Materialize  (cost=65768.92..68268.92 rows=500000 width=20)
               ->  Sort  (cost=65768.92..67018.92 rows=500000 width=20)
                     Sort Key: h.bid
                     ->  Seq Scan on pgbench_history h  (cost=0.00..8185.00 rows=500000 width=
20)
   ->  Hash  (cost=53746.15..53746.15 rows=2014215 width=4)
         ->  Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215 width=4)
(13 rows)
```

You can also use the hints in tables for subqueries. The following example shows this usage. When you query the emp table for the sample application, the emp table is joined with a subquery of the emp table identified by the alias b to list employees and their managers.

```
SELECT a.empno, a.ename, b.empno "mgr empno", b.ename "mgr ename" FROM emp a,
 (SELECT * FROM emp) b WHERE a.mgr = b.empno;

 empno | ename  | mgr empno | mgr ename
-------+--------+-----------+-----------
  7369 | SMITH  |      7902 | FORD
  7499 | ALLEN  |      7698 | BLAKE
  7521 | WARD   |      7698 | BLAKE
  7566 | JONES  |      7839 | KING
  7654 | MARTIN |      7698 | BLAKE
  7698 | BLAKE  |      7839 | KING
  7782 | CLARK  |      7839 | KING
  7788 | SCOTT  |      7566 | JONES
  7844 | TURNER |      7698 | BLAKE
  7876 | ADAMS  |      7788 | SCOTT
  7900 | JAMES  |      7698 | BLAKE
  7902 | FORD   |      7566 | JONES
  7934 | MILLER |      7782 | CLARK
```

```
(13 rows)
```

The following example shows the plan selected by the query planner:

```
EXPLAIN SELECT a.empno, a.ename, b.empno "mgr empno", b.ename "mgr ename"
FROM emp a, (SELECT * FROM emp) b WHERE a.mgr = b.empno;

                          QUERY PLAN
-----------------------------------------------------------------
 Hash Join  (cost=1.32..2.64 rows=13 width=22)
   Hash Cond: (a.mgr = emp.empno)
   ->  Seq Scan on emp a  (cost=0.00..1.14 rows=14 width=16)
   ->  Hash  (cost=1.14..1.14 rows=14 width=11)
         ->  Seq Scan on emp  (cost=0.00..1.14 rows=14 width=11)
(5 rows)
```

A hint can be used in the emp table within the subquery to perform an index scan instead of a table scan on the emp_pk index. The query plan is changed.

```
EXPLAIN SELECT /*+ INDEX(b.emp emp_pk) */ a.empno, a.ename, b.empno "mgr empno
", b.ename "mgr ename" FROM emp a, (SELECT * FROM emp) b WHERE a.mgr = b.empno;

                              QUERY PLAN
---------------------------------------------------------------------------
 Merge Join  (cost=4.17..13.11 rows=13 width=22)
   Merge Cond: (a.mgr = emp.empno)
   ->  Sort  (cost=1.41..1.44 rows=14 width=16)
         Sort Key: a.mgr
         ->  Seq Scan on emp a  (cost=0.00..1.14 rows=14 width=16)
   ->  Index Scan using emp_pk on emp  (cost=0.14..12.35 rows=14 width=11)
(6 rows)
```

## 6.6.7 Use the APPEND optimizer hint

By default, PolarDB databases compatible with Oracle add new data to the first available free-space in a table. The space is vacated by vacuumed records. The APPEND directive following an INSERT or SELECT statement instructs the server to bypass mid-table free space and affix new rows to the end of the table. This optimizer hint improves the performance of loading multiple entries.

The APPEND optimizer hint has the following syntax:

```
/*+APPEND*/
```

For example, the following statement compatible with Oracle databases instructs the server to append the data in the INSERT statement to the end of the sales table:

```
INSERT /*+APPEND*/ INTO sales VALUES
```

```
(10, 10, '01-Mar-2011', 10, 'OR');
```

PolarDB databases compatible with Oracle support the APPEND hint when you add multiple rows by using a single INSERT statement.

```
INSERT /*+APPEND*/ INTO sales VALUES
(20, 20, '01-Aug-2011', 20, 'NY'),
(30, 30, '01-Feb-2011', 30, 'FL'),
(40, 40, '01-Nov-2011', 40, 'TX');
```

The APPEND hint can also be included in the SELECT clause of an INSERT INTO statement.

```
INSERT INTO sales_history SELECT /*+APPEND*/ FROM sales;
```

## 6.6.8 Parallel hints

The PARALLEL optimizer hint is used to force parallel scanning.

The NO_PARALLEL optimizer hint prevents usage of a parallel scan.

### Synopsis

```
PARALLEL (table [ parallel_degree | DEFAULT ])

NO_PARALLEL (table)
```

### Description

Parallel scanning allows multiple background workers to simultaneously scan a table in a specified query. Compared with other methods such as a sequential scan, this scan provides improved performance.

### Parameters

| Parameter | Description |
|---|---|
| table | The table in which a parallel hint is used. |

| Parameter | Description |
|---|---|
| parallel_degree \| DEFAULT | The value of the parallel_degree parameter is a positive integer that specifies the desired number of workers to be used in a parallel scan. If this parameter is set, the smaller value between this parameter and the configuration parameter max_parallel_workers_per_gather is used as the planned number of workers. For more information about the max_parallel_workers_per_gather parameter, visit https://www.postgresql.org/docs/11/runtime-config-resource.html.<br>If DEFAULT is set, the maximum possible parallel degree is used.<br>If both parallel_degree and DEFAULT are omitted, the query optimizer determines the parallel degree. In this case, if the table parameter has been set with the parallel_workers storage parameter, the value of parallel_workers is used as the parallel degree. Otherwise, the optimizer uses the maximum possible parallel degree specified by DEFAULT. For more information about the parallel_workers storage parameter, visit https://www.postgresql.org/docs/11/sql-createtable.html.<br>Regardless of the circumstance, the parallel degree never exceeds the value of max_parallel_workers_per_gather. |

### Examples

The following configuration parameter settings are valid:

```
SHOW max_worker_processes;

 max_worker_processes
----------------------
 8
(1 row)

SHOW max_parallel_workers_per_gather;

 max_parallel_workers_per_gather
---------------------------------
 2
(1 row)
```

The following example shows the default scan on the pgbench_accounts table. A sequential scan is shown in the query plan.

```
SET trace_hints TO on;

EXPLAIN SELECT * FROM pgbench_accounts;
```

```
                              QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on pgbench_accounts  (cost=0.00..53746.15 rows=2014215 width=97)
(1 row)
```

The following example uses the PARALLEL hint. In the query plan, the Gather node that launches the background workers specifies that two workers are planned to be used.

**Note:**

If trace_hints is set to on, the INFO: [HINTS] lines are displayed to indicate that PARALLEL has been supported by pgbench_accounts and other hints. For the remaining examples, these lines are not displayed. These examples show the same output, where trace_hints is reset to off.

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;

INFO:  [HINTS] SeqScan of [pgbench_accounts] rejected due to PARALLEL hint.
INFO:  [HINTS] PARALLEL on [pgbench_accounts] accepted.
INFO:  [HINTS] Index Scan of [pgbench_accounts].[pgbench_accounts_pkey] rejected due
to PARALLEL hint.
                                    QUERY PLAN
---------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..244418.06 rows=2014215 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..41996.56 rows=839256 width=
97)
(3 rows)
```

The following example shows an increased value of max_parallel_workers_per_gather:

```
SET max_parallel_workers_per_gather TO 6;

SHOW max_parallel_workers_per_gather;

 max_parallel_workers_per_gather
---------------------------------
 6
(1 row)
```

The same query on pgbench_accounts is used again with no specified parallel degree in the PARALLEL hint. The number of planned workers has been determined by the optimizer and increased to 4.

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;

                                    QUERY PLAN
---------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..241061.04 rows=2014215 width=97)
   Workers Planned: 4
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..38639.54 rows=503554 width=
97)
```

```
(3 rows)
```

A value of 6 is specified for the parallel degree parameter of the PARALLEL hint. The value is returned as the planned number of workers in the following example:

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts 6) */ * FROM pgbench_accounts;

                                QUERY PLAN
------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..239382.52 rows=2014215 width=97)
   Workers Planned: 6
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..36961.03 rows=335702 width=
97)
(3 rows)
```

The same query is used with the DEFAULT setting for the parallel degree. The results indicate that the maximum allowable number of workers is planned.

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts DEFAULT) */ * FROM pgbench_accounts
;

                                QUERY PLAN
------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..239382.52 rows=2014215 width=97)
   Workers Planned: 6
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..36961.03 rows=335702 width=
97)
(3 rows)
```

The pgbench_accounts table is modified. In this table, the parallel_workers storage parameter is set to 3.

**Note:**

This format in which the ALTER TABLE statement sets the parallel_workers parameter is not compatible with Oracle databases.

The parallel_workers parameter is set by the PSQL \d+ statement.

```
ALTER TABLE pgbench_accounts SET (parallel_workers=3);

\d+ pgbench_accounts
                    Table "public.pgbench_accounts"
  Column  |     Type      | Modifiers | Storage  | Stats target | Description
----------+---------------+-----------+----------+--------------+-------------
 aid      | integer       | not null  | plain    |              |
 bid      | integer       |           | plain    |              |
 abalance | integer       |           | plain    |              |
 filler   | character(84) |           | extended |              |
Indexes:
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
```

```
Options: fillfactor=100, parallel_workers=3
```

If the PARALLEL hint is provided with no parallel degree, the returned number of planned workers is the value of the parallel_workers parameter.

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;

                                    QUERY PLAN
--------------------------------------------------------------------------------------
 Gather  (cost=1000.00..242522.97 rows=2014215 width=97)
   Workers Planned: 3
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..40101.47 rows=649747 width=
97)
(3 rows)
```

The parallel degree value or DEFAULT in the PARALLEL hint overwrites the parallel_workers setting.

The following example shows the NO_PARALLEL hint. If trace_hints is set to on, the INFO: [HINTS] message is displayed to indicate that the parallel scan has been rejected due to the NO_PARALLEL hint.

```
EXPLAIN SELECT /*+ NO_PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;
INFO:  [HINTS] Parallel SeqScan of [pgbench_accounts] rejected due to NO_PARALLEL hint.
                              QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on pgbench_accounts  (cost=0.00..53746.15 rows=2014215 width=97)
(1 row)
```

## 6.6.9 Conflicting hints

If a statement includes two or more conflicting hints, the server ignores the conflicting hints . The following table lists the hints that are conflicting with each other.

| Hint | Conflicting hints |
|---|---|
| ALL_ROWS | FIRST_ROWS - all formats |
| FULL(table) | INDEX(table [ index ])<br>PARALLEL(table [ degree ]) |
| INDEX(table) | FULL(table)<br>NO_INDEX(table)<br>PARALLEL(table [ degree ]) |

| Hint | Conflicting hints |
|---|---|
| INDEX(table index) | FULL(table)<br>NO_INDEX(table index)<br>PARALLEL(table [ degree ]) |
| PARALLEL(table [ degree ]) | FULL(table)<br>INDEX(table)<br>NO_PARALLEL(table) |
| USE_HASH(table) | NO_USE_HASH(table) |
| USE_MERGE(table) | NO_USE_MERGE(table) |
| USE_NL(table) | NO_USE_NL(table) |

# 7 Stored Procedure Language

## 7.1 Overview

This topic describes the Stored Procedure Language (SPL). SPL is a highly productive, procedural programming language for writing custom procedures, functions, triggers, and packages for PolarDB databases compatible with Oracle. SPL provides:

- Full procedural programming functionality to complement the SQL language
- A common language to create stored procedures, functions, triggers, and packages for PolarDB databases compatible with Oracle
- A seamless development and testing environment
- The use of reusable code
- Ease of use

This chapter describes the basic elements of an SPL program, and then provides an overview of the organization of an SPL program and how it is used to create a procedure or a function.

## 7.2 Basic SPL elements

### 7.2.1 Character sets

SPL programs are written with the following set of characters:

- Uppercase letters A to Z and lowercase letters a to z
- Digits 0 to 9
- Special characters ( ) + - * / < > = ! ~ ^ ; : . ' @ % , " # $ & _ | { } ? [ ]
- White space characters including tabs, spaces, and carriage returns

Identifiers, expressions, statements, and control structures that comprise the SPL language are written with these characters.

**Note:**
The data that can be manipulated by an SPL program is determined by the character set supported by the database encoding.

## 7.2.2 Case sensitivity

Keywords and user-defined identifiers that are used in an SPL program are case insensitive. For example, the statement `DBMS_OUTPUT.PUT_LINE('Hello World');` is interpreted to mean the same thing as `dbms_output.put_line('Hello World');`, `Dbms_Output.Put_Line('Hello World');`, or `DBMS_output.Put_line('Hello World');`.

However, character and string constants, data retrieved from the PolarDB database compatible with Oracle, or data obtained from other external sources are case sensitive. The following output is generated by the `DBMS_OUTPUT.PUT_LINE('Hello World!') ;` statement:

```
Hello World!
```

However, the following output is generated by the `DBMS_OUTPUT.PUT_LINE('HELLO WORLD!') ;` statement:

```
HELLO WORLD!
```

## 7.2.3 Identifiers

Identifiers are user-defined names that are used to identify various elements of an SPL program including variables, cursors, labels, programs, and parameters. The syntax rules for valid identifiers are the same as that for identifiers in the SQL language.

An identifier must not be the same as an SPL keyword or a keyword of the SQL language. The following content shows examples of valid identifiers:

```
x
last___name
a_$_Sign
Many$$$$$$$$signs_____
THIS_IS_AN_EXTREMELY_LONG_NAME
A1
```

## 7.2.4 Qualifiers

A qualifier is a name that specifies the owner or context of an entity that is the object of the qualification. A qualified object is specified as the qualifier name followed by a dot with no intervening white space, followed by the name of the object being qualified with no intervening white space. This syntax is called dot notation.

The syntax of a qualified object is as follows:

```
qualifier. [ qualifier. ]... object
```

qualifier is the name of the object owner. object is the name of the entity that belongs to qualifier. There can be a chain of qualifications where the preceding qualifier owns the entity identified by the subsequent qualifiers and object.

Almost any identifier can be qualified. What an identifier is qualified by depends on what the identifier represents and the context of its usage.

The following content shows examples of qualification:

- Procedure and function names qualified by the schema to which they belong, such as schema_name.procedure_name (...).
- Trigger names qualified by the schema to which they belong, such as schema_name.trigger_name.
- Column names qualified by the table to which they belong, such as emp.empno.
- Table names qualified by the schema to which they belong, such as public.emp.
- Column names qualified by table and schema, such as public.emp.empno.

Generally, wherever a name appears in the syntax of an SPL statement, its qualified name can be used as well. A qualified name would only be used if there is ambiguity associated with the name. For example, if two procedures with the same name belonging to two different schemas are invoked from within a program or if the same name is used for a table column and SPL variable within the same program.

You must avoid using qualified names. This topic uses the following conventions to avoid naming conflicts:

- All variables declared in the declaration section of an SPL program are prefixed by v_, such as v_empno.
- All formal parameters declared in a procedure or function definition are prefixed by p_, such as p_empno.
- Column names and table names do not have any special prefix conventions, such as column empno in table emp.

## 7.2.5 Constants

Constants or literals are fixed values that can be used in SPL programs to represent values of various types, such as numbers, strings, and dates. Constants come in the following types:

- Numeric (Integer and Real)
- Character and String
- Date/time

## 7.2.6 User-defined PL/SQL subtypes

PolarDB databases compatible with Oracle support user-defined PL/SQL subtypes and subtype aliases. A subtype is a data type with an optional set of constraints that restrict the values that can be stored in a column of that type. The rules that apply to the type on which the subtype is based are still enforced, but you can use other constraints to place limits on the precision or scale of values stored in the type.

You can define a subtype in the declaration of a PL function, procedure, anonymous block, or package. The syntax is as follows:

```
SUBTYPE subtype_name IS type_name[(constraint)] [NOT NULL]
```

where constraint is:

```
{precision [, scale]} | length
```

where:

- subtype_name: specifies the name of the subtype.
- type_name: specifies the name of the original type on which the subtype is based.

  type_name can be:

  - The name of any of the type supported by PolarDB databases compatible with Oracle.
  - The name of a composite type.
  - A column anchored by a %TYPE operator.
  - The name of another subtype.

Include the constraint clause to define restrictions for types that support precision or scale.

- precision: specifies the total number of digits permitted in a value of the subtype.
- scale: specifies the number of fractional digits permitted in a value of the subtype.

- length: specifies the total length permitted in a value of CHARACTER, VARCHAR, or TEXT base types.

Include the NOT NULL clause to specify that NULL values may not be stored in a column of the specified subtype.

Note that a subtype that is based on a column will inherit the column size constraints, but the subtype will not inherit NOT NULL or CHECK constraints.

**Unconstrained subtypes**

To create an unconstrained subtype, use the SUBTYPE statement to specify the new subtype name and the name of the type on which the subtype is based. For example, the following statement creates a subtype named address that has all of the attributes of the type, CHAR:

```
SUBTYPE address IS CHAR;
```

You can also create a subtype (constrained or unconstrained) of another subtype:

```
SUBTYPE cust_address IS address NOT NULL;
```

This statement creates a subtype named cust_address that shares all of the attributes of the address subtype. Include the NOT NULL clause to specify that the value of the cust_address may not be NULL.

**Constrained subtypes**

Include a length value when creating a subtype that is based on a character type to define the maximum length of the subtype. Example:

```
SUBTYPE acct_name IS VARCHAR (15);
```

This example creates a subtype named acct_name that is based on a VARCHAR data type, but is limited to 15 characters in length.

Include values for precision (to specify the maximum number of digits in a value of the subtype) and optionally, scale (to specify the number of digits to the right of the decimal point) when constraining a numeric base type. Example:

```
SUBTYPE acct_balance IS NUMBER (5, 2);
```

This example creates a subtype named acct_balance that shares all of the attributes of a NUMBER type, but that cannot exceed 3 digits to the left of the decimal point and 2 digits to the right of the decimal.

An argument declaration (in a function or procedure header) is a formal argument. The value passed to a function or procedure is an actual argument. When invoking a function or procedure, the caller provides 0 or more actual arguments. Each actual argument is assigned to a formal argument that holds the value within the body of the function or procedure.

If a formal argument is declared as a constrained subtype:

- PolarDB databases compatible with Oracle do not enforce subtype constraints when assigning an actual argument to a formal argument during the invoking of a function.
- PolarDB databases compatible with Oracle enforce subtype constraints when assigning an actual argument to a formal argument during the invoking of a procedure.

**Use the %TYPE operator**

You can use the %TYPE notation to declare a subtype anchored to a column. Example:

```
SUBTYPE emp_type IS emp.empno%TYPE
```

This statement creates a subtype named emp_type whose base type matches the type of the empno column in the emp table. A subtype that is based on a column will share the column size constraints, while NOT NULL, and CHECK constraints are not inherited.

**Subtype conversion**

Unconstrained subtypes are aliases for the type on which they are based. Any type variable of unconstrained subtype is interchangeable with a variable of the base type without conversion, and vice versa.

A variable of a constrained subtype may be interchanged with a variable of the base type without conversion, but a variable of the base type can only be interchanged with a constrained subtype if the variable of the base type complies with the constraints of the subtype. A variable of a constrained subtype can be implicitly converted to another subtype if it is based on the same subtype, and the constraint values are within the values of the subtype to which it is being converted.

## 7.3 SPL programs

## 7.3.1 Overview

SPL is a procedural, block-structured language. You can use SPL to create four types of programs, including procedures, functions, triggers, and packages.

In addition, SPL is used to create subprograms. A subprogram refers to a subprocedure or a subfunction, which are nearly identical in appearance to procedures and functions, but differ in that procedures and functions are standalone programs, which are individually stored in the database and can be invoked by other SPL programs or from PSQL. Subprograms can only be invoked from within the standalone program in which they are created.

## 7.3.2 SPL block structures

Regardless of whether the program is a procedure, function, subprogram, or trigger, an SPL program has the same block structure. A block consists of up to three sections: an optional declaration section, a mandatory executable section, and an optional exception section. A block must have at least an executable section that consists of one or more SPL statements within the keywords, BEGIN and END.

The optional declaration section is used to declare variables, cursors, types, and subprograms that are used by the statements within the executable and exception sections. Declarations appears only prior to the BEGIN keyword of the executable section . Depending on the context of where the block is used, the declaration section may begin with the keyword DECLARE.

You can include an exception section within the BEGIN - END block. The exception section begins with the keyword EXCEPTION, and continues until the end of the block in which it appears. If an exception is thrown by a statement within the block, program control goes to the exception section where the thrown exception may or may not be handled depending on the exception and the contents of the exception section.

The following content shows the general structure of a block:

```
[ [ DECLARE ]
 pragmas
 declarations ]
    BEGIN
 statements
  [ EXCEPTION
      WHEN exception_condition THEN
 statements [, ...] ]
    END;
```

pragmas are the directives (AUTONOMOUS_TRANSACTION is the currently supported pragma). declarations are one or more variable, cursor, type, or subprogram declarations that are local to the block. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations. Each declaration must be terminated by a semicolon (;). The use of the keyword DECLARE depends on the context in which the block appears.

statements are one or more SPL statements. Each statement must be terminated by a semicolon (;). The end of the block indicated by the keyword END must also be terminated by a semicolon (;).

If present, the keyword EXCEPTION marks the beginning of the exception section. exception_condition is a conditional expression testing for one or more types of exceptions . If an exception matches one of the exceptions in exception_condition, the statements following the WHEN exception_condition clause are executed. There may be one or more WHEN exception_condition clauses that are followed by statements. Note: A BEGIN/END block can be considered as a statement. Therefore, blocks can be nested. The exception section may also contain nested blocks.

The following content describes the simplest possible block consisting of the NULL statement within the executable section. The NULL statement is an executable statement that does not perform any operations.

```
BEGIN
    NULL;
END;
```

The following content describes a block that contains a declaration section as well as the executable section:

```
DECLARE
    v_numerator     NUMBER(2);
    v_denominator   NUMBER(2);
    v_result        NUMBER(5,2);
BEGIN
    v_numerator := 75;
    v_denominator := 14;
    v_result := v_numerator / v_denominator;
    DBMS_OUTPUT.PUT_LINE(v_numerator || ' divided by ' || v_denominator ||
        ' is ' || v_result);
END;
```

In this example, three numeric variables are declared for the data type NUMBER. Values are assigned to two of the variables, and one number is divided by the other, storing the results in the third variable that is then displayed. If executed, the following output is generated:

```
75 divided by 14 is 5.36
```

The following content describes a block that contains a declaration, an executable, and an exception:

```
DECLARE
    v_numerator     NUMBER(2);
    v_denominator   NUMBER(2);
    v_result        NUMBER(5,2);
BEGIN
    v_numerator := 75;
    v_denominator := 0;
    v_result := v_numerator / v_denominator;
    DBMS_OUTPUT.PUT_LINE(v_numerator || ' divided by ' || v_denominator ||
        ' is ' || v_result);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An exception occurred');
END;
```

The following output shows that the statement within the exception section is executed as the result of the division by zero:

```
An exception occurred
```

### 7.3.3 Anonymous blocks

Blocks are typically written as part of a procedure, function, subprogram, or trigger. Procedure, function, and trigger programs are named and stored in the database for reuse . For quick (one-time) execution (such as testing), you can enter the block without providing a name or storing it in the database.

A block of this type is called an anonymous block. An anonymous block is unnamed and is not stored in the database. After the block has been executed and erased from the application buffer, it cannot be re-executed unless the block code is re-entered into the application.

Typically, the same block of code will be re-executed many times. To run a block of code repeatedly without the necessity of re-entering the code each time, you can turn an anonymous block into a procedure or function by making some simple modifications. The following topics discuss how to create a procedure or function that can be stored in the database and invoked repeatedly by another procedure, function, or application program.

## 7.3.4 Create a procedure

Procedures are standalone SPL programs that are invoked or called as an individual SPL program statement. When called, procedures may optionally receive values from the caller in the form of input parameters and optionally return values to the caller in the form of output parameters.

The CREATE PROCEDURE statement defines and names a standalone procedure that will be stored in the database.

If a schema name is included, the procedure is created in the specified schema. Otherwise , it is created in the current schema. The name of the new procedure must not match any existing procedure with the same input argument types in the same schema. However, procedures of different input argument types may share a name. This is called overloading . Overloading of procedures is a feature of the PolarDB database compatible with Oracle - overloading of stored, standalone procedures is not compatible with Oracle databases.

To update the definition of an existing procedure, use CREATE OR REPLACE PROCEDURE. It is not possible to change the name or argument types of a procedure this way (if you tried , you would actually be creating a new, distinct procedure). When using OUT parameters, you cannot change the types of any OUT parameters except by dropping the procedure.

```
CREATE [OR REPLACE] PROCEDURE name [ (parameters) ]
   [
        IMMUTABLE
      | STABLE
      | VOLATILE
      | DETERMINISTIC
      | [ NOT ] LEAKPROOF
      | CALLED ON NULL INPUT
      | RETURNS NULL ON NULL INPUT
      | STRICT
      | [ EXTERNAL ] SECURITY INVOKER
      | [ EXTERNAL ] SECURITY DEFINER
      | AUTHID DEFINER
      | AUTHID CURRENT_USER
      | PARALLEL { UNSAFE | RESTRICTED | SAFE }
      | COST execution_cost
      | ROWS result_rows
      | SET configuration_parameter
        { TO value | = value | FROM CURRENT }
    ...]
{ IS | AS }
    [ PRAGMA AUTONOMOUS_TRANSACTION; ]
    [ declarations ]
  BEGIN
 statements
```

```
END [ name ];
```

**Table 7-1: Arguments**

| Argument | Description |
|---|---|
| name | name is the identifier of the procedure. |
| parameters | parameters is a list of formal parameters. |
| declarations | declarations are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations. |
| statements | statements are SPL program statements (the BEGIN - END block may contain an EXCEPTION section). |
| IMMUTABLE<br>STABLE<br>VOLATILE | These attributes inform the query optimizer about the behavior of the procedure. You can specify only one option. VOLATILE is the default behavior.<br>• IMMUTABLE specifies that the procedure cannot modify the database and always reaches the same result when given the same argument values. It does not do database lookups or use information not directly present in its argument list. If you include this clause, any call of the procedure with all-constant arguments can be immediately replaced with the procedure value.<br>• STABLE specifies that the procedure cannot modify the database, and that within a single table scan, it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is suitable for procedures that depend on database lookups and parameter variables such as the current time zone.<br>• VOLATILE specifies that the procedure value can change even within a single table scan, so no optimizations can be made. Note that any function that has side effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away. |
| DETERMINISTIC | DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC procedure cannot modify the database and always reaches the same result when given the same argument values. It does not do database lookups or use information not directly present in its argument list. If you include this clause, any call of the procedure with all-constant arguments can be immediately replaced with the procedure value. |

| Argument | Description |
|---|---|
| [ NOT ] LEAKPROOF | LEAKPROOF has no side effects, and reveals no information about the values used to call the procedure. |
| CALLED ON NULL INPUT<br>RETURNS NULL ON NULL INPUT<br>STRICT | • CALLED ON NULL INPUT indicates that the procedure is called normally when some of its arguments are NULL. This is the default value. If necessary, the author needs to check for NULL values and respond appropriately.<br>• RETURNS NULL ON NULL INPUT, or STRICT specifies that the procedure always returns NULL whenever any of its arguments are NULL. If these clauses are specified, the procedure is not executed when there are NULL arguments. A NULL result is assumed automatically. |
| [ EXTERNAL ] SECURITY DEFINER | SECURITY DEFINER specifies that the procedure will execute with the privileges of the user that created it. This is the default value . The keyword EXTERNAL is allowed for SQL conformance, but is optional. |
| [ EXTERNAL ] SECURITY INVOKER | SECURITY INVOKER specifies that the procedure will execute with the privileges of the user that calls it. The keyword EXTERNAL is allowed for SQL conformance, but is optional. |
| AUTHID DEFINER<br>AUTHID CURRENT_USER | • AUTHID DEFINER is a synonym for [EXTERNAL] SECURITY DEFINER. If the AUTHID clause is omitted or if AUTHID DEFINER is specified, the rights of the procedure owner are used to determine access privileges to database objects.<br>• AUTHID CURRENT_USER is a synonym for [EXTERNAL] SECURITY INVOKER. If AUTHID CURRENT_USER is specified, the rights of the current user executing the procedure are used to determine access privileges. |

| Argument | Description |
|---|---|
| PARALLEL { UNSAFE \| RESTRICTED \| SAFE } | PARALLEL enables the use of parallel sequential scans (parallel mode). In contrast to a serial sequential scan, a parallel sequential scan uses multiple workers to scan a relation in parallel during a query.<br>• When set to UNSAFE, the procedure cannot be executed in parallel mode. The presence of such a procedure forces a serial execution plan. This is the default setting if the PARALLEL clause is omitted.<br>• When set to RESTRICTED, the procedure can be executed in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation will not be chosen for parallelism.<br>• When set to SAFE, the procedure can be executed in parallel mode with no restriction. |
| COST execution_cost | execution_cost is a positive number giving the estimated execution cost for the procedure. Units: cpu_operator_cost. If the procedure returns a set, this is the cost per returned row. Larger values cause the planner to try to avoid evaluating the function more often than necessary. |
| ROWS result_rows | result_rows is a positive number giving the estimated number of rows that the planner expects the procedure to return. This is allowed only when the procedure is declared to return a set. The default assumption is 1,000 rows. |
| SET configuration_parameter { TO value \| = value \| FROMCURRENT } | SET causes the specified configuration parameter to be set to the specified value when the procedure is entered, and then restored to its prior value when the procedure exits. SET FROM CURRENT saves the current parameter value of the session as the value to be applied when the procedure is entered.<br>If a SET clause is attached to a procedure, then the effects of a SET LOCAL statement executed inside the procedure for the same variable are restricted to the procedure. The prior value of the configuration parameter is restored at procedure exit. An ordinary SET statement (without LOCAL) overrides the SET clause, much as it would do for a previous SET LOCAL statement, with the effects of such a statement persisting after procedure exit, unless the current transaction is rolled back. |

| Argument | Description |
|---|---|
| PRAGMA AUTONOMOUS _TRANSACTION | PRAGMA AUTONOMOUS_TRANSACTION is the directive that sets the procedure as an autonomous transaction. |

The STRICT, LEAKPROOF, PARALLEL, COST, ROWS, and SET keywords provide extended functionality for PolarDB databases compatible with Oracle but are not supported by Oracle .

**Note:**

By default, stored procedures are created as SECURITY DEFINERS, but when written in PL/pgSQL, the stored procedures are created as SECURITY INVOKERS.

**Example**

The following example shows a simple procedure that takes no parameters:

```
CREATE OR REPLACE PROCEDURE simple_procedure
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('That''s all folks!') ;
END simple_procedure;
```

The procedure is stored in the database by entering the procedure code ina PolarDB database compatible with Oracle.

The following example describes how to use the AUTHID DEFINER and SET clauses in a procedure declaration. The update_salary procedure conveys the privileges of the role that defined the procedure to the role that is calling the procedure while the procedure executes :

```
CREATE OR REPLACE PROCEDURE update_salary(id INT, new_salary NUMBER)
  SET SEARCH_PATH = 'public' SET WORK_MEM = '1MB'
  AUTHID DEFINER IS
BEGIN
  UPDATE emp SET salary = new_salary WHERE emp_id = id;
END;
```

Include the SET clause to set the search path of the procedure to public and the work memory to 1 MB. Other procedures, functions, and objects are affected by these settings.

In this example, the AUTHID DEFINER clause temporarily grants privileges to a role that may not be allowed to execute the statements within the procedure. To instruct the server to use the privileges associated with the role invoking the procedure, replace the AUTHID DEFINER clause with the AUTHID CURRENT_USER clause.

## 7.3.5 Call a procedure

You can invoke a procedure from another SPL program by specifying the procedure name and parameters (if any) followed by a semicolon (;) in the following format:

```
name [ ([ parameters ]) ];
```

where:

- name is the identifier of the procedure.
- parameters is a list of actual parameters.

**Note:**

If there are no actual parameters to be passed, the procedure may be called with an empty parameter list, or the opening and closing parenthesis may be omitted entirely.

The syntax for calling a procedure is the same as in the preceding syntax diagram when executing it with the EXEC statement in PSQL or PolarDB databases compatible with Oracle *Plus.

The following example describes how to call the procedure from an anonymous block:

```
BEGIN
    simple_procedure;
END;

That's all folks!
```

**Note:**

Each application has its own unique way to call a procedure. For example, in a Java application, the application programming interface JDBC is used.

## 7.3.6 Delete a procedure

You can use the DROP PROCEDURE statement to delete a procedure form the database.

```
DROP PROCEDURE [ IF EXISTS ] name [ (parameters) ]
    [ CASCADE | RESTRICT ];
```

name is the name of the procedure to be dropped.

**Note:**

In a PolarDB database compatible with Oracle, you need to specify the parameter list under certain circumstances such as if this is an overloaded procedure. Oracle requires that the parameter list always be omitted.

Usage of IF EXISTS, CASCADE, or RESTRICT is not compatible with Oracle databases. For more information about these options, see the DROP PROCEDURE statement in the Database Compatibility for Oracle Developers Reference Guide.

The following example describes how to drop the previously created procedure:

```
DROP PROCEDURE simple_procedure;
```

## 7.3.7 Create a function

Functions are standalone SPL programs that are invoked as expressions. When evaluated, a function returns a value that is substituted in the expression in which the function is embedded. Functions can optionally take values from the calling program in the form of input parameters. In addition to returning a value by itself, a function can optionally return additional values to the caller in the form of output parameters. However, we recommend that you do not use output parameters in functions in programming practice.

The CREATE FUNCTION statement defines and names a standalone function that will be stored in the database.

If a schema name is included, the function is created in the specified schema. Otherwise, it is created in the current schema. The name of the new function must not match any existing function with the same input argument types in the same schema. However, functions of different input argument types can share a name. This is called overloading. Overloading of functions is a feature of PolarDB databases compatible with Oracle - overloading of stored, standalone functions is not compatible with Oracle databases.

To update the definition of an existing function, use CREATE OR REPLACE FUNCTION. You cannot change the name or argument types of a function in this way. If you tried, you actually created a new, distinct function. Also, CREATE OR REPLACE FUNCTION does not change the return type of an existing function. To do that, you must drop and recreate the function. Also when using OUT parameters, you cannot change the types of any OUT parameters except by dropping the function.

```
CREATE [ OR REPLACE ] FUNCTION name [ (parameters) ]
  RETURN data_type
   [
        IMMUTABLE
      | STABLE
      | VOLATILE
```

```
     | DETERMINISTIC
     | [ NOT ] LEAKPROOF
     | CALLED ON NULL INPUT
     | RETURNS NULL ON NULL INPUT
     | STRICT
     | [ EXTERNAL ] SECURITY INVOKER
     | [ EXTERNAL ] SECURITY DEFINER
     | AUTHID DEFINER
     | AUTHID CURRENT_USER
     | PARALLEL { UNSAFE | RESTRICTED | SAFE }
     | COST execution_cost
     | ROWS result_rows
     | SET configuration_parameter
       { TO value | = value | FROM CURRENT }
  ...]
{ IS | AS }
    [ PRAGMA AUTONOMOUS_TRANSACTION; ]
    [ declarations ]
  BEGIN
    statements
  END [ name ];
```

| Argument | Description |
|---|---|
| name | name is the identifier of the function. |
| parameters | parameters is a list of formal parameters. |
| data_type | data_type is the data type of the value returned by the RETURN statement of the function. |
| declarations | declarations are variable, cursor, type, or subprogram declarations . If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations. |
| statements | statements are SPL program statements. The BEGIN - END block can contain an EXCEPTION section. |

| Argument | Description |
|---|---|
| IMMUTABLE<br>STABLE<br>VOLATILE | These attributes inform the query optimizer about the behavior of the function. You can specify only one option. VOLATILE is the default behavior.<br>• IMMUTABLE specifies that the function cannot modify the database and always reaches the same result when given the same argument values. It does not do database lookups or use information that is not directly present in its argument list in any other way. If you include this clause, any call of the function with all-constant arguments can be immediately replaced with the function value.<br>• STABLE specifies that the function cannot modify the database, and that within a single table scan, it will consistently return the same result for the same argument values. However, its result could change across SQL statements. This is the suitable option for functions that depend on database lookups and parameter variables such as the current time zone.<br>• VOLATILE specifies that the function value can change even in a single table scan, so no optimizations can be made. Note that any function that has side effects must be classified as a volatile function, even if its result is predictable, to prevent calls from being optimized away. |
| DETERMINISTIC | DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC function cannot modify the database and always reaches the same result when given the same argument values. It does not do database lookups or use information that is not directly present in its argument list in any other way. If you include this clause, any call of the function with all-constant arguments can be immediatel y replaced with the function value. |
| [ NOT ] LEAKPROOF | LEAKPROOF has no side effects, and reveals no information about the values used to call the function. |
| CALLED ON NULL INPUT<br>RETURNS NULL ON NULL INPUT<br>STRICT | • CALLED ON NULL INPUT specifies that the procedure is called normally when some of its arguments are NULL. CALLED ON NULL INPUT is the default value. If necessary, the author needs to check for NULL values and respond appropriately.<br>• RETURNS NULL ON NULL INPUT or STRICT specifies that the procedure returns NULL if any of its arguments is NULL. If these clauses are specified, the procedure is not executed when there are NULL arguments. A NULL result is assumed automatically. |

| Argument | Description |
|---|---|
| [ EXTERNAL ] SECURITY DEFINER | SECURITY DEFINER specifies that the function will execute with the privileges of the user that created it. SECURITY DEFINER is the default value. The keyword EXTERNAL is allowed for SQL conformance. This is optional. |
| [ EXTERNAL ] SECURITY INVOKER | SECURITY INVOKER specifies that the function will execute with the privileges of the user that calls it. The keyword EXTERNAL is allowed for SQL conformance. This is optional. |
| AUTHID DEFINER<br>AUTHID CURRENT_USER | AUTHID DEFINER is a synonym for [EXTERNAL] SECURITY DEFINER. If the AUTHID clause is omitted or if AUTHID DEFINER is specified, the rights of the function owner are used to determine access privileges to database objects.<br>AUTHID CURRENT_USER is a synonym for [EXTERNAL] SECURITY INVOKER. If AUTHID CURRENT_USER is specified, the rights of the current user who is executing the function are used to determine access privileges. |
| PARALLEL { UNSAFE \| RESTRICTED \| SAFE } | PARALLEL enables the use of parallel sequential scans (parallel mode). In contrast to a serial sequential scan, a parallel sequential scan uses multiple workers to scan a relation in parallel during a query.<br>• When set to UNSAFE, the function cannot be executed in a parallel mode. The presence of such a function in a SQL statement forces a serial execution plan. If the PARALLEL clause is omitted, this is the default setting.<br>• When set to RESTRICTED, the function can be executed in a parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation will not be chosen for parallelism.<br>• When set to SAFE, the function can be executed in a parallel mode with no restriction. |
| COST execution_cost | execution_cost is a positive number giving the estimated execution cost for the function. Unit: cpu_operator_cost. If the function returns a set, this is the cost per returned row. Larger values cause the planner to try to avoid evaluating the function more often than necessary. |

| Argument | Description |
|---|---|
| ROWS result_rows | result_rows is a positive number giving the estimated number of rows that the planner expects the function to return. This is allowed only when the function is declared to return a set. The default assumption is 1,000 rows. |
| SET configuration_parameter { TO value \| = value \| FROMCURRENT } | SET causes the specified configuration parameter to be set to the specified value when the function is entered, and then restored to its prior value when the function exits. SET FROM CURRENT saves the current parameter value of the session as the value to be applied when the function is entered.<br>If a SET clause is attached to a function, the effects of a SET LOCAL statement executed inside the function for the same variable are restricted to the function. The prior configuration parameter value is restored when the function exits. An ordinary SET statement ( without LOCAL) overrides the SET clause, much as it would do for a previous SET LOCAL statement, with the effects of such a statement persisting after procedure exit, unless the current transaction is rolled back. |
| PRAGMA AUTONOMOUS _TRANSACTION | PRAGMA AUTONOMOUS_TRANSACTION is the directive that sets the function to an autonomous transaction. |

**Note:**

The STRICT, LEAKPROOF, PARALLEL, COST, ROWS, and SET keywords provide extended functionality for PolarDB databases compatible with Oracle but are not supported by Oracle.

### Examples

The following example describes a simple function that takes no parameters:

```
CREATE OR REPLACE FUNCTION simple_function
   RETURN VARCHAR2
IS
BEGIN
   RETURN 'That''s All Folks!' ;
```

```
END simple_function;
```

The following example describes a function that takes two input parameters. Parameters are discussed in subsequent topics.

```
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal           NUMBER,
    p_comm          NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END emp_comp;
```

The following example describes how to use the AUTHID CURRENT_USER clause and STRICT keyword in a function declaration.

```
CREATE OR REPLACE FUNCTION dept_salaries(dept_id int) RETURN NUMBER
  STRICT
  AUTHID CURRENT_USER
BEGIN
  RETURN QUERY (SELECT sum(salary) FROM emp WHERE deptno = id);
END;
```

Include the STRICT keyword to instruct the server to return NULL if any input parameter passed is NULL. If the NULL value is passed, the function will not execute.

The dept_salaries function executes with the privileges of the role that is calling the function. If the current user has insufficient privileges to perform the SELECT statement to query the emp table and display employee salaries, the function will report an error. To instruct the server to use the privileges associated with the role that defined the function, replace the AUTHID CURRENT_USER clause with the AUTHID DEFINER clause.

## 7.3.8 Call a function

A function can be used anywhere an expression can appear within an SPL statement. You can invoke a function by specifying its name followed by its parameters enclosed in parentheses (), if any.

```
name [ ([ parameters ]) ]
```

name is the name of the function. parameters is a list of actual parameters.

If there are no actual parameters to be passed, the function may be called with an empty parameter list, or the opening and closing parenthesis may be omitted entirely.

The following example shows how to call the function from another SPL program:

```
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE(simple_function);
END;

That's All Folks!
```

The following example describes how functions are generally used in an SQL statement:

```
SELECT empno "EMPNO", ename "ENAME", sal "SAL", comm "COMM",
    emp_comp(sal, comm) "YEARLY COMPENSATION" FROM emp;

 EMPNO | ENAME  |   SAL   |  COMM   | YEARLY COMPENSATION
-------+--------+---------+---------+---------------------
  7369 | SMITH  |  800.00 |         |            19200.00
  7499 | ALLEN  | 1600.00 |  300.00 |            45600.00
  7521 | WARD   | 1250.00 |  500.00 |            42000.00
  7566 | JONES  | 2975.00 |         |            71400.00
  7654 | MARTIN | 1250.00 | 1400.00 |            63600.00
  7698 | BLAKE  | 2850.00 |         |            68400.00
  7782 | CLARK  | 2450.00 |         |            58800.00
  7788 | SCOTT  | 3000.00 |         |            72000.00
  7839 | KING   | 5000.00 |         |           120000.00
  7844 | TURNER | 1500.00 |    0.00 |            36000.00
  7876 | ADAMS  | 1100.00 |         |            26400.00
  7900 | JAMES  |  950.00 |         |            22800.00
  7902 | FORD   | 3000.00 |         |            72000.00
  7934 | MILLER | 1300.00 |         |            31200.00
(14 rows)
```

## 7.3.9 Delete a function

A function can be deleted from the database with the DROP FUNCTION statement.

```
DROP FUNCTION [ IF EXISTS ] name [ (parameters) ]
    [ CASCADE | RESTRICT ];
```

name is the name of the function to be deleted.

**Note:**

In a PolarDB database compatible with Oracle, you need to specify the parameter list under certain circumstances such as if this is an overloaded function. Oracle requires that the parameter list always be omitted.

Usage of IF EXISTS, CASCADE, or RESTRICT is not compatible with Oracle databases.

The following example describes how to drop the previously created function:

```
DROP FUNCTION simple_function;
```

## 7.3.10 Procedure overview

## 7.3.11 Function overview

## 7.3.12 Compilation errors in procedures and functions

When the PolarDB database compatible with Oracle parsers compile a procedure or function, they confirm that both the CREATE statement and the program body (the portion of the program that follows the AS keyword) conform to the grammar rules for SPL and SQL constructs. By default, the server terminates the compilation process if a parser detects an error. Note that the parsers detect syntax errors in expressions, not semantic errors (that is an expression referencing a non-existent column, table, or function, or an incorrect type).

spl.max_error_count instructs the server to stop parsing if it encounters the specified number of errors in SPL code, or when it encounters an error in SQL code. The default value of the spl.max_error_count parameter is 10. The maximum value is 1000. Setting the value of spl.max_error_count to 1 instructs the server to stop parsing when it encounters the first error in either SPL or SQL code.

You can use the SET statement to specify a value for spl.max_error_count for your current session. The syntax is as follows:

```
SET spl.max_error_count = number_of_errors
```

number_of_errors specifies the number of SPL errors that may occur before the server stops the compilation process. Example:

```
SET spl.max_error_count = 6
```

In the example, codes instruct the server to continue passing the first five SPL errors it encounters. When the server encounters the sixth error it will stop validating, and print six detailed error messages and one error summary.

To save time when you develop new code, or when you import existing code from another source, you can set the spl.max_error_count configuration parameter to a relatively high number of errors.

Note that if you configure the server to continue parsing and ignoring errors in the SPL code in a program body, and the parser encounters an error in a segment of SQL code, there may be errors in any SPL or SQL code that follows the erroneous SQL code. For example, the following content describes a function that results in two errors:

```
CREATE FUNCTION computeBonus(baseSalary number) RETURN number AS
BEGIN

    bonus := baseSalary * 1.10;
    total := bonus + 100;

    RETURN bonus;
```

```
END;

ERROR:  "bonus" is not a known variable
LINE 4:     bonus := baseSalary * 1.10;
         ^
ERROR:  "total" is not a known variable
LINE 5:     total := bonus + 100;
         ^
ERROR:  compilation of SPL function/procedure "computebonus" failed due to 2 errors
```

The following example adds a SELECT statement to the preceding example. The error in the SELECT statement masks other errors that follow:

```
CREATE FUNCTION computeBonus(employeeName number) RETURN number AS
BEGIN
    SELECT salary INTO baseSalary FROM emp
      WHERE ename = employeeName;

    bonus := baseSalary * 1.10;
    total := bonus + 100;

    RETURN bonus;

END;

ERROR:  "basesalary" is not a known variable
LINE 3:     SELECT salary INTO baseSalary FROM emp WHERE ename = emp...
```

# 7.4 Procedure and function parameters

## 7.4.1 Overview

An important aspect of using procedures and functions is the capability to pass data from the calling program to the procedure or function and to receive data back from the procedure or function. This is completed with parameters.

Parameters are declared in the procedure or function definition, enclosed within parentheses () following the procedure or function name. Parameters declared in the procedure or function definition are formal parameters. When the procedure or function is invoked, the calling program provides the actual data that is to be used in the processing of the called program as well as the variables that are to receive the results of the processing of the called program. The data and variables provided by the calling program when the procedure or function is called are actual parameters.

The following content shows the general format of a formal parameter declaration:

```
(name [ IN | OUT | IN OUT ] data_type [ DEFAULT value ])
```

name is an identifier assigned to the formal parameter. If specified, IN defines the parameter for receiving input data into the procedure or function. An IN parameter can also be initialized to a default value. If specified, OUT defines the parameter for returning data from the procedure or function. If specified, IN OUT allows the parameter to be used for both input and output. If all of IN, OUT, and IN OUT are omitted, the parameter acts as if it were defined as IN by default. Whether a parameter is IN, OUT, or IN OUT, it is called parameter mode. data_type defines the data type of the parameter. value is a default value assigned to an IN parameter in the called program when an actual parameter is not specified in the call.

The following example describes a procedure that takes parameters:

```
CREATE OR REPLACE PROCEDURE emp_query (
    p_deptno        IN      NUMBER,
    p_empno         IN OUT NUMBER,
    p_ename         IN OUT VARCHAR2,
    p_job           OUT     VARCHAR2,
    p_hiredate      OUT     DATE,
    p_sal           OUT     NUMBER
)
IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
        INTO p_empno, p_ename, p_job, p_hiredate, p_sal
        FROM emp
        WHERE deptno = p_deptno
          AND (empno = p_empno
           OR  ename = UPPER(p_ename));
END;
```

In this example, p_deptno is an IN formal parameter, p_empno and p_ename are IN OUT formal parameters, and p_job, p_hiredate, and p_sal are OUT formal parameters.

**Note:**

In the previous example, no maximum length was specified on the VARCHAR2 parameters and no precision and scale were specified on the NUMBER parameters. It is invalid to specify a length, precision, scale, or other constraints on parameter declarations. These constraints are automatically inherited from the actual parameters that are used when the procedure or function is called.

The emp_query procedure can be called by another program to pass the actual parameters to the program. The following example describes another SPL program that calls emp_query:

```
DECLARE
    v_deptno        NUMBER(2);
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER;
BEGIN
    v_deptno := 30;
    v_empno  := 7900;
    v_ename  := '';
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
END;
```

In this example, v_deptno, v_empno, v_ename, v_job, v_hiredate, and v_sal are actual parameters.

The following output is generated:

```
Department : 30
Employee No: 7900
Name       : JAMES
Job        : CLERK
Hire Date  : 03-DEC-81
Salary     : 950
```

## 7.4.2 Positional and named parameter notation

You can use either positional or named parameter notation when parameters are passed to a function or procedure. If you specify parameters by using positional notation, you must list the parameters in the order that they are declared. If you specify parameters with named notation, the order of the parameters is not significant.

To specify parameters using named notation, list the name of each parameter followed by an arrow (=>) and the parameter value. Named notation is more verbose, but makes your code easier to read and maintain.

A simple example that demonstrates using positional and named parameter notation is as follows:

```
CREATE OR REPLACE PROCEDURE emp_info (
    p_deptno        IN     NUMBER,
    p_empno         IN OUT NUMBER,
    p_ename         IN OUT VARCHAR2,
)
IS
BEGIN
    dbms_output.put_line('Department Number =' || p_deptno);
    dbms_output.put_line('Employee Number =' || p_empno);
    dbms_output.put_line('Employee Name =' || p_ename;
END;
```

To call the procedure using positional notation, pass the following parameters:

```
emp_info(30, 7455, 'Clark');
```

To call the procedure using named notation, pass the following parameters:

```
emp_info(p_ename =>'Clark', p_empno=>7455, p_deptno=>30);
```

Using named notation can alleviate the need to re-arrange a parameter list of a procedure if the parameter list changes, if the parameters are reordered, or if a new optional parameter is added.

In a case where you have a default value for an argument and the argument is not a trailing argument, you must use named notation to call the procedure or function. The following case demonstrates a procedure with two leading default arguments.

```
CREATE OR REPLACE PROCEDURE check_balance (
    p_customerID  IN NUMBER DEFAULT NULL,
    p_balance     IN NUMBER DEFAULT NULL,
    p_amount      IN NUMBER
)
IS
DECLARE
    balance NUMBER;
BEGIN
   IF (p_balance IS NULL AND p_customerID IS NULL) THEN
      RAISE_APPLICATION_ERROR
          (-20010, 'Must provide balance or customer');
   ELSEIF (p_balance IS NOT NULL AND p_customerID IS NOT NULL) THEN
      RAISE_APPLICATION_ERROR
          (-20020,'Must provide balance or customer, not both');
   ELSEIF (p_balance IS NULL) THEN
      balance := getCustomerBalance(p_customerID);
   ELSE
      balance := p_balance;
   END IF;

   IF (amount > balance) THEN
      RAISE_APPLICATION_ERROR
        (-20030, 'Balance insufficient');
   END IF;
```

```
END;
```

You can only omit non-trailing argument values (when you call this procedure) by using named notation. When using positional notation, only trailing arguments are allowed to default. You can call this procedure with the following arguments:

```
check_balance(p_customerID => 10, p_amount = 500.00)

check_balance(p_balance => 1000.00, p_amount = 500.00)
```

You can use a combination of positional and named notation (mixed notation) to specify parameters. A simple example that demonstrates how to use mixed parameter notation is as follows:

```
CREATE OR REPLACE PROCEDURE emp_info (
    p_deptno       IN     NUMBER,
    p_empno        IN OUT NUMBER,
    p_ename        IN OUT VARCHAR2,
)
IS
BEGIN
    dbms_output.put_line('Department Number =' || p_deptno);
    dbms_output.put_line('Employee Number =' || p_empno);
    dbms_output.put_line('Employee Name =' || p_ename;
END;
```

You can call the procedure by using mixed notation:

```
emp_info(30, p_ename =>'Clark', p_empno=>7455);
```

If you do use mixed notation, remember that named arguments cannot precede positional arguments.

## 7.4.3 Parameter modes

As previously discussed, a parameter has one of three possible modes - IN, OUT, or IN OUT. The following characteristics of a formal parameter are dependent upon its mode:

- Its initial value when the procedure or function is called.
- Whether the called procedure or function can modify the formal parameter.
- How the actual parameter value is passed from the calling program to the called program.
- What happens to the formal parameter value when an unhandled exception occurs in the called program.

The following table summarizes the behavior of parameters based on their mode.

| Mode property | IN | IN OUT | OUT |
|---|---|---|---|
| Formal parameter initialized to: | Actual parameter value | Actual parameter value | Actual parameter value |
| Formal parameter modifiable by the called program? | No | Yes | Yes |
| Actual parameter contains: (after normal called program termination ) | Original actual parameter value prior to the call | Last value of the formal parameter | Last value of the formal parameter |
| Actual parameter contains: (after a handled exception in the called program) | Original actual parameter value prior to the call | Last value of the formal parameter | Last value of the formal parameter |
| Actual parameter contains: (after an unhandled exception in the called program) | Original actual parameter value prior to the call | Original actual parameter value prior to the call | Original actual parameter value prior to the call |

As shown by the table, an IN formal parameter is initialized to the actual parameter with which it is called unless it was explicitly initialized with a default value. The IN parameter may be referenced within the called program. However, the called program may not assign a new value to the IN parameter. After control returns to the calling program, the actual parameter always contains the same value as it was set to prior to the call.

The OUT formal parameter is initialized to the actual parameter with which it is called. The called program may reference and assign new values to the formal parameter. If the called program terminates without an exception, the actual parameter takes on the value last set in the formal parameter. If a handled exception occurs, the value of the actual parameter takes on the last value assigned to the formal parameter. If an unhandled exception occurs , the value of the actual parameter remains as it was prior to the call.

Like an IN parameter, an IN OUT formal parameter is initialized to the actual parameter with which it is called. Like an OUT parameter, an IN OUT formal parameter is modifiable by the called program and the last value in the formal parameter is passed to the actual parameter of the calling program if the called program terminates without an exception . If a handled exception occurs, the value of the actual parameter takes on the last value assigned to the formal parameter. If an unhandled exception occurs, the value of the actual parameter remains as it was prior to the call.

## 7.4.4 Use default values in parameters

You can set a default value of a formal parameter by including the DEFAULT clause or using the assignment operator (:=) in the CREATE PROCEDURE or CREATE FUNCTION statement.

The general form of a formal parameter declaration is as follows:

```
(name [ IN|OUT|IN OUT ] data_type [{DEFAULT | := } expr ])
```

- name is an identifier assigned to the parameter.
- IN|OUT|IN OUT specifies the parameter mode.
- data_type is the data type assigned to the variable.
- expr is the default value assigned to the parameter. If you do not include a DEFAULT clause, the caller must provide a value for the parameter.

The default value is evaluated every time the function or procedure is invoked. For example , assigning SYSDATE to a parameter of type DATE causes the parameter to have the time of the current invocation, not the time when the procedure or function was created.

The following simple procedure demonstrates how to use the assignment operator to set a default value of SYSDATE into the hiredate parameter:

```
CREATE OR REPLACE PROCEDURE hire_emp (
    p_empno         NUMBER,
    p_ename         VARCHAR2,
    p_hiredate      DATE := SYSDATE
)
IS
BEGIN
    INSERT INTO emp(empno, ename, hiredate)
                VALUES(p_empno, p_ename, p_hiredate);

    DBMS_OUTPUT.PUT_LINE('Hired!') ;
END hire_emp;
```

If the parameter declaration includes a default value, you can omit the parameter from the actual parameter list when you call the procedure. Calls to the sample procedure (hire_emp

) must include two arguments: the employee number (p_empno) and employee name ( p_empno). The third parameter (p_hiredate) defaults to the value of SYSDATE:

```
hire_emp (7575, Clark)
```

If you do include a value for the actual parameter when you call the procedure, that value takes precedence over the default value:

```
hire_emp (7575, Clark, 15-FEB-2010)
```

Adds a new employee with a hiredate of February 15, 2010, regardless of the current value of SYSDATE.

You can write the same procedure by substituting the DEFAULT keyword for the assignment operator:

```
CREATE OR REPLACE PROCEDURE hire_emp (
    p_empno         NUMBER,
    p_ename         VARCHAR2,
    p_hiredate      DATE DEFAULT SYSDATE
)
IS
BEGIN
    INSERT INTO emp(empno, ename, hiredate)
                VALUES(p_empno, p_ename, p_hiredate);

    DBMS_OUTPUT.PUT_LINE('Hired!') ;
END hire_emp;
```

## 7.5 Subprograms - subprocedures and subfunctions

### 7.5.1 Overview

The capability and functionality of SPL procedure and function programs can be used in an advantageous manner to build well-structured and maintainable programs by organizing the SPL code into subprocedures and subfunctions.

The same SPL code can be invoked multiple times from different locations within a relatively large SPL program by declaring subprocedures and subfunctions within the SPL program.

Subprocedures and subfunctions have the following characteristics:

- The syntax, structure, and functionality of subprocedures and subfunctions are practically identical to standalone procedures and functions. The major difference is the use of the keyword PROCEDURE or FUNCTION instead of CREATE PROCEDURE or CREATE FUNCTION to declare the subprogram.

- Subprocedures and subfunctions provide isolation for the identifiers (that is, variables, cursors, types, and other subprograms) declared within itself. That is, these identifiers cannot be accessed or altered from the upper parent level SPL programs or subprograms outside of the subprocedure or subfunction. This ensures that the subprocedure and subfunction results are reliable and predictable.
- The declaration section of subprocedures and subfunctions can include its own subprocedures and subfunctions. Thus, a multi-level hierarchy of subprograms can exist in the standalone program. Within the hierarchy, a subprogram can access the identifiers of upper level parent subprograms and also invoke upper level parent subprograms. However, the same access to identifiers and invocation cannot be done for lower level child subprograms in the hierarchy.

Subprocedures and subfunctions can be declared and invoked from within any of the following types of SPL programs:

- Standalone procedures and functions
- Anonymous blocks
- Triggers
- Packages
- Procedure and function methods of an object type body
- Subprocedures and subfunctions declared within any of the preceding programs

The rules regarding subprocedure and subfunction structure and access are discussed in more detail in the next topics.

## 7.5.2 Create a subprocedure

The PROCEDURE clause specified in the declaration section defines and names a subprocedure local to that block.

The term **block** refers to the SPL block structure consisting of an optional declaration section, a mandatory executable section, and an optional exception section. Blocks are the structures for standalone procedures and functions, anonymous blocks, subprograms, triggers, packages, and object type methods.

The phrase **the identifier is local to the block** means that the identifier (that is, a variable, cursor, type, or subprogram) is declared within the declaration section of that block and is therefore accessible by the SPL code within the executable section and optional exception section of that block.

Subprocedures can only be declared after all other variable, cursor, and type declarations included in the declaration section. ( That is, subprograms must be the last set of declaratio ns.)

```
PROCEDURE name [ (parameters) ]{ IS | AS }
  [ PRAGMA AUTONOMOUS_TRANSACTION; ]
  [ declarations ]
 BEGIN
  statements
 END [ name ];
```

**Arguments**

| Argument | Description |
|---|---|
| name | name is the identifier of the subprocedure. |
| parameters | parameters is a list of formal parameters. |
| PRAGMA AUTONOMOUS _TRANSACTION | PRAGMA AUTONOMOUS_TRANSACTION is the directive that sets the subprocedure as an autonomous transaction. |
| declarations | declarations are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations. |
| statements | statements are SPL program statements (the BEGIN - END block may contain an EXCEPTION section). |

**Examples**

The following example is a subprocedure within an anonymous block:

```
DECLARE
  PROCEDURE list_emp
  IS
    v_empno     NUMBER(4);
    v_ename     VARCHAR2(10);
    CURSOR emp_cur IS
      SELECT empno, ename FROM emp ORDER BY empno;
  BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE('Subprocedure list_emp:');
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
      FETCH emp_cur INTO v_empno, v_ename;
      EXIT WHEN emp_cur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_cur;
    END;
BEGIN
    list_emp;
END;
```

The following output is generated by invoking this anonymous block:

```
Subprocedure list_emp:
EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
```

The following example is a subprocedure within a trigger:

```
CREATE OR REPLACE TRIGGER dept_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON dept
DECLARE
    v_action        VARCHAR2(24);
    PROCEDURE display_action (
        p_action IN VARCHAR2
    )
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('User ' || USER || ' ' || p_action ||
            ' dept on ' || TO_CHAR(SYSDATE,'YYYY-MM-DD'));
    END display_action;
BEGIN
    IF INSERTING THEN
        v_action := 'added';
    ELSIF UPDATING THEN
        v_action := 'updated';
    ELSIF DELETING THEN
        v_action := 'deleted';
    END IF;
    display_action(v_action);
END;
```

The following output is generated by invoking this trigger:

```
INSERT INTO dept VALUES (50,'HR','DENVER');
```

```
User enterprisedb added dept on 2016-07-26
```

## 7.5.3 Create a subfunction

The FUNCTION clause specified in the declaration topic defines and names a subfunction local to that block.

The term **block** refers to the SPL block structure consisting of an optional declaration topic, a mandatory executable section, and an optional exception section. Blocks are the structures for standalone procedures and functions, anonymous blocks, subprograms, triggers, packages, and object type methods.

The phrase **the identifier is local to the block** means that the identifier (that is, a variable, cursor, type, or subprogram) is declared within the declaration section of that block and is therefore accessible by the SPL code within the executable section and optional exception section of that block.

Subprocedures can only be declared after all other variable, cursor, and type declarations included in the declaration section. ( That is, subprograms must be the last set of declarations.)

```
PROCEDURE name [ (parameters) ]{ IS | AS }
  [ PRAGMA AUTONOMOUS_TRANSACTION; ]
  [ declarations ]
 BEGIN
  statements
 END [ name ];
```

**Table 7-2: Arguments**

| Argument | Description |
|---|---|
| name | name is the identifier of the subprocedure. |
| parameters | parameters is a list of formal parameters. |
| data_type | data_type is the data type of the value returned by the RETURN statement of the function. |
| PRAGMA AUTONOMOUS_TRANSACTION | PRAGMA AUTONOMOUS_TRANSACTION is the directive that sets the subfunction as an autonomous transaction. |
| declarations | declarations are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations. |

| Argument | Description |
|---|---|
| statements | statements are SPL program statements (the BEGIN - END block may contain an EXCEPTION section). |

**Examples**

The following example shows the use of a recursive subfunction:

```
DECLARE
    FUNCTION factorial (
        n          BINARY_INTEGER
    ) RETURN BINARY_INTEGER
    IS
    BEGIN
        IF n = 1 THEN
            RETURN n;
        ELSE
            RETURN n * factorial(n-1);
        END IF;
    END factorial;
BEGIN
    FOR i IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE(i || '! = ' || factorial(i));
    END LOOP;
END;
```

The output from the example is as follows:

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

## 7.5.4 Block relationships

This topic describes the terminology of the relationship between blocks that can be declared in an SPL program. The ability to invoke subprograms and access identifiers declared within a block depends upon this relationship.

The following content describes the basic terms:

- A block is the basic SPL structure consisting of an optional declaration section, a mandatory executable section, and an optional exception section. Blocks implement standalone procedure and function programs, anonymous blocks, triggers, packages, subprocedures, and subfunctions.
- An identifier (variable, cursor, type, or subprogram) local to a block means that it is declared within the declaration topic of the given block. Such local identifiers are accessible from the executable section and optional exception section of the block.

- The parent block contains the declaration of another block (the child block).
- Descendent blocks are the set of blocks forming the child relationship starting from a given parent block.
- Ancestor blocks are the set of blocks forming the parental relationship starting from a given child block.
- The set of descendent (or ancestor) blocks form a hierarchy.
- The level is an ordinal number of a given block from the highest ancestor block. For example, given a standalone procedure, the subprograms declared within the declaration topic of this procedure are all at the same level. For example, call this procedure at level 1. Additional subprograms within the declaration topic of the subprograms declared in the standalone procedure are at the next level which is level 2.
- The sibling blocks are the set of blocks that have the same parent block (that is, they are all locally declared in the same block). Sibling blocks are also always at the same level relative to each other.

The following schematic of a set of procedure declaration topics provides an example of a set of blocks and their relationships to their surrounding blocks.

The two vertical lines on the left-hand side of the blocks indicate that two pairs of sibling blocks exist. block_1a and block_1b are one pair, and block_2a and block_2b are the second pair.

The relationship of each block with its ancestors is shown on the right-hand side of the blocks. Three hierarchical paths are formed when progressing up the hierarchy from the lowest level child blocks. The first consists of block_0, block_1a, block_2a, and block_3. The second is block_0, block_1a, and block_2b. The third is block_0, block_1b, and block_2b.

```
CREATE PROCEDURE block_0
IS
    .
  +---- PROCEDURE block_1a    ------- Local to block_0
  |    IS
  |      .                |
  |      .                |
  |      .                |
  |    +-- PROCEDURE block_2a   ---- Local to block_1a and descendant
  |    |   IS                  of block_0
  |    |      .               |
  |    |      .               |
  |    |      .               |
  |    |     PROCEDURE block_3   -- Local to block_2a and descendant
  |    |     IS                of block_1a, and block_0
  | Siblings       .           |
  |    |         .            |
  |    |         .            |
  |    |      END block_3;        |
    |    |  END block_2a;              |
    |    +-- PROCEDURE block_2b   ---- Local to block_1a and descendant
    |    |  IS                      of block_0
 Siblings |      ,                  |
    |    |      .                  |
    |    |      .                  |
    |    +-- END block_2b;            |
    |                              |
    |    END block_1a;        ---------+
    +---- PROCEDURE block_1b;   ------- Local to block_0
    |    IS
    |       .                      |
    |       .                      |
    |       .                      |
    |       PROCEDURE block_2b   ---- Local to block_1b and descendant
    |       IS                      of block_0
    |          .                   |
    |          .                   |
    |          .                   |
    |       END block_2b;            |
    |                              |
    +---- END block_1b;        ---------+
BEGIN
     .
     .
     .
END block_0;
```

## 7.5.5 Invoke subprograms

You can specify the name and any actual parameters to invoke a subprogram in the same way you invoke a standalone procedure or function.

The subprogram can be invoked with none, one, or more qualifiers, which are the names of the parent subprograms or labeled anonymous blocks forming the ancestor hierarchy from where the subprogram has been declared.

The following example describes the invocation that is specified as a dot-separated list of qualifiers ending with the subprogram name and any arguments of the subprogram:

```
[[qualifier_1.][...]qualifier_n.]subprog [(arguments)]
```

If specified, qualifier_n is the subprogram in which subprog has been declared in the declaration section of the subprogram. The preceding list of qualifiers must reside in a continuous path up the hierarchy from qualifier_n to qualifier_1. qualifier_1 may be any ancestor subprogram in the path as well as any of the following options:

- Standalone procedure name containing the subprogram.
- Standalone function name containing subprogram.
- Package name containing the subprogram.
- Object type name containing the subprogram within an object type method.

- An anonymous block label included prior to the DECLARE keyword if a declaration section exists, or prior to the BEGIN keyword if there is no declaration section.

**Note:**
qualifier_1 cannot be a schema name. Otherwise, an error is thrown upon invocation of the subprogram. This PolarDB database compatible with Oracle restriction is not compatible with Oracle databases, which allow use of the schema name as a qualifier.

arguments is the list of actual parameters to be passed to the subprocedure or subfunction .

Upon invocation, the search for the subprogram occurs as follows:

- The invoked subprogram name of its type (that is, subprocedure or subfunction) along with any qualifiers in the specified order, (referred to as the invocation list) is used to find a matching set of blocks residing in the same hierarchical order. The search begins in the block hierarchy where the lowest level is the block from where the subprogram is invoked. The declaration of the subprogram must be in the SPL code prior to the code line where it is invoked when the code is observed from top to bottom.
- If the invocation list does not match the hierarchy of blocks starting from the block where the subprogram is invoked, a comparison is made by matching the invocation list starting with the parent of the previous starting block. In other words, the comparison progresses up the hierarchy.
- If there are sibling blocks of the ancestors, the invocation list comparison also includes the hierarchy of the sibling blocks, but always comparing in an upward level, never comparing the descendants of the sibling blocks.
- This comparison process continues up the hierarchies until the first complete match is found in which case the located subprogram is invoked. Note that the formal parameter list of the matched subprogram must comply with the actual parameter list specified for the invoked subprogram. Otherwise, an error occurs upon invocation of the subprogram.
- If no match is found after searching up to the standalone program, an error is thrown upon invocation of the subprogram.

**Note:**
The PolarDB database compatible with Oracle search algorithm for subprogram invocation is not compatible with Oracle databases. For Oracle, the search looks for the first match of the first qualifier (that is qualifier_1). When such a match is found, all remaining qualifiers,

the subprogram name, the subprogram type, and arguments of the invocation must match the hierarchy content where the matching first qualifier is found. Otherwise, an error is thrown. For PolarDB databases compatible with Oracle, a match is not found unless all qualifiers, the subprogram name, and the subprogram type of the invocation match the hierarchy content. If such an exact match is not found at first, the PolarDB database compatible with Oracle continues the search progressing up the hierarchy.

The location of subprograms relative to the block from where the invocation is made can be accessed as follows:

- Subprograms declared in the local block can be invoked from the executable section or the exception section of the same block.
- Subprograms declared in the parent or other ancestor blocks can be invoked from the child block of the parent or other ancestors.
- Subprograms declared in sibling blocks can be called from a sibling block or from any descendent block of the sibling.

However, the following location of subprograms cannot be accessed relative to the block from where the invocation is made:

- Subprograms declared in blocks that are descendants of the block from where the invocation is attempted.
- Subprograms declared in blocks that are descendants of the sibling block from where the invocation is attempted.

The following examples illustrate the various conditions previously described.

### Invoke locally declared subprograms

The following example contains a single hierarchy of blocks contained within the level_0 standalone procedure. Within the executable section of the level_1a procedure, the means of invoking the level_2a local procedure are shown, both with and without qualifiers.

Note that access to the descendant of the level_2a local procedure, which is the level_3a procedure, is not permitted, with or without qualifiers. The following example comments out these calls:

```
CREATE OR REPLACE PROCEDURE level_0
IS
  PROCEDURE level_1a
  IS
    PROCEDURE level_2a
    IS
      PROCEDURE level_3a
```

```
         IS
         BEGIN
            DBMS_OUTPUT.PUT_LINE('........ BLOCK level_3a');
            DBMS_OUTPUT.PUT_LINE('........ END BLOCK level_3a');
         END level_3a;
      BEGIN
         DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2a');
         DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2a');
      END level_2a;
   BEGIN
      DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
      level_2a;                   -- Local block called
      level_1a.level_2a;          -- Qualified local block called
      level_0.level_1a.level_2a;  -- Double qualified local block called
--    level_3a;                   -- Error - Descendant of local block
--    level_2a.level_3a;          -- Error - Descendant of local block
      DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
   END level_1a;
BEGIN
   DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
   level_1a;
   DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;
```

When the standalone procedure is invoked, the following output is generated, which indicates that the level_2a procedure is invoked from the calls in the executable section of the level_1a procedure.

```
BEGIN
   level_0;
END;

BLOCK level_0
.. BLOCK level_1a
...... BLOCK level_2a
...... END BLOCK level_2a
...... BLOCK level_2a
...... END BLOCK level_2a
...... BLOCK level_2a
...... END BLOCK level_2a
.. END BLOCK level_1a
END BLOCK level_0
```

If you were to attempt to run the level_0 procedure with any of the calls to the descendent block uncommented, an error occurs.

#### Invoke subprograms declared in ancestor blocks

The following example shows how subprograms can be invoked that are declared in parent and other ancestor blocks relative to the block where the invocation is made.

In this example, the executable section of the level_3a procedure invokes the level_2a procedure, which is the parent block of the level_3a procedure. Note that v_cnt is used to avoid an infinite loop.

```
CREATE OR REPLACE PROCEDURE level_0
```

```
IS
   v_cnt         NUMBER(2) := 0;
   PROCEDURE level_1a
   IS
      PROCEDURE level_2a
      IS
         PROCEDURE level_3a
         IS
         BEGIN
            DBMS_OUTPUT.PUT_LINE('........ BLOCK level_3a');
            v_cnt := v_cnt + 1;
            IF v_cnt < 2 THEN
               level_2a;                -- Parent block called
            END IF;
            DBMS_OUTPUT.PUT_LINE('........ END BLOCK level_3a');
         END level_3a;
      BEGIN
         DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2a');
         level_3a;                      -- Local block called
         DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2a');
      END level_2a;
   BEGIN
      DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
      level_2a;                         -- Local block called
      DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
   END level_1a;
BEGIN
   DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
   level_1a;
   DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;
```

The following output is generated:

```
BEGIN
   level_0;
END;

BLOCK level_0
.. BLOCK level_1a
...... BLOCK level_2a
........ BLOCK level_3a
...... BLOCK level_2a
........ BLOCK level_3a
........ END BLOCK level_3a
...... END BLOCK level_2a
........ END BLOCK level_3a
...... END BLOCK level_2a
.. END BLOCK level_1a
END BLOCK level_0
```

In a similar example, the executable section of the level_3a procedure invokes the level_1a procedure, which is further up the ancestor hierarchy. Note that v_cnt is used to avoid an infinite loop.

```
CREATE OR REPLACE PROCEDURE level_0
IS
   v_cnt         NUMBER(2) := 0;
   PROCEDURE level_1a
   IS
```

```
        PROCEDURE level_2a
        IS
            PROCEDURE level_3a
            IS
            BEGIN
                DBMS_OUTPUT.PUT_LINE('........ BLOCK level_3a');
                v_cnt := v_cnt + 1;
                IF v_cnt < 2 THEN
                    level_1a;                -- Ancestor block called
                END IF;
                DBMS_OUTPUT.PUT_LINE('........ END BLOCK level_3a');
            END level_3a;
        BEGIN
            DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2a');
            level_3a;                        -- Local block called
            DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2a');
        END level_2a;
    BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
        level_2a;                            -- Local block called
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
    END level_1a;
BEGIN
    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
    level_1a;
    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;
```

The following output is generated:

```
BEGIN
    level_0;
END;

BLOCK level_0
.. BLOCK level_1a
...... BLOCK level_2a
........ BLOCK level_3a
.. BLOCK level_1a
...... BLOCK level_2a
........ BLOCK level_3a
........ END BLOCK level_3a
...... END BLOCK level_2a
.. END BLOCK level_1a
........ END BLOCK level_3a
...... END BLOCK level_2a
.. END BLOCK level_1a
END BLOCK level_0
```

### Invoke subprograms declared in sibling blocks

The following examples show how subprograms can be invoked that are declared in a sibling block relative to the local, parent, or other ancestor blocks from where the invocation of the subprogram is made.

In this example, the executable section of the level_1b procedure invokes the level_1a procedure, which is the sibling block of the level_1b procedure. Both are local to the level_0 standalone procedure.

Note that invocation of level_2a or equivalently level_1a.level_2a from within the level_1b procedure is commented out because this call would result in an error. Invoking a descendent subprogram (level_2a) of sibling block (level_1a) is not permitted.

```
CREATE OR REPLACE PROCEDURE level_0
IS
   v_cnt     NUMBER(2) := 0;
   PROCEDURE level_1a
   IS
      PROCEDURE level_2a
      IS
      BEGIN
         DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2a');
         DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2a');
      END level_2a;
   BEGIN
      DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
      DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
   END level_1a;
   PROCEDURE level_1b
   IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
      level_1a;                          -- Sibling block called
--     level_2a;                          -- Error - Descendant of sibling block
--     level_1a.level_2a;                   -- Error - Descendant of sibling block
      DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
   END level_1b;
BEGIN
   DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
   level_1b;
   DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;
```

The following output is generated:

```
BEGIN
   level_0;
END;

BLOCK level_0
.. BLOCK level_1b
.. BLOCK level_1a
.. END BLOCK level_1a
.. END BLOCK level_1b
END BLOCK level_0
```

In the following example, the level_1a procedure is invoked. This procedure is the sibling of the level_1b procedure, which is an ancestor of the level_3b procedure.

```
CREATE OR REPLACE PROCEDURE level_0
IS
   PROCEDURE level_1a
   IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
      DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
   END level_1a;
   PROCEDURE level_1b
   IS
      PROCEDURE level_2b
      IS
         PROCEDURE level_3b
         IS
         BEGIN
            DBMS_OUTPUT.PUT_LINE('........ BLOCK level_3b');
            level_1a;                    -- Ancestor's sibling block called
            level_0.level_1a;              -- Qualified ancestor's sibling block
            DBMS_OUTPUT.PUT_LINE('........ END BLOCK level_3b');
         END level_3b;
      BEGIN
         DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2b');
         level_3b;                          -- Local block called
         DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2b');
      END level_2b;
   BEGIN
      DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
      level_2b;                             -- Local block called
      DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
   END level_1b;
BEGIN
   DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
   level_1b;
   DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;
```

The following output is generated:

```
BEGIN
   level_0;
END;

BLOCK level_0
.. BLOCK level_1b
...... BLOCK level_2b
........ BLOCK level_3b
.. BLOCK level_1a
.. END BLOCK level_1a
.. BLOCK level_1a
.. END BLOCK level_1a
........ END BLOCK level_3b
...... END BLOCK level_2b
.. END BLOCK level_1b
END BLOCK level_0
```

## 7.5.6 Use forward declarations

When a subprogram is to be invoked, it must have been declared somewhere in the hierarchy of blocks within the standalone program, but prior to where it is invoked. In other words, when scanning the SPL code from beginning to end, the subprogram declaration must be found before its invocation.

However, there is a method of constructing the SPL code so that the full declaration of the subprogram (that is, the optional declaration section, mandatory executable section, and optional exception section of the subprogram) appears in the SPL code after the point in the code where it is invoked.

This is accomplished by inserting a forward declaration in the SPL code prior to its invocation. The forward declaration is the specification of a subprocedure or subfunction name, formal parameters, and return type if it is a subfunction.

The full subprogram specification consisting of the optional declaration section, the executable section, and the optional exception section must be specified in the same declaration section as the forward declaration, but may appear following other subprogram declarations that invoke this subprogram with the forward declaration.

The following example shows the typical usage of a forward declaration, which is when two subprograms invoke each other:

```
DECLARE
    FUNCTION add_one (
        p_add       IN NUMBER
    ) RETURN NUMBER;
    FUNCTION test_max (
        p_test      IN NUMBER)
    RETURN NUMBER
    IS
    BEGIN
        IF p_test < 5 THEN
            RETURN add_one(p_test);
        END IF;
        DBMS_OUTPUT.PUT('Final value is ');
        RETURN p_test;
    END;
    FUNCTION add_one (
        p_add       IN NUMBER)
    RETURN NUMBER
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Increase by 1');
        RETURN test_max(p_add + 1);
    END;
BEGIN
    DBMS_OUTPUT.PUT_LINE(test_max(3));
END;
```

Subfunction test_max invokes subfunction add_one, which also invokes subfunction test_max. Therefore, a forward declaration is required for one of the subprograms, which is implemented for add_one at the beginning of the anonymous block declaration section.

The following output is generated by the anonymous block:

```
Increase by 1
Increase by 1
```

```
Final value is 5
```

## 7.5.7 Overload subprograms

Generally, subprograms of the same type (subprocedure or subfunction) with the same name and the same formal parameter specification can appear multiple times within the same standalone program as long as they are not sibling blocks (that is, the subprograms are not declared in the same local block).

Each subprogram can be individually invoked depending on the use of qualifiers and the location where the subprogram invocation is made.

However, if subprograms are of the same subprogram type and name as long as certain aspects of the formal parameters differ, you can declare the subprograms, even as siblings . These characteristics, such as subprogram type, name, and formal parameter specification, are generally known as the signature of a program.

The declaration of multiple subprograms where the signatures are identical except for certain aspects of the formal parameter specification is referred to as subprogram overloading.

Therefore, the determination of which specified overloaded subprogram is to be invoked is determined by a match of the actual parameters specified by the subprogram invocation and the formal parameter lists of the overloaded subprograms.

Any of the following differences permit overloaded subprograms:

- The number of formal parameters is different.
- At least one pair of data types of the corresponding formal parameters (that is, compared according to the same order of appearance in the formal parameter list) are different, but are not aliases. Data type aliases are discussed later in this topic.

Note that the following differences alone do not permit overloaded subprograms:

- Different formal parameter names
- Different parameter modes (IN, IN OUT, OUT) for the corresponding formal parameters
- For subfunctions, different data types in the RETURN clause

Therefore, one of the differences that allows overloaded subprograms are different data types.

However, certain data types have alternative names referred to as aliases, which can be used for the table definition.

For example, there are fixed length character data types that can be specified as CHAR or CHARACTER. There are variable length character data types that can be specified as CHAR VARYING, CHARACTER VARYING, VARCHAR, or VARCHAR2. For integers, there are BINARY_INTEGER, PLS_INTEGER, and INTEGER data types. For numbers, there are NUMBER, NUMERIC, DEC, and DECIMAL data types.

Therefore, in an attempt to create overloaded subprograms, the formal parameter data types are not considered different if the specified data types are aliases of each other.

It can be determined if certain data types are aliases of other types by displaying the table definition that contains the data types in question.

For example, the following table definition contains some data types and their aliases:

```
CREATE TABLE data_type_aliases (
    dt_BLOB             BLOB,
    dt_LONG_RAW         LONG RAW,
    dt_RAW              RAW(4),
    dt_BYTEA            BYTEA,
    dt_INTEGER          INTEGER,
    dt_BINARY_INTEGER   BINARY_INTEGER,
    dt_PLS_INTEGER      PLS_INTEGER,
    dt_REAL             REAL,
    dt_DOUBLE_PRECISION DOUBLE PRECISION,
    dt_FLOAT            FLOAT,
    dt_NUMBER           NUMBER,
    dt_DECIMAL          DECIMAL,
    dt_NUMERIC          NUMERIC,
    dt_CHAR             CHAR,
    dt_CHARACTER        CHARACTER,
    dt_VARCHAR2         VARCHAR2(4),
    dt_CHAR_VARYING     CHAR VARYING(4),
    dt_VARCHAR          VARCHAR(4)
);
```

Using the PSQL \d statement to display the table definition, the Type column displays the data type internally assigned to each column based on its data type in the table definition:

```
\d data_type_aliases
       Column        |         Type         | Modifiers
---------------------+----------------------+-----------
 dt_blob             | bytea                |
 dt_long_raw         | bytea                |
 dt_raw              | bytea(4)             |
 dt_bytea            | bytea                |
 dt_integer          | integer              |
 dt_binary_integer   | integer              |
 dt_pls_integer      | integer              |
 dt_real             | real                 |
 dt_double_precision | double precision     |
 dt_float            | double precision     |
 dt_number           | numeric              |
 dt_decimal          | numeric              |
 dt_numeric          | numeric              |
 dt_char             | character(1)         |
 dt_character      | character(1)       |
 dt_varchar2       | character varying(4) |
 dt_char_varying    | character varying(4) |
 dt_varchar        | character varying(4) |
```

In the example, the base set of data types are bytea, integer, real, double precision, numeric, character, and character varying.

When attempting to declare overloaded subprograms, a pair of formal parameter data types that are aliases is insufficient to allow subprogram overloading. Therefore , parameters with data types INTEGER and PLS_INTEGER cannot overload a pair of subprograms, but data types INTEGER and REAL, INTEGER and FLOAT, or INTEGER and NUMBER can overload the subprograms.

**Note:**

The overloading rules based on formal parameter data types are not compatible with Oracle databases. Generally, the PolarDB database compatible with Oracle rules are more flexible. However, some combinations that are allowed in PolarDB databases compatible with Oracle would result in an error when attempting to create the procedure or function in Oracle databases.

For certain pairs of data types used for overloading, casting of the arguments specified by the subprogram invocation may be required to avoid an error encountered during runtime of the subprogram. Invocation of a subprogram must include the actual parameter list that can specifically identify the data types. Certain pairs of overloaded data types may require the CAST function to explicitly identify data types. For example, pairs of overloaded data types that may require casting during the invocation are CHAR and VARCHAR2, or NUMBER and REAL.

The following example shows a group of overloaded subfunctions invoked from within an anonymous block. The executable section of the anonymous block contains the use of the CAST function to invoke overloaded functions that have certain data types.

```
DECLARE
    FUNCTION add_it (
        p_add_1     IN BINARY_INTEGER,
        p_add_2     IN BINARY_INTEGER
    ) RETURN VARCHAR2
    IS
    BEGIN
        RETURN 'add_it BINARY_INTEGER: ' || TO_CHAR(p_add_1 + p_add_2,9999.9999);
    END add_it;
    FUNCTION add_it (
        p_add_1     IN NUMBER,
        p_add_2     IN NUMBER
    ) RETURN VARCHAR2
    IS
    BEGIN
        RETURN 'add_it NUMBER: ' || TO_CHAR(p_add_1 + p_add_2,999.9999);
    END add_it;
    FUNCTION add_it (
        p_add_1     IN REAL,
        p_add_2     IN REAL
    ) RETURN VARCHAR2
    IS
    BEGIN
        RETURN 'add_it REAL: ' || TO_CHAR(p_add_1 + p_add_2,9999.9999);
    END add_it;
    FUNCTION add_it (
        p_add_1     IN DOUBLE PRECISION,
        p_add_2     IN DOUBLE PRECISION
    ) RETURN VARCHAR2
    IS
    BEGIN
        RETURN 'add_it DOUBLE PRECISION: ' || TO_CHAR(p_add_1 + p_add_2,9999.9999);
    END add_it;
BEGIN
    DBMS_OUTPUT.PUT_LINE(add_it (25, 50));
    DBMS_OUTPUT.PUT_LINE(add_it (25.3333, 50.3333));
    DBMS_OUTPUT.PUT_LINE(add_it (TO_NUMBER(25.3333), TO_NUMBER(50.3333)));
    DBMS_OUTPUT.PUT_LINE(add_it (CAST('25.3333' AS REAL), CAST('50.3333' AS REAL)));
    DBMS_OUTPUT.PUT_LINE(add_it (CAST('25.3333' AS DOUBLE PRECISION),
        CAST('50.3333' AS DOUBLE PRECISION)));
END;
```

The following output is displayed from the anonymous block:

```
add_it BINARY_INTEGER:    75.0000
add_it NUMBER:   75.6666
add_it NUMBER:   75.6666
add_it REAL:     75.6666
add_it DOUBLE PRECISION:    75.6666
```

## 7.5.8 Access subprogram variables

Variable declared in blocks such as subprograms or anonymous blocks can be accessed from the executable section or the exception section of other blocks depending on their relative location.

Accessing a variable means being able to reference it within a SQL statement or an SPL statement as is done with any local variable.

**Note:**

If the subprogram signature contains formal parameters, these may be accessed in the same manner as local variables of the subprogram. In this topic, all discussion related to variables of a subprogram also applies to formal parameters of the subprogram.

Access of variables includes those defined as a data type and others such as record types, collection types, and cursors.

The variable may be accessed by up to one qualifier, which is the name of the subprogram or labeled anonymous block in which the variable has been locally declared.

The following content shows the syntax to reference a variable:

```
[qualifier.]variable
```

If specified, qualifier is the subprogram or labeled anonymous block in which variable has been declared in its declaration section (that is, variable is a local variable).

In PolarDB databases compatible with Oracle, there is only one circumstance where two qualifiers are permitted. This scenario is for accessing public variables of packages where the reference can be specified in the following format:

```
schema_name.package_name.public_variable_name
```

The following content summarizes how variables can be accessed:

- Variables can be accessed as long as the block in which the variable has been locally declared is within the ancestor hierarchical path starting from the block containing the reference to the variable. Such variables declared in ancestor blocks are referred to as global variables.
- If a reference to an unqualified variable is made, the first attempt is to locate a local variable of that name. If the specified local variable does not exist, the search for the variable is made in the parent of the current block, and so forth, proceeding up the ancestor hierarchy. If the specified variable is not found, an error occurs upon invocation of the subprogram.
- If a reference to a qualified variable is made, the same search process is performed as described in the previous bullet point, but searching for the first match of the subprogram or labeled anonymous block that contains the local variable. The search proceeds up the ancestor hierarchy until a match is found. If the specified match is not found, an error occurs upon invocation of the subprogram.

The following location of variables cannot be accessed relative to the block from where the reference to the variable is made:

- Variables declared in a descendent block cannot be accessed
- Variables declared in a sibling block, a sibling block of an ancestor block, or any descendants within the sibling block cannot be accessed.

**Note:**

The PolarDB database compatible with Oracle process for accessing variables is not compatible with Oracle databases. For Oracle, any number of qualifiers can be specified and the search is based on the first match of the first qualifier in a similar manner to the Oracle matching algorithm for invoking subprograms.

The following example displays how variables in various blocks are accessed, with and without qualifiers. The lines that are commented out illustrate attempts to access variables that would result in an error.

```
CREATE OR REPLACE PROCEDURE level_0
IS
   v_level_0     VARCHAR2(20) := 'Value from level_0';
   PROCEDURE level_1a
   IS
      v_level_1a  VARCHAR2(20) := 'Value from level_1a';
      PROCEDURE level_2a
      IS
         v_level_2a     VARCHAR2(20) := 'Value from level_2a';
      BEGIN
         DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2a');
         DBMS_OUTPUT.PUT_LINE('........ v_level_2a: ' || v_level_2a);
         DBMS_OUTPUT.PUT_LINE('........ v_level_1a: ' || v_level_1a);
         DBMS_OUTPUT.PUT_LINE('........ level_1a.v_level_1a: ' ||
                                level_1a.v_level_1a);
         DBMS_OUTPUT.PUT_LINE('........ v_level_0: ' || v_level_0);
         DBMS_OUTPUT.PUT_LINE('........ level_0.v_level_0: ' || level_0.v_level_0);
         DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2a');
      END level_2a;
   BEGIN
      DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
      level_2a;
--      DBMS_OUTPUT.PUT_LINE('.... v_level_2a: ' || v_level_2a);
--                      Error - Descendent block ----^
--      DBMS_OUTPUT.PUT_LINE('.... level_2a.v_level_2a: ' || level_2a.v_level_2a);
--                      Error - Descendent block ---------------^
      DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
   END level_1a;
   PROCEDURE level_1b
   IS
      v_level_1b  VARCHAR2(20) := 'Value from level_1b';
   BEGIN
      DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
      DBMS_OUTPUT.PUT_LINE('.... v_level_1b: ' || v_level_1b);
      DBMS_OUTPUT.PUT_LINE('.... v_level_0 : ' || v_level_0);
--      DBMS_OUTPUT.PUT_LINE('.... level_1a.v_level_1a: ' || level_1a.v_level_1a);
--                       Error - Sibling block -----------------^
--      DBMS_OUTPUT.PUT_LINE('.... level_2a.v_level_2a: ' || level_2a.v_level_2a);
--                       Error - Sibling block descendant ------^
      DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
   END level_1b;
BEGIN
   DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
   DBMS_OUTPUT.PUT_LINE('.. v_level_0: ' || v_level_0);
   level_1a;
   level_1b;
   DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
```

```
END level_0;
```

The following output shows the content of each variable when the procedure is invoked:

```
BEGIN
    level_0;
END;

BLOCK level_0
.. v_level_0: Value from level_0
.. BLOCK level_1a
...... BLOCK level_2a
........ v_level_2a: Value from level_2a
........ v_level_1a: Value from level_1a
........ level_1a.v_level_1a: Value from level_1a
........ v_level_0: Value from level_0
........ level_0.v_level_0: Value from level_0
...... END BLOCK level_2a
.. END BLOCK level_1a
.. BLOCK level_1b
.... v_level_1b: Value from level_1b
.... v_level_0 : Value from level_0
.. END BLOCK level_1b
END BLOCK level_0
```

The following example shows similar access attempts when all variables in all blocks have the same name:

```
CREATE OR REPLACE PROCEDURE level_0
IS
    v_common        VARCHAR2(20) := 'Value from level_0';
    PROCEDURE level_1a
    IS
        v_common    VARCHAR2(20) := 'Value from level_1a';
        PROCEDURE level_2a
        IS
            v_common    VARCHAR2(20) := 'Value from level_2a';
        BEGIN
            DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2a');
            DBMS_OUTPUT.PUT_LINE('........ v_common: ' || v_common);
            DBMS_OUTPUT.PUT_LINE('........ level_2a.v_common: ' || level_2a.v_common);
            DBMS_OUTPUT.PUT_LINE('........ level_1a.v_common: ' || level_1a.v_common);
            DBMS_OUTPUT.PUT_LINE('........ level_0.v_common: ' || level_0.v_common);
            DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2a');
        END level_2a;
    BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
        DBMS_OUTPUT.PUT_LINE('.... v_common: ' || v_common);
        DBMS_OUTPUT.PUT_LINE('.... level_0.v_common: ' || level_0.v_common);
        level_2a;
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
    END level_1a;
    PROCEDURE level_1b
    IS
        v_common    VARCHAR2(20) := 'Value from level_1b';
    BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
        DBMS_OUTPUT.PUT_LINE('.... v_common: ' || v_common);
        DBMS_OUTPUT.PUT_LINE('.... level_0.v_common : ' || level_0.v_common);
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
    END level_1b;
```

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
    DBMS_OUTPUT.PUT_LINE('.. v_common: ' || v_common);
    level_1a;
    level_1b;
    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;
```

The following output shows the content of each variable when the procedure is invoked:

```
BEGIN
    level_0;
END;

BLOCK level_0
.. v_common: Value from level_0
.. BLOCK level_1a
.... v_common: Value from level_1a
.... level_0.v_common: Value from level_0
...... BLOCK level_2a
........ v_common: Value from level_2a
........ level_2a.v_common: Value from level_2a
........ level_1a.v_common: Value from level_1a
........ level_0.v_common: Value from level_0
...... END BLOCK level_2a
.. END BLOCK level_1a
.. BLOCK level_1b
.... v_common: Value from level_1b
.... level_0.v_common : Value from level_0
.. END BLOCK level_1b
END BLOCK level_0
```

As previously discussed, the labels on anonymous blocks can also be used to control access to variables. The following example shows variable access within a set of nested anonymous blocks:

```
DECLARE
    v_common        VARCHAR2(20) := 'Value from level_0';
BEGIN
    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
    DBMS_OUTPUT.PUT_LINE('.. v_common: ' || v_common);
    <<level_1a>>
    DECLARE
        v_common    VARCHAR2(20) := 'Value from level_1a';
    BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
        DBMS_OUTPUT.PUT_LINE('.... v_common: ' || v_common);
        <<level_2a>>
        DECLARE
            v_common    VARCHAR2(20) := 'Value from level_2a';
        BEGIN
            DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2a');
            DBMS_OUTPUT.PUT_LINE('........ v_common: ' || v_common);
            DBMS_OUTPUT.PUT_LINE('........ level_1a.v_common: ' || level_1a.v_common);
            DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2a');
        END;
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
    END;
    <<level_1b>>
    DECLARE
```

```
        v_common    VARCHAR2(20) := 'Value from level_1b';
    BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
        DBMS_OUTPUT.PUT_LINE('.... v_common: ' || v_common);
        DBMS_OUTPUT.PUT_LINE('.... level_1b.v_common: ' || level_1b.v_common);
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
    END;
    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END;
```

The following output shows the content of each variable when the anonymous block is invoked:

```
BLOCK level_0
.. v_common: Value from level_0
.. BLOCK level_1a
.... v_common: Value from level_1a
...... BLOCK level_2a
........ v_common: Value from level_2a
........ level_1a.v_common: Value from level_1a
...... END BLOCK level_2a
.. END BLOCK level_1a
.. BLOCK level_1b
.... v_common: Value from level_1b
.... level_1b.v_common: Value from level_1b
.. END BLOCK level_1b
END BLOCK level_0
```

The following example is an object type whose object type method of display_emp contains the emp_typ record type and the emp_sal_query subprocedure. The r_emp record variable declared locally to emp_sal_query is able to access the emp_typ record type declared in the display_emp parent block.

```
CREATE OR REPLACE TYPE emp_pay_obj_typ AS OBJECT
(
    empno           NUMBER(4),
    MEMBER PROCEDURE display_emp(SELF IN OUT emp_pay_obj_typ)
);

CREATE OR REPLACE TYPE BODY emp_pay_obj_typ AS
    MEMBER PROCEDURE display_emp (SELF IN OUT emp_pay_obj_typ)
    IS
        TYPE emp_typ IS RECORD (
            ename           emp.ename%TYPE,
            job             emp.job%TYPE,
            hiredate        emp.hiredate%TYPE,
            sal             emp.sal%TYPE,
            deptno          emp.deptno%TYPE
         );
        PROCEDURE emp_sal_query (
            p_empno         IN emp.empno%TYPE
        )
        IS
            r_emp           emp_typ;
            v_avgsal        emp.sal%TYPE;
        BEGIN
            SELECT ename, job, hiredate, sal, deptno
                INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
                FROM emp WHERE empno = p_empno;
            DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
            DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
            DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
            DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
            DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
            DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);

            SELECT AVG(sal) INTO v_avgsal
            FROM emp WHERE deptno = r_emp.deptno;
            IF r_emp.sal > v_avgsal THEN
                DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '
                    || 'department average of ' || v_avgsal);
            ELSE
                DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the '
                    || 'department average of ' || v_avgsal);
            END IF;
        END;
    BEGIN
        emp_sal_query(SELF.empno);
    END;
END;
```

The following output is displayed when an instance of the object type is created and the display_emp procedure is invoked:

```
DECLARE
    v_emp           EMP_PAY_OBJ_TYP;
BEGIN
    v_emp := emp_pay_obj_typ(7900);
    v_emp.display_emp;
END;

Employee # : 7900
Name       : JAMES
Job        : CLERK
Hire Date  : 03-DEC-81 00:00:00
Salary     : 950.00
Dept #     : 30
Employee's salary does not exceed the department average of 1566.67
```

The following example is a package with three levels of subprocedures. A record type, collection type, and cursor type declared in the upper level procedure can be accessed by the descendent subprocedure.

```
CREATE OR REPLACE PACKAGE emp_dept_pkg
IS
    PROCEDURE display_emp (
        p_deptno        NUMBER
    );
END;

CREATE OR REPLACE PACKAGE BODY emp_dept_pkg
IS
    PROCEDURE display_emp (
        p_deptno        NUMBER
    )
    IS
        TYPE emp_rec_typ IS RECORD (
            empno           emp.empno%TYPE,
            ename           emp.ename%TYPE
        );
        TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
        TYPE emp_cur_type IS REF CURSOR RETURN emp_rec_typ;
        PROCEDURE emp_by_dept (
            p_deptno        emp.deptno%TYPE
        )
        IS
            emp_arr         emp_arr_typ;
            emp_refcur      emp_cur_type;
            i               BINARY_INTEGER := 0;
            PROCEDURE display_emp_arr
            IS
            BEGIN
                DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
                DBMS_OUTPUT.PUT_LINE('-----    -------');
                FOR j IN emp_arr.FIRST .. emp_arr.LAST LOOP
                    DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
                        emp_arr(j).ename);
                END LOOP;
            END display_emp_arr;
        BEGIN
            OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno =
p_deptno;
            LOOP
                i := i + 1;
                FETCH emp_refcur INTO emp_arr(i).empno, emp_arr(i).ename;
                EXIT WHEN emp_refcur%NOTFOUND;
            END LOOP;
            CLOSE emp_refcur;
            display_emp_arr;
        END emp_by_dept;
    BEGIN
        emp_by_dept(p_deptno);
    END;
END;
```

The following output is generated when the top level package procedure is invoked:

```
BEGIN
    emp_dept_pkg.display_emp(20);
END;

EMPNO    ENAME
-----    -------
7369     SMITH
7566     JONES
7788     SCOTT
7876     ADAMS
7902     FORD
```

# 7.6 Program security

## 7.6.1 EXECUTE privileges

An SPL program (function, procedure, or package) can begin execution only if any of the following conditions are met:

- The current user has been granted the EXECUTE privilege on the SPL program.
- The current user inherits the EXECUTE privilege on the SPL program by virtue of being a member of a group which have such privilege.
- EXECUTE privilege has been granted to the PUBLIC group.

Whenever you create an SPL program in a PolarDB database compatible with Oracle, the EXECUTE privilege is automatically granted to the PUBLIC group by default. Therefore, any user can immediately execute the program.

This default privilege can be removed by using the REVOKE EXECUTE statement. Example:

```
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
```

Then explicit the EXECUTE privilege on the program can be granted to individual users or groups.

```
GRANT EXECUTE ON PROCEDURE list_emp TO john;
```

Now, User john can execute the list_emp program. Other users who do not meet any of the conditions listed at the beginning of this section cannot.

After a program begins execution, the next aspect of security is what privilege checks occur if the program attempts to perform an action on any database object including:

- Reading or modifying table or view data
- Creating, modifying, or deleting a database object such as a table, view, index, or sequence
- Obtaining the current or next value from a sequence
- Calling another program such as function, procedure, or package

Each such action can be protected by privileges on the database object that is allowed or disallowed for the user.

**Note:**

A database can have multiple objects of the same type with the same name, but each such object belonging to a different schema in the database. For more information about which object is being referenced by an SPL program in this case, see Database object name resolution.

## 7.6.2 Database object name resolution

A database object inside an SPL program may be referenced by the qualified name or by an unqualified name of the database object. A qualified name is in the form of schema.name where schema is the name of the schema under which the database object with identifier, name, exists. An unqualified name does not have the schema. portion. When a reference is made to a qualified name, there cannot be ambiguity as to exactly which database object is intended - it does or does not exist in the specified schema.

However, finding an object with an unqualified name requires the use of the search path of the current user. When a user becomes the current user of a session, a default search path is always associated with that user. The search path consists of a list of schemas, which are searched in left-to-right order for finding an unqualified database object reference. The object is considered non-existent if it cannot be found in any of the schemas in the search path. The default search path can be displayed in PSQL by using the SHOW search_path statement.

```
edb=# SHOW search_path;
   search_path
-----------------
 "$user", public
(1 row)
```

$user in the above search path is a generic placeholder that refers to the current user. Therefore, if the current user of the above session is enterprisedb, an unqualified database object would be searched for in the following schemas in this order - first, enterprisedb, then public.

After an unqualified name has been resolved in the search path, it can be determined if the current user has the appropriate privilege to perform the action on that specific object.

**Note:**

The concept of the search path is not compatible with Oracle databases. For an unqualified reference, Oracle looks only in the schema of the current user for the named database object. Also note that in Oracle, a user and the schema of the user is the same entity while in PolarDB databases compatible with Oracle, a user and a schema are two distinct objects.

## 7.6.3 Database object privileges

After an SPL program begins execution, any attempt to access a database object from within the program results in a check to ensure the current user has the authorization to perform the intended action against the referenced object. Privileges on database objects are respectively bestowed and removed by using the GRANT and REVOKE statements. If the current user attempts unauthorized access on a database object, the program will throw an exception.

## 7.6.4 Rights of definers and invokers

When an SPL program is about to execute, you need to determine what user is to be associated with this process. This user is referred to as the current user. Database object privileges of the current user are used to determine whether access to database objects referenced in the program will be permitted. The current, prevailing search path in effect when the program is invoked will be used to resolve any unqualified object references.

The selection of the current user is influenced by whether the SPL program was created with the rights of definers or invokers. The AUTHID clause determines that selection . Appearance of the clause AUTHID DEFINER gives the program rights of the definer. This is also the default value if the AUTHID clause is omitted. Use of the clause AUTHID CURRENT_USER gives the program rights of invokers. The following content summarizes the differences between the two rights:

- If a program has rights of the definer, the owner of the program becomes the current user when program execution begins. Database object privileges of the program owner are used to determine if access to a referenced object is permitted. In a program that has rights of the definer, it is irrelevant as to which user actually invoked the program.
- If a program has the rights of the invoker, the current user at the time when the program is called remains the current user while the program is executing (but not necessaril y within called subprograms, see the following bullet points). When a program that has rights of the invoker is invoked, the current user is typically the user that started the session (that is, made the database connection). However, the current user can be changed after the session has started by using the SET ROLE statement. In a program that has rights of the invoker, it is irrelevant as to which user actually owns the program.

From the previous definitions, the following observations can be made:

- If a program that has rights of the definer calls a program that has rights of the definer , the current user changes from the owner of the calling program to the owner of the called program during execution of the called program.
- If a program that has rights of the definer calls a program that has rights of the invoker, the owner of the calling program remains the current user during execution of both the calling and called programs.
- If a program that has rights of the invoker calls a program that has rights of the invoker, the current user of the calling program remains the current user during execution of the called program.
- If a program that has rights of the invoker calls a program that has rights of the definer , the current user switches to the owner of the program that has rights of the definer during execution of the called program.

The same principles apply if the called program in turn calls another program in the cases cited above.

## 7.6.5 Security examples

In the following example, a new database will be created along with two users: hr_mgr and sales_mgr. hr_mgr will own a copy of the entire sample application in the hr_mgr schema. sales_mgr will own a schema named sales_mgr that will have a copy of only the emp table containing only the employees who work in sales.

The list_emp procedure, hire_clerk function, and emp_admin package will be used in this example. All of the default privileges that are granted upon installation of the sample application will be removed and then be explicitly re-granted to present a more secure environment in this example.

The list_emp and hire_clerk programs will be changed from the default of definer rights to invoker rights. It will be then illustrated that when sales_mgr runs these programs, they act upon the emp table in the sales_mgr schema because the search path and privileges of sales_mgr will be used for name resolution and authorization checking.

The get_dept_name and hire_emp programs in the emp_admin package will then be executed by sales_mgr. In this case, the dept table and emp table in the hr_mgr schema will be accessed because hr_mgr is the owner of the emp_admin package which is using definer rights. Because the default search path is in effect with the $user placeholder, the schema matching the user (in this case, hr_mgr) is used to find the tables.

### Create a database and users

Create the hr database as user enterprisedb:

```
CREATE DATABASE hr;
```

Switch to the hr database and create users:

```
\c hr enterprisedb
CREATE USER hr_mgr IDENTIFIED BY password;
CREATE USER sales_mgr IDENTIFIED BY password;
```

### Create the sample application

Create the entire sample application owned by hr_mgr in the hr_mgr schema.

```
\c - hr_mgr
\i /usr/edb/as11/share/edb-sample.sql

BEGIN
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE VIEW
CREATE SEQUENCE
        .
        .
        .
CREATE PACKAGE
CREATE PACKAGE BODY
COMMIT
```

### Create the emp table in the sales_mgr schema

Create a subset of the emp table owned by sales_mgr in the sales_mgr schema.

```
\c – hr_mgr
GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
\c – sales_mgr
CREATE TABLE emp AS SELECT * FROM hr_mgr.emp WHERE job = 'SALESMAN';
```

In the above example, the GRANT USAGE ON SCHEMA statement is given to allow sales_mgr access into the hr_mgr schema to make a copy of the emp table of hr_mgr. This step is required in a PolarDB database compatible with Oracle but is not compatible with Oracle databases because Oracle does not have the concept of a schema that is distinct from its user.

### Remove default privileges

Remove all privileges to later illustrate the minimum required privileges.

```
\c – hr_mgr
REVOKE USAGE ON SCHEMA hr_mgr FROM sales_mgr;
REVOKE ALL ON dept FROM PUBLIC;
REVOKE ALL ON emp FROM PUBLIC;
REVOKE ALL ON next_empno FROM PUBLIC;
REVOKE EXECUTE ON FUNCTION new_empno() FROM PUBLIC;
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
REVOKE EXECUTE ON FUNCTION hire_clerk(VARCHAR2,NUMBER) FROM PUBLIC;
REVOKE EXECUTE ON PACKAGE emp_admin FROM PUBLIC;
```

**Change list_emp to invoker rights**

While you are connected as user hr_mgr, add the AUTHID CURRENT_USER clause to the list_emp program and resave it in the PolarDB database compatible with Oracle. When you are performing this step, make sure that you log on as hr_mgr. Otherwise, the modified program may wind up in the public schema instead of in the hr_mgr schema.

```
CREATE OR REPLACE PROCEDURE list_emp
AUTHID CURRENT_USER
IS
   v_empno        NUMBER(4);
   v_ename        VARCHAR2(10);
   CURSOR emp_cur IS
      SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
   OPEN emp_cur;
   DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
   DBMS_OUTPUT.PUT_LINE('-----    -------');
   LOOP
      FETCH emp_cur INTO v_empno, v_ename;
      EXIT WHEN emp_cur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(v_empno ‖ '     ' ‖ v_ename);
   END LOOP;
   CLOSE emp_cur;
END;
```

**Change hire_clerk to invoker rights and qualify call to new_empno**

While you are connected as user hr_mgr, add the AUTHID CURRENT_USER clause to the hire_clerk program.

Additionally, after the BEGIN statement, fully qualify the new_empno reference to hr_mgr.new_empno to ensure that the hire_clerk function call to the new_empno function resolves to the hr_mgr schema.

When you resave the program, make sure that you log on as hr_mgr. Otherwise, the modified program may wind up in the public schema instead of in the hr_mgr schema.

```
CREATE OR REPLACE FUNCTION hire_clerk (
   p_ename        VARCHAR2,
   p_deptno       NUMBER
) RETURN NUMBER
AUTHID CURRENT_USER
IS
   v_empno        NUMBER(4);
   v_ename        VARCHAR2(10);
   v_job          VARCHAR2(9);
   v_mgr          NUMBER(4);
```

```
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_deptno        NUMBER(2);
BEGIN
    v_empno := hr_mgr.new_empno;
    INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
        TRUNC(SYSDATE), 950.00, NULL, p_deptno);
    SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
        FROM emp WHERE empno = v_empno;
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Manager    : ' || v_mgr);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission : ' || v_comm);
    RETURN v_empno;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
        RETURN -1;
END;
```

### Grant required privileges

While you are connected as user hr_mgr, grant the privileges needed so sales_mgr can execute the list_emp procedure, hire_clerk function, and emp_admin package. Note that the only data object that can be accessed by sales_mgr is the emp table in the sales_mgr schema. sales_mgr has no privileges on any table in the hr_mgr schema.

```
GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
GRANT EXECUTE ON PROCEDURE list_emp TO sales_mgr;
GRANT EXECUTE ON FUNCTION hire_clerk(VARCHAR2,NUMBER) TO sales_mgr;
GRANT EXECUTE ON FUNCTION new_empno() TO sales_mgr;
GRANT EXECUTE ON PACKAGE emp_admin TO sales_mgr;
```

### Run the list_emp and hire_clerk programs

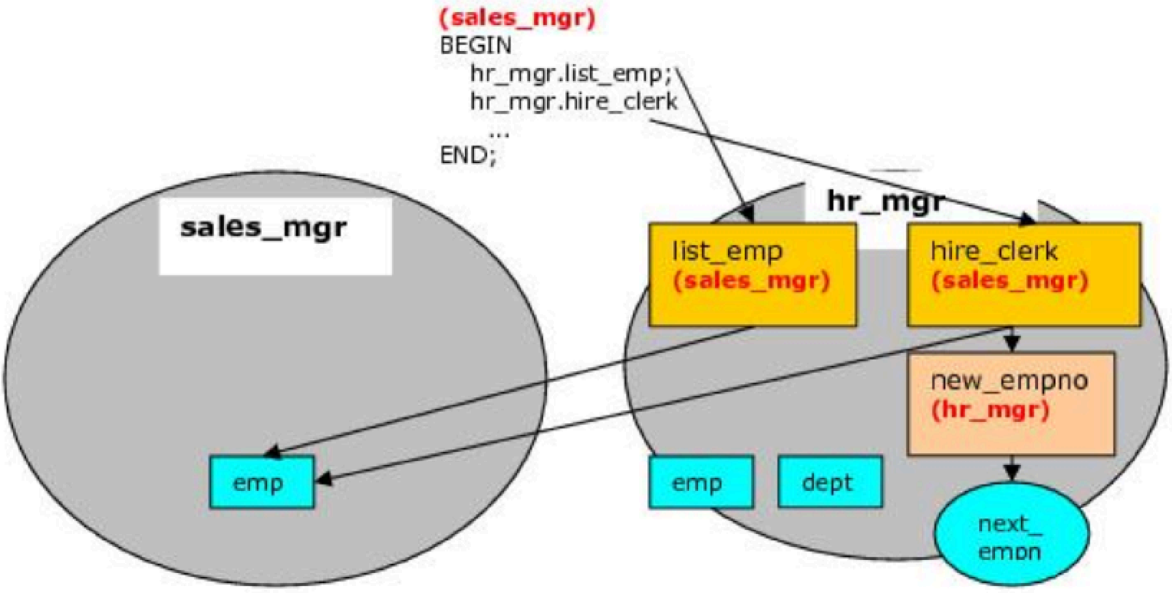
Connect as user sales_mgr and run the following anonymous block:

```
\c – sales_mgr
DECLARE
    v_empno         NUMBER(4);
BEGIN
    hr_mgr.list_emp;
    DBMS_OUTPUT.PUT_LINE('*** Adding new employee ***');
    v_empno := hr_mgr.hire_clerk('JONES',40);
    DBMS_OUTPUT.PUT_LINE('*** After new employee added ***');
    hr_mgr.list_emp;
END;

EMPNO    ENAME
-----    -------
```

```
7499    ALLEN
7521    WARD
7654    MARTIN
7844    TURNER
*** Adding new employee ***
Department : 40
Employee No: 8000
Name       : JONES
Job        : CLERK
Manager    : 7782
Hire Date  : 08-NOV-07 00:00:00
Salary     : 950.00
*** After new employee added ***
EMPNO    ENAME
-----    -------
7499    ALLEN
7521    WARD
7654    MARTIN
7844    TURNER
8000    JONES
```

The table and sequence accessed by the programs of the anonymous block are illustrated in the following diagram. The gray ovals represent the schemas of sales_mgr and hr_mgr . The current user during each program execution is shown within parenthesis in bold red font.



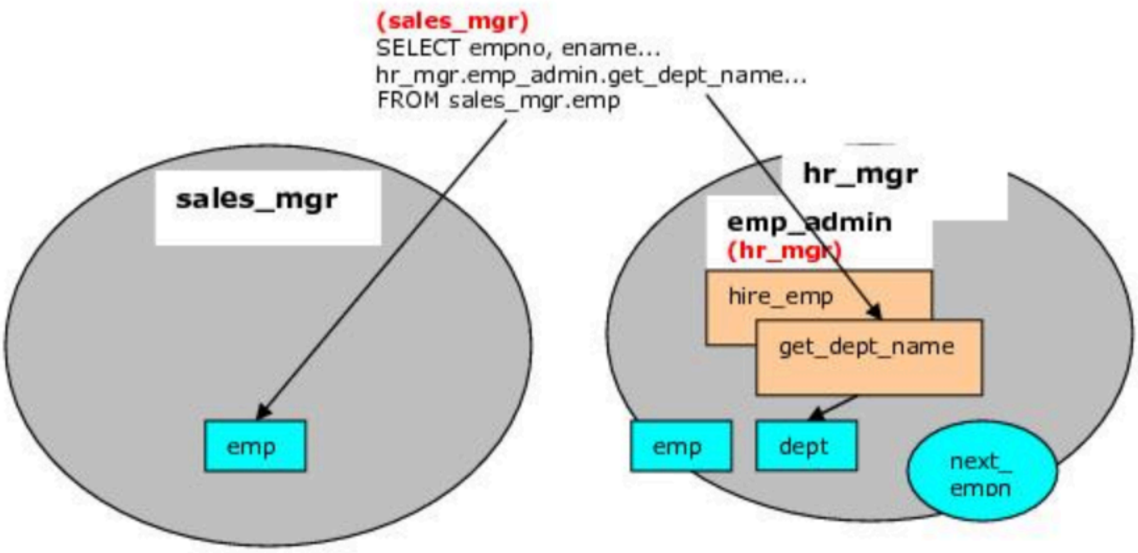
Selecting from the emp table of sales_mgr shows that the update was made in this table.

```
SELECT empno, ename, hiredate, sal, deptno, hr_mgr.emp_admin.get_dept_name(
deptno) FROM sales_mgr.emp;

empno | ename  |      hiredate      |   sal   | deptno | get_dept_name
-------+--------+--------------------+---------+--------+---------------
  7499 | ALLEN  | 20-FEB-81 00:00:00 | 1600.00 |     30 | SALES
  7521 | WARD   | 22-FEB-81 00:00:00 | 1250.00 |     30 | SALES
  7654 | MARTIN | 28-SEP-81 00:00:00 | 1250.00 |     30 | SALES
  7844 | TURNER | 08-SEP-81 00:00:00 | 1500.00 |     30 | SALES
```

```
 8000 | JONES  | 08-NOV-07 00:00:00 |  950.00 |     40 | OPERATIONS
(5 rows)
```

The following diagram shows that the SELECT statement references the emp table in the sales_mgr schema, but the dept table referenced by the get_dept_name function in the emp_admin package is from the hr_mgr schema because the emp_admin package has definer rights and is owned by hr_mgr. The default search path setting with the $user placeholder resolves the access by hr_mgr to the dept table in the hr_mgr schema.

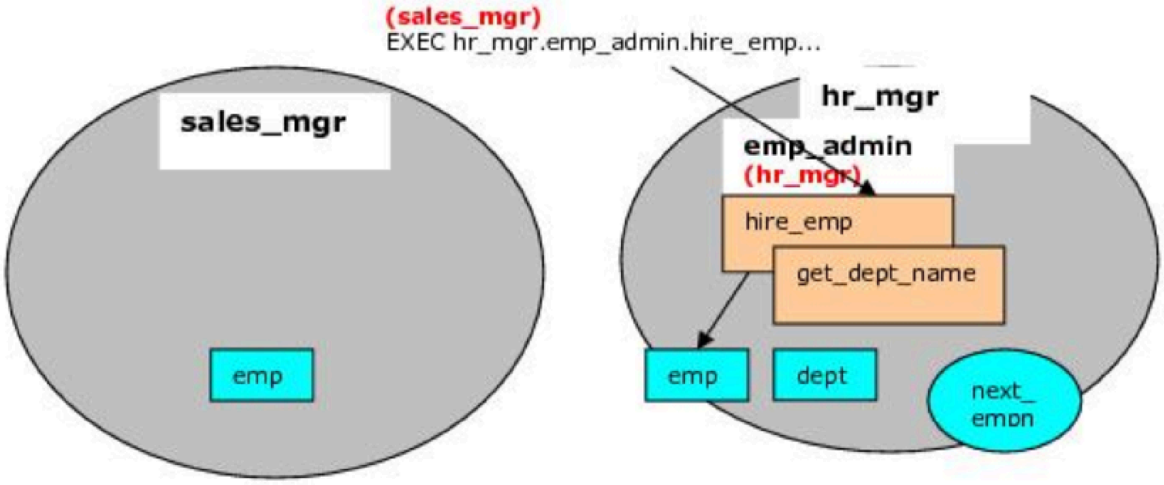


## Run the hire_emp program in the emp_admin package

While you are connected as user sales_mgr, run the hire_emp procedure in the emp_admin package.

```
EXEC hr_mgr.emp_admin.hire_emp(9001, 'ALICE','SALESMAN',8000,TRUNC(SYSDATE),1000
,7369,40);
```

This diagram illustrates that the hire_emp procedure in the rights package of the emp_admin definer updates the emp table belonging to hr_mgr because the object privileges of hr_mgr are used and the default search path setting with the $user placeholder resolves to the hr_mgr schema.



Connect as user hr_mgr. The following SELECT statement verifies that the new employee was added to the emp table of hr_mgr because the emp_admin package has definer rights and hr_mgr is the owner of emp_admin.

```
\c – hr_mgr
SELECT empno, ename, hiredate, sal, deptno, hr_mgr.emp_admin.get_dept_name(
deptno) FROM hr_mgr.emp;

empno | ename  |      hiredate      |   sal   | deptno | get_dept_name
-------+--------+--------------------+---------+--------+---------------
  7369 | SMITH  | 17-DEC-80 00:00:00 |  800.00 |     20 | RESEARCH
  7499 | ALLEN  | 20-FEB-81 00:00:00 | 1600.00 |     30 | SALES
  7521 | WARD   | 22-FEB-81 00:00:00 | 1250.00 |     30 | SALES
  7566 | JONES  | 02-APR-81 00:00:00 | 2975.00 |     20 | RESEARCH
  7654 | MARTIN | 28-SEP-81 00:00:00 | 1250.00 |     30 | SALES
  7698 | BLAKE  | 01-MAY-81 00:00:00 | 2850.00 |     30 | SALES
  7782 | CLARK  | 09-JUN-81 00:00:00 | 2450.00 |     10 | ACCOUNTING
  7788 | SCOTT  | 19-APR-87 00:00:00 | 3000.00 |     20 | RESEARCH
  7839 | KING   | 17-NOV-81 00:00:00 | 5000.00 |     10 | ACCOUNTING
  7844 | TURNER | 08-SEP-81 00:00:00 | 1500.00 |     30 | SALES
  7876 | ADAMS  | 23-MAY-87 00:00:00 | 1100.00 |     20 | RESEARCH
  7900 | JAMES  | 03-DEC-81 00:00:00 |  950.00 |     30 | SALES
  7902 | FORD   | 03-DEC-81 00:00:00 | 3000.00 |     20 | RESEARCH
  7934 | MILLER | 23-JAN-82 00:00:00 | 1300.00 |     10 | ACCOUNTING
  9001 | ALICE  | 08-NOV-07 00:00:00 | 8000.00 |     40 | OPERATIONS
(15 rows)
```

## 7.7 Variable declarations

## 7.7.1 Declare a variable

SPL is a block-structured language. The first section that can appear in a block is the declaration section. The declaration section contains the definition of variables, cursors, and other types that can be used in SPL statements contained in the block.

Typically, all variables used in a block must be declared in the declaration section of the block. A variable declaration consists of a name that is assigned to the variable and the data type of the variable. Optionally, the variable can be initialized to a default value in the variable declaration.

The following example shows the general syntax of a variable declaration:

```
name type [ { := | DEFAULT } { expression | NULL } ];
```

- name is an identifier assigned to the variable.
- type is the data type assigned to the variable.

[ := expression ], if given, specifies the initial value assigned to the variable when the block is entered. If the clause is not given, the variable is initialized to the SQL NULL value.

The default value is evaluated every time the block is entered. For example, assigning SYSDATE to a variable of the DATE type causes the variable to have the time of the current invocation, not the time when the procedure or function was precompiled.

The following procedure illustrates some variable declarations that utilize default values consisting of string and numeric expressions.

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno        NUMBER
)
IS
    todays_date     DATE := SYSDATE;
    rpt_title       VARCHAR2(60) := 'Report For Department # ' || p_deptno
        || ' on ' || todays_date;
    base_sal        INTEGER := 35525;
    base_comm_rate  NUMBER := 1.33333;
    base_annual     NUMBER := ROUND(base_sal * base_comm_rate, 2);
BEGIN
    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

The following output of the above procedure shows that default values in the variable declarations are indeed assigned to the variables.

```
EXEC dept_salary_rpt(20);

Report For Department # 20 on 10-JUL-07 16:44:45
```

```
Base Annual Salary: 47366.55
```

## 7.7.2 Use %TYPE in variable declarations

Typically, variables that are used to hold values from tables in a database are declared in SPL programs. To ensure compatibility between the table columns and the SPL variables, the data types of the columns and variables must be the same.

However, as quite often happens, a change might be made to the table definition. If the data type of the column is changed, the corresponding change may be required to the variable in the SPL program.

Instead of coding the specific column data type into the variable declaration, the %TYPE column attribute can be used. A qualified column name in dot notation or the name of a previously declared variable must be specified as a prefix to %TYPE. The data type of the column or variable prefixed to %TYPE is assigned to the variable being declared. If the data type of the given column or variable changes, the new data type will be associated with the variable without the need to modify the declaration code.

**Note:**

The %TYPE attribute can also be used with formal parameter declarations.

```
name { { table | view }.column | variable }%TYPE;
```

name is the identifier assigned to the variable or formal parameter that is being declared. column is the name of a column in table or view. variable is the name of a variable that was declared prior to the variable identified by name.

**Note:**

The variable does not inherit any other attributes of the column such as the attributes that might be specified on the column by using the NOT NULL clause or the DEFAULT clause.

In the following example, a procedure queries the emp table by using an employee number , displays data about the employee, finds the average salary of all employees in the department to which the employee belongs, and then compares the salary of the chosen employee with the department average.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN NUMBER
)
IS
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal          NUMBER(7,2);
    v_deptno        NUMBER(2);
    v_avgsal        NUMBER(7,2);
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '
            || 'department average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the '
            || 'department average of ' || v_avgsal);
    END IF;
END;
```

Instead of the above, you can write the procedure as follows without explicitly coding the emp table data types into the declaration section of the procedure.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_avgsal        v_sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '
            || 'department average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the '
            || 'department average of ' || v_avgsal);
    END IF;
```

```
END;
```

**Note:**

p_empno shows an example of a formal parameter defined using %TYPE.

v_avgsal illustrates the usage of %TYPE referring to another variable instead of a table column.

The following example shows the sample output from executing this procedure:

```
EXEC emp_sal_query(7698);

Employee # : 7698
Name       : BLAKE
Job        : MANAGER
Hire Date  : 01-MAY-81 00:00:00
Salary     : 2850.00
Dept #     : 30
Employee's salary is more than the department average of 1566.67
```

## 7.7.3 Use %ROWTYPE in record declarations

The %TYPE attribute provides an easy way to create a variable dependent upon the data type of a column. Using the %ROWTYPE attribute, you can define a record that contains fields that correspond to all columns of a given table. Each field takes on the data type of its corresponding column. The fields in the record do not inherit any other attributes of the columns such as the attributes that might be specified by using the NOT NULL clause or the DEFAULT clause.

A record is a named, ordered collection of fields. A field is similar to a variable. A field has an identifier and data type, but has the additional property of belonging to a record. A field must be referenced using dot notation with the record name as its qualifier.

You can use the %ROWTYPE attribute to declare a record. The %ROWTYPE attribute is prefixed by a table name. Each column in the named table defines an identically named field in the record with the same data type as the column.

```
record table%ROWTYPE;
```

record is an identifier assigned to the record. table is the name of a table (or view) whose columns are to define the fields in the record. The following example shows how the emp_sal_query procedure from the prior topic can be modified to use emp%ROWTYPE to create a record named r_emp instead of declaring individual variables for the columns in emp.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
    v_avgsal        emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);
    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '
            || 'department average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the '
            || 'department average of ' || v_avgsal);
    END IF;
END;
```

## 7.7.4 User-defined record types and record variables

Records can be declared based on a table definition by using the %ROWTYPE attribute as shown in Use %ROWTYPE in record declarations. This topic describes how to define a new record structure that is not tied to any particular table definition.

The TYPE IS RECORD statement is used to create the definition of a record type. A record type is a definition of a record comprised of one or more identifiers and their corresponding data types. A record type cannot, by itself, be used to manipulate data.

The following example shows the syntax for a TYPE IS RECORD statement:

```
TYPE rec_type IS RECORD ( fields )
```

fields is a comma-separated list of one or more field definitions in the following form:

```
field_name data_type [NOT NULL][{:= | DEFAULT} default_value]
```

The following table describes parameters in the preceding statement.

| Parameter | Description |
|---|---|
| rec_type | rec_type is an identifier assigned to the record type. |

| Parameter | Description |
|---|---|
| field_name | field_name is the identifier assigned to the field of the record type. |
| data_type | data_type specifies the data type of field_name. |
| DEFAULT default_value | The DEFAULT clause assigns a default data value for the corresponding field. The data type of the default expression must match the data type of the column. If no default value is specified, the default value is NULL. |

A record variable or simply put, a record, is an instance of a record type. A record is declared from a record type. The properties of the record such as field names and types are inherited from the record type.

The following example shows the syntax for a record declaration:

```
record rectype
```

record is an identifier assigned to the record variable. rectype is the identifier of a previously defined record type. After being declared, a record can be used to hold data.

Dot notation is used to make reference to the fields in the record.

```
record.field
```

record is a previously declared record variable and field is the identifier of a field belonging to the record type from which record is defined.

emp_sal_query is again modified - this time using a user-defined record type and record variable.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    TYPE emp_typ IS RECORD (
        ename       emp.ename%TYPE,
        job         emp.job%TYPE,
        hiredate    emp.hiredate%TYPE,
        sal         emp.sal%TYPE,
        deptno      emp.deptno%TYPE
    );
    r_emp           emp_typ;
    v_avgsal        emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' ‖ p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' ‖ r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' ‖ r_emp.job);
```

```
   DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
   DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
   DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);

   SELECT AVG(sal) INTO v_avgsal
      FROM emp WHERE deptno = r_emp.deptno;
   IF r_emp.sal > v_avgsal THEN
      DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '
         || 'department average of ' || v_avgsal);
   ELSE
      DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the '
         || 'department average of ' || v_avgsal);
   END IF;
END;
```

Note that instead of specifying data type names, you can use the %TYPE attribute for the field data types in the record type definition.

The following output is generated after this stored procedure is executed:

```
EXEC emp_sal_query(7698);

Employee # : 7698
Name       : BLAKE
Job        : MANAGER
Hire Date  : 01-MAY-81 00:00:00
Salary     : 2850.00
Dept #     : 30
Employee's salary is more than the department average of 1566.67
```

# 7.8 Basic statements

## 7.8.1 NULL

The simplest statement is the NULL statement. This statement is an executable statement that does nothing.

```
NULL;
```

The following example shows the simplest, possible valid SPL program:

```
BEGIN
   NULL;
END;
```

The NULL statement can act as a placeholder where an executable statement is required such as in a branch of an IF-THEN-ELSE statement.

Example:

```
CREATE OR REPLACE PROCEDURE divide_it (
   p_numerator     IN  NUMBER,
   p_denominator   IN  NUMBER,
    p_result        OUT NUMBER
)
IS
BEGIN
    IF p_denominator = 0 THEN
        NULL;
    ELSE
        p_result := p_numerator / p_denominator;
    END IF;
END;
```

## 7.8.2 Assignment

An assignment statement sets a variable or a formal parameter of OUT or IN OUT mode specified on the left side of the assignment operator := to the evaluated expression specified on the right side of the assignment operator.

```
variable := expression;
```

variable is an identifier for a previously declared variable, OUT formal parameter, or IN OUT formal parameter.

expression is an expression that produces a single value. The value produced by the expression must have a compatible data type with that of variable.

The following example shows the typical use of assignment statements in the executable section of a procedure:

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno        NUMBER
)
IS
    todays_date     DATE;
    rpt_title       VARCHAR2(60);
    base_sal        INTEGER;
    base_comm_rate  NUMBER;
    base_annual     NUMBER;
BEGIN
    todays_date := SYSDATE;
    rpt_title := 'Report For Department # ' || p_deptno || ' on '
        || todays_date;
    base_sal := 35525;
    base_comm_rate := 1.33333;
    base_annual := ROUND(base_sal * base_comm_rate, 2);

    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

## 7.8.3 SELECT INTO

The SELECT INTO statement is an SPL variation of the SQL SELECT statement. The differences are as follows:

- The SELECT INTO statement is designed to assign the results to variables or records where they can then be used in SPL program statements.
- The accessible result set of SELECT INTO contains at most one row.

Other than the above, all of the clauses of the SELECT statement, such as WHERE, ORDER BY , GROUP BY, and HAVING, are valid for SELECT INTO. The following example shows the two variations of SELECT INTO:

```
SELECT select_expressions INTO target FROM ... ;
```

target is a comma-separated list of simple variables. select_expressions and the remainder of the statement are the same as those of the SELECT statement. The selected values must exactly match the structure of the target in data type, number, and order. Otherwise, a runtime error occurs.

```
SELECT * INTO record FROM table ... ;
```

record is a record variable that has previously been declared.

If the query returns zero rows, null values are assigned to the target. If the query returns multiple rows, the first row is assigned to the target and the rest are discarded. Note that " the first row" is not well-defined unless you have used ORDER BY.

**Note:**

If no row is returned or more than one row is returned, SPL throws an exception.

A variation of SELECT INTO uses the BULK COLLECT clause. The variation allows a result set of more than one row that is returned into a collection.

You can use the WHEN NO_DATA_FOUND clause in an EXCEPTION block to determine whether the assignment was successful. When the assignment was successful, at least one row was returned by the query.

This version of the emp_sal_query procedure uses the variation of SELECT INTO that returns the result set into a record. Note the addition of the EXCEPTION block containing the WHEN NO_DATA_FOUND conditional expression.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
    v_avgsal        emp.sal%TYPE;
BEGIN
    SELECT * INTO r_emp
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '
            || 'department average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the '
            || 'department average of ' || v_avgsal);
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
END;
```

If the query is executed with a non-existent employee number, the following results appear :

```
EXEC emp_sal_query(0);

Employee # 0 not found
```

Another conditional clause used in the EXCEPTION section with SELECT INTO is the TOO_MANY_ROWS exception. If more than one row is selected by the SELECT INTO statement, an exception is thrown by SPL.

When the following block is executed, the TOO_MANY_ROWS exception is thrown because many employees exist in the specified department.

```
DECLARE
    v_ename         emp.ename%TYPE;
BEGIN
    SELECT ename INTO v_ename FROM emp WHERE deptno = 20 ORDER BY ename;
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('More than one employee found');
        DBMS_OUTPUT.PUT_LINE('First employee returned is ' || v_ename);
END;

More than one employee found
First employee returned is ADAMS
```

## 7.8.4 INSERT

The INSERT statement available in the SQL language can also be used in SPL programs.

An expression in the SPL language can be used wherever an expression is allowed in the SQL INSERT statement. Therefore, SPL variables and parameters can be used to supply values to the insert operation.

The following example shows the procedure that inserts a new employee by using data passed from a calling program:

```
CREATE OR REPLACE PROCEDURE emp_insert (
    p_empno         IN emp.empno%TYPE,
    p_ename         IN emp.ename%TYPE,
    p_job           IN emp.job%TYPE,
    p_mgr           IN emp.mgr%TYPE,
    p_hiredate      IN emp.hiredate%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm          IN emp.comm%TYPE,
    p_deptno        IN emp.deptno%TYPE
)
IS
BEGIN
    INSERT INTO emp VALUES (
        p_empno,
        p_ename,
        p_job,
        p_mgr,
        p_hiredate,
        p_sal,
        p_comm,
        p_deptno);

    DBMS_OUTPUT.PUT_LINE('Added employee...') ;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || p_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || p_job);
    DBMS_OUTPUT.PUT_LINE('Manager    : ' || p_mgr);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || p_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || p_sal);
    DBMS_OUTPUT.PUT_LINE('Commission : ' || p_comm);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || p_deptno);
    DBMS_OUTPUT.PUT_LINE('----------------------');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('OTHERS exception on INSERT of employee # '
            || p_empno);
        DBMS_OUTPUT.PUT_LINE('SQLCODE : ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('SQLERRM : ' || SQLERRM);
END;
```

If an exception occurs, all database changes made in the procedure are automatically rolled back. In this example, the EXCEPTION section with the WHEN OTHERS clause catches all exceptions. Two variables appear. SQLCODE is a number that identifies the specific exception that occurred. SQLERRM is a text message explaining the error.

The following output is generated when this procedure is executed:

```
EXEC emp_insert(9503,'PETERSON','ANALYST',7902,'31-MAR-05',5000,NULL,40);

Added employee...
Employee # : 9503
Name       : PETERSON
Job        : ANALYST
Manager    : 7902
Hire Date  : 31-MAR-05 00:00:00
```

```
Salary     : 5000
Dept #     : 40
----------------------

SELECT * FROM emp WHERE empno = 9503;

 empno | ename  | job   | mgr |     hiredate      | sal   | comm | deptno
-------+----------+---------+------+--------------------+---------+------+--------
  9503 | PETERSON | ANALYST | 7902 | 31-MAR-05 00:00:00 | 5000.00 |      |     40
(1 row)
```

**Note:**

The INSERT statement can be included in a FORALL statement. A FORALL statement allows a single INSERT statement to insert multiple rows from values supplied in one or more collections.

## 7.8.5 UPDATE

The UPDATE statement available in the SQL language can also be used in SPL programs.

An expression in the SPL language can be used wherever an expression is allowed in the SQL UPDATE statement. Therefore, SPL variables and parameters can be used to supply values to the update operation.

```
CREATE OR REPLACE PROCEDURE emp_comp_update (
    p_empno         IN emp.empno%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm          IN emp.comm%TYPE
)
IS
BEGIN
    UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || p_empno);
        DBMS_OUTPUT.PUT_LINE('New Salary         : ' || p_sal);
        DBMS_OUTPUT.PUT_LINE('New Commission     : ' || p_comm);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

If a row is updated, the SQL%FOUND conditional expression returns TRUE. Otherwise, the expression returns FALSE.

The following example shows the update on the employee using this procedure:

```
EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
New Salary         : 6540
New Commission     : 1200

SELECT * FROM emp WHERE empno = 9503;
```

```
 empno | ename   |  job   | mgr |     hiredate      |  sal   | comm   | deptno
-------+----------+---------+------+--------------------+---------+---------+--------
  9503 | PETERSON | ANALYST | 7902 | 31-MAR-05 00:00:00 | 6540.00 | 1200.00 |     40
(1 row)
```

**Note:**

The UPDATE statement can be included in a FORALL statement. A FORALL statement allows a single UPDATE statement to update multiple rows from values supplied in one or more collections.

### 7.8.6 DELETE

The DELETE statement available in the SQL language can also be used in SPL programs.

An expression in the SPL language can be used wherever an expression is allowed in the SQL DELETE statement. Therefore, SPL variables and parameters can be used to supply values to the delete operation.

```
CREATE OR REPLACE PROCEDURE emp_delete (
    p_empno         IN emp.empno%TYPE
)
IS
BEGIN
    DELETE FROM emp WHERE empno = p_empno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || p_empno);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

If a row is deleted, the SQL%FOUND conditional expression returns TRUE. Otherwise, the expression returns FALSE.

```
EXEC emp_delete(9503);

Deleted Employee # : 9503

SELECT * FROM emp WHERE empno = 9503;

 empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
```

```
(0 rows)
```

## 7.8.7 Use the RETURNING INTO clause

The INSERT, UPDATE, and DELETE statements may be appended by the optional RETURNING INTO clause. This clause allows the SPL program to capture the newly added, modified, or deleted values from the results of an INSERT, an UPDATE, or a DELETE statement.

The following example shows the syntax:

```
{ insert | update | delete }
  RETURNING { * | expr_1 [, expr_2 ] ...}
    INTO { record | field_1 [, field_2 ] ...} ;
```

insert is a valid INSERT statement. update is a valid UPDATE statement. delete is a valid DELETE statement. If * is specified, the values from the row affected by the INSERT, UPDATE , or DELETE statement are made available for assignment to the record or fields to the right of the INTO keyword. (Note that the use of * is an extension for PolarDB databases compatible with Oracle and is not compatible with Oracle databases.) expr_1, expr_2... are expressions evaluated upon the row affected by the INSERT, UPDATE, or DELETE statement . The evaluated results are assigned to the record or fields to the right of the INTO keyword . record is the identifier of a record that must contain fields that match in number and order , and are data type compatible with the values in the RETURNING clause. field_1, field_2,... are variables that must match in number and order, and are data type compatible with the set of values in the RETURNING clause.

If the INSERT, UPDATE, or DELETE statement returns a result set with more than one row, an exception is thrown with the message of "SQLCODE 01422, query returned more than one row." If no rows are in the result set, the variables following the INTO keyword are set to null .

**Note:**

A variation of RETURNING INTO uses the BULK COLLECT clause. The variation allows a result set of more than one row that is returned into a collection.

The following example is a modification of the emp_comp_update procedure introduced in UPDATE, with the addition of the RETURNING INTO clause:

```
CREATE OR REPLACE PROCEDURE emp_comp_update (
    p_empno         IN emp.empno%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm          IN emp.comm%TYPE
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_deptno        emp.deptno%TYPE;
BEGIN
    UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno
    RETURNING
        empno,
        ename,
        job,
        sal,
        comm,
        deptno
    INTO
        v_empno,
        v_ename,
        v_job,
        v_sal,
        v_comm,
        v_deptno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || v_empno);
        DBMS_OUTPUT.PUT_LINE('Name              : ' || v_ename);
        DBMS_OUTPUT.PUT_LINE('Job               : ' || v_job);
        DBMS_OUTPUT.PUT_LINE('Department        : ' || v_deptno);
        DBMS_OUTPUT.PUT_LINE('New Salary        : ' || v_sal);
        DBMS_OUTPUT.PUT_LINE('New Commission    : ' || v_comm);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

The following example shows the output from this procedure (assuming that employee 9503 created by the emp_insert procedure still exists within the table):

```
EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Department         : 40
New Salary         : 6540.00
New Commission     : 1200.00
```

The following example is a modification of the emp_delete procedure, with the addition of the RETURNING INTO clause using record types:

```
CREATE OR REPLACE PROCEDURE emp_delete (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
BEGIN
    DELETE FROM emp WHERE empno = p_empno
    RETURNING
        *
    INTO
        r_emp;
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || r_emp.empno);
        DBMS_OUTPUT.PUT_LINE('Name               : ' || r_emp.ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || r_emp.job);
        DBMS_OUTPUT.PUT_LINE('Manager            : ' || r_emp.mgr);
        DBMS_OUTPUT.PUT_LINE('Hire Date          : ' || r_emp.hiredate);
        DBMS_OUTPUT.PUT_LINE('Salary             : ' || r_emp.sal);
        DBMS_OUTPUT.PUT_LINE('Commission         : ' || r_emp.comm);
        DBMS_OUTPUT.PUT_LINE('Department         : ' || r_emp.deptno);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

The following example shows the output from this procedure:

```
EXEC emp_delete(9503);

Deleted Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Manager            : 7902
Hire Date          : 31-MAR-05 00:00:00
Salary             : 6540.00
Commission         : 1200.00
Department         : 40
```

## 7.8.8 Obtain the result status

Several attributes can be used to determine the effect of a statement. SQL%FOUND has a Boolean value. SQL%FOUND returns TRUE if at least one row was affected by an INSERT, an UPDATE, or a DELETE statement or if a SELECT INTO statement retrieved one or more rows.

The following anonymous block inserts a row and then displays the fact that the row has been inserted:

```
BEGIN
    INSERT INTO emp (empno,ename,job,sal,deptno) VALUES (
        9001, 'JONES', 'CLERK', 850.00, 40);
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Row has been inserted');
    END IF;
END;

Row has been inserted
```

SQL%ROWCOUNT provides the number of rows affected by an INSERT, an UPDATE, a DELETE , or a SELECT INTO statement. The SQL%ROWCOUNT value is returned as a BIGINT data type. The following example updates the row that was just inserted and displays SQL% ROWCOUNT:

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9001;
    DBMS_OUTPUT.PUT_LINE('# rows updated: ' || SQL%ROWCOUNT);
```

```
END;

# rows updated: 1
```

SQL%NOTFOUND is the opposite of SQL%FOUND. SQL%NOTFOUND returns TRUE if no rows were affected by an INSERT, an UPDATE, or a DELETE statement or if a SELECT INTO statement retrieved no rows.

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9000;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('No rows were updated');
    END IF;
END;

No rows were updated
```

# 7.9 Control structures

## 7.9.1 RETURN statement

The RETURN statement terminates the current function, procedure, or anonymous block and returns control to the caller.

The RETURN statement is available in two forms. The first form of the RETURN statement is used to terminate a procedure or function that returns void. The following example shows the syntax of the first form:

```
RETURN;
```

The second form of the RETURN statement returns a value to the caller. The following example shows the syntax of the second form:

```
RETURN expression;
```

expression must evaluate to the same data type as the return type of the function.

The following example uses the RETURN statement to return a value to the caller:

```
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal          NUMBER,
    p_comm         NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
```

```
END emp_comp;
```

## 7.9.2 GOTO statement

The GOTO statement causes the point of execution to jump to the statement with the specified label. The following example shows the syntax of the GOTO statement:

```
GOTO label
```

label is a name assigned to an executable statement. label must be unique within the scope of the function, procedure, or anonymous block.

To label a statement, use the following syntax:

```
<<label>> statement
```

statement is the point of execution that the program jumps to.

You can label assignment statements, any SQL statement (such as INSERT, UPDATE, and CREATE), and selected procedural language statements. The following procedural language statements can be labeled:

- IF
- EXIT
- RETURN
- RAISE
- EXECUTE
- PERFORM
- GET DIAGNOSTICS
- OPEN
- FETCH
- MOVE
- CLOSE
- NULL
- COMMIT
- ROLLBACK
- GOTO
- CASE
- LOOP
- WHILE

- FOR

Note that exit is considered as a keyword, and cannot be used as the name of a label.

GOTO statements cannot transfer control into a conditional block or sub-block, but can transfer control from a conditional block or sub-block.

The following example verifies that an employee record contains a name, a job descriptio n, and an employee hire date. If any piece of information is missing, a GOTO statement transfers the point of execution to a statement that prints a message that the employee is not valid.

```
CREATE OR REPLACE PROCEDURE verify_emp (
    p_empno         NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_hiredate      emp.hiredate%TYPE;
BEGIN
    SELECT ename, job, hiredate
        INTO v_ename, v_job, v_hiredate FROM emp
        WHERE empno = p_empno;
    IF v_ename IS NULL THEN
        GOTO invalid_emp;
    END IF;
    IF v_job IS NULL THEN
        GOTO invalid_emp;
    END IF;
    IF v_hiredate IS NULL THEN
        GOTO invalid_emp;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
        ' validated without errors.') ;
    RETURN;
    <<invalid_emp>> DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
        ' is not a valid employee.') ;
END;
```

GOTO statements have the following restrictions:

- A GOTO statement cannot jump to a declaration.
- A GOTO statement cannot transfer control to another function or procedure.

A label should not be placed at the end of a block, function, or procedure.

## 7.9.3 CASE expression

The CASE expression returns a value that is substituted where the CASE expression is located within an expression.

The CASE expression is available in two formats. One is called a searched CASE and the other uses a selector.

### Selector CASE expression

The selector CASE expression attempts to match an expression called the selector to the expression specified in one or more WHEN clauses. result is an expression that is type-compatible in the context where the CASE expression is used. If a match is found, the value given in the corresponding THEN clause is returned by the CASE expression. If no match is found, the value following ELSE is returned. If ELSE is omitted, the CASE expression returns null.

```
CASE selector-expression
  WHEN match-expression THEN
    result
[ WHEN match-expression THEN
    result
[ WHEN match-expression THEN
    result ] ...]
[ ELSE
    result ]
END;
```

match-expression is evaluated in the order in which it appears within the CASE expression . result is an expression that is type-compatible in the context where the CASE expression is used. When the first match-expression that equals selector-expression is encountered, result in the corresponding THEN clause is returned as the value of the CASE expression. If none of match-expression equals selector-expression, result following ELSE is returned. If no ELSE is specified, the CASE expression returns null.

The following example uses a selector CASE expression to assign the department name to a variable based on the department number:

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        v_dname :=
            CASE v_deptno
                WHEN 10 THEN 'Accounting'
                WHEN 20 THEN 'Research'
                WHEN 30 THEN 'Sales'
                WHEN 40 THEN 'Operations'
                ELSE 'unknown'
            END;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            ' ' || v_deptno || '     ' || v_dname);
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following output is generated from this program:

```
EMPNO    ENAME     DEPTNO    DNAME
-----    -------   ------    ----------
7369     SMITH      20       Research
7499     ALLEN      30       Sales
7521     WARD       30       Sales
7566     JONES      20       Research
7654     MARTIN     30       Sales
7698     BLAKE      30       Sales
7782     CLARK      10       Accounting
7788     SCOTT      20       Research
7839     KING       10       Accounting
7844     TURNER     30       Sales
7876     ADAMS      20       Research
7900     JAMES      30       Sales
7902     FORD       20       Research
7934     MILLER     10       Accounting
```

### Searched CASE expression

A searched CASE expression uses one or more Boolean expressions to determine the resulting value to return.

```
CASE WHEN boolean-expression THEN
    result
[ WHEN boolean-expression THEN
    result
 [ WHEN boolean-expression THEN
    result ] ...]
[ ELSE
    result ]
END;
```

boolean-expression is evaluated in the order in which it appears within the CASE expression. result is an expression that is type-compatible in the context where the CASE expression is used. When the first boolean-expression that evaluates to TRUE is encountere d, result in the corresponding THEN clause is returned as the value of the CASE expression . If none of boolean-expression evaluates to TRUE, result following ELSE is returned. If no ELSE is specified, the CASE expression returns null.

The following example uses a searched CASE expression to assign the department name to a variable based on the department number:

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        v_dname :=
            CASE
                WHEN v_deptno = 10 THEN 'Accounting'
                WHEN v_deptno = 20 THEN 'Research'
                WHEN v_deptno = 30 THEN 'Sales'
                WHEN v_deptno = 40 THEN 'Operations'
                ELSE 'unknown'
            END;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || v_dname);
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following output is generated from this program:

```
EMPNO    ENAME     DEPTNO    DNAME
-----    -------   ------    ----------
7369     SMITH      20       Research
7499     ALLEN      30       Sales
7521     WARD       30       Sales
7566     JONES      20       Research
7654     MARTIN     30       Sales
7698     BLAKE      30       Sales
7782     CLARK      10       Accounting
7788     SCOTT      20       Research
7839     KING       10       Accounting
7844     TURNER     30       Sales
7876     ADAMS      20       Research
7900     JAMES      30       Sales
7902     FORD       20       Research
7934     MILLER     10       Accounting
```

## 7.9.4 CASE statement

The CASE statement executes a set of one or more statements when a specified search condition is TRUE. The CASE statement is a standalone statement in itself while the previously discussed CASE expression must appear as part of an expression.

The CASE statement is available in two formats. One is called a searched CASE and the other uses a selector.

### Selector CASE statement

The selector CASE statement attempts to match an expression called selector to the expression specified in one or more WHEN clauses. When a match is found, one or more corresponding statements are executed.

```
  CASE selector-expression
  WHEN match-expression THEN
    statements
[ WHEN match-expression THEN
    statements
[ WHEN match-expression THEN
    statements ] ...]
[ ELSE
    statements ]
  END CASE;
```

selector-expression returns a value that is type-compatible with each match-expression. match-expression is evaluated in the order in which it appears within the CASE statement . statements indicates one or more SPL statements, each of which is terminated by a semicolon. When the value of selector-expression equals the first match-expression, the statements in the corresponding THEN clause are executed and control continues following the END CASE keywords. If no match is found, the statements following ELSE are executed. If no match is found and no ELSE clause exists, an exception is thrown.

The following example uses a selector CASE statement to assign a department name and location to a variable based on the department number:

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_loc           dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE v_deptno
            WHEN 10 THEN v_dname := 'Accounting';
                         v_loc   := 'New York';
            WHEN 20 THEN v_dname := 'Research';
                         v_loc   := 'Dallas';
            WHEN 30 THEN v_dname := 'Sales';
                         v_loc   := 'Chicago';
            WHEN 40 THEN v_dname := 'Operations';
                         v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
```

```
                v_loc    := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || RPAD(v_dname, 14) || ' ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following output is generated from this program:

```
EMPNO    ENAME     DEPTNO    DNAME          LOC
-----    -------   ------    ----------     ---------
7369     SMITH       20      Research       Dallas
7499     ALLEN       30      Sales          Chicago
7521     WARD        30      Sales          Chicago
7566     JONES       20      Research       Dallas
7654     MARTIN      30      Sales          Chicago
7698     BLAKE       30      Sales          Chicago
7782     CLARK       10      Accounting     New York
7788     SCOTT       20      Research       Dallas
7839     KING        10      Accounting     New York
7844     TURNER      30      Sales          Chicago
7876     ADAMS       20      Research       Dallas
7900     JAMES       30      Sales          Chicago
7902     FORD        20      Research       Dallas
7934     MILLER      10      Accounting     New York
```

**Searched CASE statement**

A searched CASE statement uses one or more Boolean expressions to determine the resulting set of statements to execute.

```
  CASE WHEN boolean-expression THEN
    statements
[ WHEN boolean-expression THEN
    statements
[ WHEN boolean-expression THEN
    statements ] ...]
[ ELSE
    statements ]
  END CASE;
```

boolean-expression is evaluated in the order in which it appears within the CASE statement . When the first boolean-expression that evaluates to TRUE is encountered, the statements in the corresponding THEN clause are executed and control continues following the END CASE keywords. If none of boolean-expression evaluates to TRUE, the statements following ELSE are executed. If none of boolean-expression evaluates to TRUE and no ELSE clause exists, an exception is thrown.

The following example uses a searched CASE statement to assign a department name and location to a variable based on the department number:

```
DECLARE
```

```
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_loc           dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME      '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE
            WHEN v_deptno = 10 THEN v_dname := 'Accounting';
                                    v_loc   := 'New York';
            WHEN v_deptno = 20 THEN v_dname := 'Research';
                                    v_loc   := 'Dallas';
            WHEN v_deptno = 30 THEN v_dname := 'Sales';
                                    v_loc   := 'Chicago';
            WHEN v_deptno = 40 THEN v_dname := 'Operations';
                                    v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                                    v_loc   := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || RPAD(v_dname, 14) || ' ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following output is generated from this program:

```
EMPNO    ENAME     DEPTNO    DNAME          LOC
-----    -------   ------    ----------     ---------
7369     SMITH       20      Research       Dallas
7499     ALLEN       30      Sales          Chicago
7521     WARD        30      Sales          Chicago
7566     JONES       20      Research       Dallas
7654     MARTIN      30      Sales          Chicago
7698     BLAKE       30      Sales          Chicago
7782     CLARK       10      Accounting     New York
7788     SCOTT       20      Research       Dallas
7839     KING        10      Accounting     New York
7844     TURNER      30      Sales          Chicago
7876     ADAMS       20      Research       Dallas
7900     JAMES       30      Sales          Chicago
7902     FORD        20      Research       Dallas
```

```
7934    MILLER     10     Accounting     New York
```

## 7.9.5 Loops

Using the LOOP, EXIT, CONTINUE, WHILE, and FOR statements, you can arrange for your SPL program to repeat a series of statements.

**LOOP**

```
LOOP
   statements
END LOOP;
```

LOOP defines an unconditional loop that is repeated indefinitely until terminated by an EXIT or a RETURN statement.

**EXIT**

```
EXIT [ WHEN expression ];
```

The innermost loop is terminated and the statement following END LOOP is executed next.

If WHEN is present, loop exit occurs only when the specified condition is TRUE. Otherwise, control passes to the statement after EXIT.

EXIT can be used to cause early exit from all types of loops. It is not limited to use with unconditional loops.

The following simple example shows a loop that iterates ten times and then uses the EXIT statement to terminate:

```
DECLARE
   v_counter      NUMBER(2);
BEGIN
   v_counter := 1;
   LOOP
      EXIT WHEN v_counter > 10;
      DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
      v_counter := v_counter + 1;
   END LOOP;
END;
```

The following output is generated from this program:

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

### CONTINUE

The CONTINUE statement provides a way to proceed with the next iteration of a loop while skipping intervening statements.

When the CONTINUE statement is encountered, the next iteration of the innermost loop is begun, skipping all statements following the CONTINUE statement until the end of the loop. Control is passed back to the loop control expression, if any, and the body of the loop is re-evaluated.

If the WHEN clause is used, the next iteration of the loop is begun only when the specified expression in the WHEN clause evaluates to TRUE. Otherwise, control is passed to the next statement following the CONTINUE statement.

The CONTINUE statement may not be used outside of a loop.

The following example shows a variation of the previous example that uses the CONTINUE statement to skip the display of the odd numbers:

```
DECLARE
    v_counter       NUMBER(2);
BEGIN
    v_counter := 0;
    LOOP
        v_counter := v_counter + 1;
        EXIT WHEN v_counter > 10;
        CONTINUE WHEN MOD(v_counter,2) = 1;
        DBMS_OUTPUT.PUT_LINE('Iteration # ' ‖ v_counter);
    END LOOP;
END;
```

The following output is generated from the above program:

```
Iteration # 2
Iteration # 4
Iteration # 6
Iteration # 8
Iteration # 10
```

### WHILE

```
WHILE expression LOOP
    statements
END LOOP;
```

The WHILE statement repeats a sequence of statements so long as the condition expression evaluates to TRUE. The condition is checked just before each entry to the loop body.

The following example contains the same logic as in the previous example except the WHILE statement is used to take the place of the EXIT statement to determine when to exit the loop.

**Note:**

The conditional expression used to determine when to exit the loop must be altered. The EXIT statement terminates the loop when its conditional expression is true. The WHILE statement terminates (or never begins the loop) when its conditional expression is false.

```
DECLARE
  v_counter     NUMBER(2);
BEGIN
  v_counter := 1;
  WHILE v_counter <= 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
    v_counter := v_counter + 1;
  END LOOP;
END;
```

The same result is generated by this example as in the prior example.

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

**FOR (integer variant)**

```
FOR name IN [REVERSE] expression .. expression LOOP
  statements
END LOOP;
```

This form of FOR creates a loop that iterates over a range of integer values. The name variable is automatically defined as the INTEGER type and exists only inside the loop. The two expressions giving the loop range are evaluated once when they enter the loop. The iteration step is +1 and name begins with the value of expression to the left of .. and terminates once name exceeds the value of expression to the right of ... Therefore, the two expressions take on the following roles: start-value .. end-value.

The optional REVERSE clause specifies that the loop must iterate in reverse order. The first time name passes through the loop, name is set to the value of the right-most expression. The loop terminates when name is less than the left-most expression.

The following example simplifies the WHILE loop example even further by using a FOR loop that iterates from 1 to 10:

```
BEGIN
  FOR i IN 1 .. 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

The following output is generated from the FOR statement:

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

If the start value is greater than the end value, the loop body is not executed. No error is raised as shown by the following example:

```
BEGIN
  FOR i IN 10 .. 1 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

There is no output from this example because the loop body is never executed.

**Note:**

SPL also supports CURSOR FOR loops.

## 7.9.6 Exception handling

By default, any error occurring in an SPL program aborts execution of the program. You can trap errors and recover from them by using a BEGIN block that has an EXCEPTION section. The corresponding syntax is an extension of the normal syntax for a BEGIN block:

```
[ DECLARE
    declarations ]
  BEGIN
    statements
  EXCEPTION
    WHEN condition [ OR condition ]... THEN
      handler_statements
  [ WHEN condition [ OR condition ]... THEN
      handler_statements ]...
```

```
END;
```

If no error occurs, this form of block simply executes all the statements, and then control passes to the next statement after END. If an error occurs within the statements, further processing of the statements is abandoned, and control passes to the EXCEPTION list. The list is searched for the first condition matching the error that occurred. If a match is found, the corresponding handler_statements are executed, and then control passes to the next statement after END. If no match is found, the error propagates out as though the EXCEPTION clause did not exist. The error can be caught by an enclosing block with EXCEPTION. If no enclosing block exists, the error aborts processing of the subprogram.

The special condition name OTHERS matches every error type. Condition names are not case-sensitive.

If a new error occurs within the selected handler_statements, the error cannot be caught by this EXCEPTION clause, but is propagated out. A surrounding EXCEPTION clause can catch the error.

The following table lists the condition names that may be used.

| Condition name | Description |
|---|---|
| CASE_NOT_FOUND | The application has encountered a situation where none of the cases in a CASE statement evaluates to TRUE and no ELSE condition exists. |
| COLLECTION_IS_NULL | The application has attempted to invoke a collection method on a null collection such as an uninitialized nested table. |
| CURSOR_ALREADY_OPEN | The application has attempted to open a cursor that is already open. |
| DUP_VAL_ON_INDEX | The application has attempted to store a duplicate value that currently exists within a constrained column. |
| INVALID_CURSOR | The application has attempted to access an unopened cursor. |
| INVALID_NUMBER | The application has encountered a data exception ( equivalent to SQLSTATE class code 22). INVALID_NUMBER is an alias for VALUE_ERROR. |
| NO_DATA_FOUND | No rows satisfy the selection criteria. |
| OTHERS | The application has encountered an exception that has not been caught by a prior condition in the exception section. |

| Condition name | Description |
|---|---|
| SUBSCRIPT_BEYOND_COUNT | The application has attempted to reference a subscript of a nested table or varray beyond its initialized or extended size. |
| SUBSCRIPT_OUTSIDE_LIMIT | The application has attempted to reference a subscript or extend a varray beyond its maximum size limit. |
| TOO_MANY_ROWS | The application has encountered more than one row that satisfies the selection criteria (where only one row is allowed to be returned). |
| VALUE_ERROR | The application has encountered a data exception ( equivalent to SQLSTATE class code 22). VALUE_ERROR is an alias for INVALID_NUMBER. |
| ZERO_DIVIDE | The application has tried to divide by zero. |
| User-defined Exception | For more information, see User-defined exceptions. |

**Note:**

Condition names INVALID_NUMBER and VALUE_ERROR are not compatible with Oracle databases. For Oracle databases, these condition names are for exceptions resulting only from a failed conversion of a string to a numeric literal. In addition, for Oracle databases, an INVALID_NUMBER exception is applicable only to SQL statements while a VALUE_ERROR exception is applicable only to procedural statements.

## 7.9.7 User-defined exceptions

Any number of errors (referred to in PL/SQL as exceptions) can occur during program execution. When an exception is thrown, normal execution of the program stops, and control of the program transfers to the error-handling portion of the program. An exception may be a predefined error that is generated by the server, or may be a logical error that raises a user-defined exception.

User-defined exceptions are never raised by the server. Instead, they are raised explicitly by a RAISE statement. A user-defined exception is raised when a developer-defined logical rule is broken. A common example of a logical rule being broken occurs when a check is presented against an account with insufficient funds. An attempt to cash a check against an account with insufficient funds will provoke a user-defined exception.

You can define exceptions in functions, procedures, packages, or anonymous blocks. You cannot declare the same exception twice in the same block, but you can declare the same exception in two different blocks.

Before implementing a user-defined exception, you must declare the exception in the declaration section of a function, a procedure, a package, or an anonymous block. You can then raise the exception by using the RAISE statement:

```
DECLARE
   exception_name EXCEPTION;

BEGIN
   ...
   RAISE exception_name;
   ...
END;
```

exception_name is the name of the exception.

Unhandled exceptions propagate back through the call stack. If the exception remains unhandled, the exception is eventually reported to the client application.

User-defined exceptions declared in a block are considered to be local to that block and global to any nested blocks within the block. To reference an exception that resides in an outer block, you must assign a label to the outer block, and then preface the name of the exception with the block name:

```
block_name.exception_name
```

Conversely, outer blocks cannot reference exceptions declared in nested blocks.

The scope of a declaration is limited to the block in which it is declared unless it is created in a package, and when referenced, qualified by the package name. For example, to raise an exception named out_of_stock that resides in a package named inventory_control, a program must raise an error named:

```
inventory_control.out_of_stock
```

The following example demonstrates declaring a user-defined exception in a package. The user-defined exception does not require a package qualifier when it is raised in check_balance, because it resides in the same package as the exception:

```
CREATE OR REPLACE PACKAGE ar AS
  overdrawn EXCEPTION;
  PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
```

```
    PROCEDURE check_balance(p_balance NUMBER, p_amount  NUMBER)
    IS
    BEGIN
       IF (p_amount > p_balance) THEN
         RAISE overdrawn;
       END IF;
     END;
```

The following procedure (purchase) calls the check_balance procedure. If p_amount is greater than p_balance, check_balance raises an exception, and purchase catches the ar. overdrawn exception. purchase must refer to the exception with a package qualified name ( ar.overdrawn) because purchase is not defined within the ar package.

```
CREATE PROCEDURE purchase(customerID INT, amount NUMERIC)
AS
  BEGIN
     ar.check_ balance(getcustomerbalance(customerid), amount);
       record_purchase(customerid, amount);
  EXCEPTION
     WHEN ar.overdrawn THEN
       raise_credit_limit(customerid, amount*1.5);
  END;
```

When ar.check_balance raises an exception, execution jumps to the exception handler defined in purchase:

```
EXCEPTION
     WHEN ar.overdrawn THEN
       raise_credit_limit(customerid, amount*1.5);
```

The exception handler raises the credit limit of the customer and ends. When the exception handler ends, execution resumes with the statement that follows ar.check_balance.

## 7.9.8 PRAGMA EXCEPTION_INIT

PRAGMA EXCEPTION_INIT associates a user-defined error code with an exception. A PRAGMA EXCEPTION_INIT declaration may be included in any block, sub-block, or package . You can only assign an error code to an exception (using PRAGMA EXCEPTION_INIT) after declaring the exception. The format of a PRAGMA EXCEPTION_INIT declaration is as follows:

```
PRAGMA EXCEPTION_INIT(exception_name,
                    {exception_number | exception_code})
```

where:

- exception_name is the name of the associated exception.
- exception_number is a user-defined error code associated with the pragma. If you specify an unmapped exception_number value, the server will return a warning.

- exception_code is the name of a predefined exception. For a complete list of valid exceptions, see the PostgreSQL core documentation available at: https://www.postgresql.org/docs/11/static/errcodes-appendix.html.

User-defined exceptions included an example that demonstrates how to declare a user-defined exception in a package. The following example uses the same basic structure, but adds a PRAGMA EXCEPTION_INIT declaration:

```
CREATE OR REPLACE PACKAGE ar AS
  overdrawn EXCEPTION;
  PRAGMA EXCEPTION_INIT (overdrawn, -20100);
  PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
   PROCEDURE check_balance(p_balance NUMBER, p_amount  NUMBER)
   IS
   BEGIN
       IF (p_amount > p_balance) THEN
         RAISE overdrawn;
       END IF;
    END;
```

The following procedure (purchase) calls the check_balance procedure. If p_amount is greater than p_balance, check_balance raises an exception, and purchase catches the ar.overdrawn exception.

```
CREATE PROCEDURE purchase(customerID int, amount NUMERIC)
AS
  BEGIN
     ar.check_ balance(getcustomerbalance(customerid), amount);
       record_purchase(customerid, amount);
  EXCEPTION
     WHEN ar.overdrawn THEN
      DBMS_OUTPUT.PUT_LINE ('This account is overdrawn.') ;
      DBMS_OUTPUT.PUT_LINE ('SQLCode :'||SQLCODE||' '||SQLERRM );
END;
```

When ar.check_balance raises an exception, execution jumps to the exception handler defined in purchase.

```
EXCEPTION
     WHEN ar.overdrawn THEN
      DBMS_OUTPUT.PUT_LINE ('This account is overdrawn.') ;
      DBMS_OUTPUT.PUT_LINE ('SQLCode :'||SQLCODE||' '||SQLERRM );
```

The exception handler returns an error message, followed by SQLCODE information:

```
This account is overdrawn.
```

```
SQLCODE: -20100 User-Defined Exception
```

The following example demonstrates how to use a predefined exception. The code creates a more meaningful name for the no_data_found exception. If the given customer does not exist, the code catches the exception, calls DBMS_OUTPUT.PUT_LINE to report the error, and then re-raises the original exception:

```
CREATE OR REPLACE PACKAGE ar AS
  overdrawn EXCEPTION;
  PRAGMA EXCEPTION_INIT (unknown_customer, no_data_found);
  PROCEDURE check_balance(p_customer_id NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
   PROCEDURE check_balance(p_customer_id NUMBER)
   IS
   DECLARE
     v_balance NUMBER;
   BEGIN
     SELECT balance INTO v_balance FROM customer
       WHERE cust_id = p_customer_id;
   EXCEPTION WHEN unknown_customer THEN
     DBMS_OUTPUT.PUT_LINE('invalid customer id');
     RAISE;
   END;
```

## 7.9.9 RAISE_APPLICATION_ERROR

The RAISE_APPLICATION_ERROR procedure allows a developer to intentionally abort processing within an SPL program from which the procedure is called by causing an exception. The exception is handled in the same manner as described in the topic of Exception handling. In addition, the RAISE_APPLICATION_ERROR procedure makes a user-defined code and error message available to the program which can then be used to identify the exception.

```
RAISE_APPLICATION_ERROR(error_number, message);
```

where:

- error_number is an integer value or expression that is returned in a variable named SQLCODE when the procedure is executed. error_number must be a value between -20000 and -20999.
- message is a string literal or expression that is returned in a variable named SQLERRM.

The following example uses the RAISE_APPLICATION_ERROR procedure to display a different code and message depending upon the information missing from an employee record:

```
CREATE OR REPLACE PROCEDURE verify_emp (
    p_empno        NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_mgr           emp.mgr%TYPE;
    v_hiredate      emp.hiredate%TYPE;
BEGIN
    SELECT ename, job, mgr, hiredate
        INTO v_ename, v_job, v_mgr, v_hiredate FROM emp
        WHERE empno = p_empno;
    IF v_ename IS NULL THEN
        RAISE_APPLICATION_ERROR(-20010, 'No name for ' || p_empno);
    END IF;
    IF v_job IS NULL THEN
        RAISE_APPLICATION_ERROR(-20020, 'No job for' || p_empno);
    END IF;
    IF v_mgr IS NULL THEN
        RAISE_APPLICATION_ERROR(-20030, 'No manager for ' || p_empno);
    END IF;
    IF v_hiredate IS NULL THEN
        RAISE_APPLICATION_ERROR(-20040, 'No hire date for ' || p_empno);
    END IF;
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
        ' validated without errors');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
END;
```

The following output is generated in a case where the manager number is missing from an employee record:

```
EXEC verify_emp(7839);

SQLCODE: -20030
SQLERRM: EDB-20030: No manager for 7839
```

## 7.10 IF statements

### 7.10.1 IF-THEN

```
IF boolean-expression THEN
  statements
END IF;
```

IF-THEN statements are the simplest form of IF. The statements between THEN and END IF will be executed if the condition is TRUE. Otherwise, they are skipped.

In the following example, an IF-THEN statement is used to test and display employees who have a commission.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comm          emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
            TO_CHAR(v_comm,'$99999.99'));
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following output is generated from this program:

```
EMPNO    COMM
-----    -------
7499     $300.00
7521     $500.00
7654    $1400.00
```

## 7.10.2 IF-THEN-ELSE

```
IF boolean-expression THEN
  statements
ELSE
  statements
END IF;
```

IF-THEN-ELSE statements can be added to IF-THEN to allow you to specify an alternative set of statements that must be executed if the condition evaluates to false.

The previous example is modified so an IF-THEN-ELSE statement is used to display the Non-commission text if the employee does not get a commission.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comm          emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
            TO_CHAR(v_comm,'$99999.99'));
        ELSE
            DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || 'Non-commission');
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following output is generated from this program:

```
EMPNO    COMM
-----    -------
7369     Non-commission
7499  $   300.00
7521  $   500.00
7566     Non-commission
7654  $  1400.00
7698     Non-commission
7782     Non-commission
7788     Non-commission
7839     Non-commission
7844     Non-commission
7876     Non-commission
7900     Non-commission
7902     Non-commission
7934     Non-commission
```

## 7.10.3 IF-THEN-ELSE IF

IF statements can be nested so that alternative IF statements can be invoked after it is determined whether the conditional of an outer IF statement is TRUE or FALSE.

In the following example, the outer IF-THEN-ELSE statement tests whether an employee has a commission. The inner IF-THEN-ELSE statements then test whether the total compensation of the employee exceeds or is less than the company average.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_avg           NUMBER(7,2);
    CURSOR emp_cursor IS SELECT empno, sal, comm FROM emp;
BEGIN
--
--  Calculate the average yearly compensation in the company
--
    SELECT AVG((sal + NVL(comm,0)) * 24) INTO v_avg FROM emp;
    DBMS_OUTPUT.PUT_LINE('Average Yearly Compensation: ' ||
        TO_CHAR(v_avg,'$999,999.99'));
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    YEARLY COMP');
    DBMS_OUTPUT.PUT_LINE('-----    -----------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_sal, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
```

```
--
--  Test if the employee's compensation with commission exceeds the average
--
         IF (v_sal + v_comm) * 24 > v_avg THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
               TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
               ' Exceeds Average');
         ELSE
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
               TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
               ' Below Average');
         END IF;
      ELSE
--
--  Test if the employee's compensation without commission exceeds the average
--
         IF v_sal * 24 > v_avg THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
               TO_CHAR(v_sal * 24,'$999,999.99') || ' Exceeds Average');
         ELSE
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
               TO_CHAR(v_sal * 24,'$999,999.99') || ' Below Average');
         END IF;
      END IF;
   END LOOP;
   CLOSE emp_cursor;
END;
```

**Note:**

To significantly simplify the logic in this program, you can calculate the yearly compensation of the employee by using the NVL function within the SELECT statement of the cursor declaration. However, the purpose of this example is to demonstrate how IF statements can be used.

The following output is generated from this program:

```
Average Yearly Compensation: $  53,528.57
EMPNO    YEARLY COMP
-----    -----------
7369  $  19,200.00 Below Average
7499  $  45,600.00 Below Average
7521  $  42,000.00 Below Average
7566  $  71,400.00 Exceeds Average
7654  $  63,600.00 Exceeds Average
7698  $  68,400.00 Exceeds Average
7782  $  58,800.00 Exceeds Average
7788  $  72,000.00 Exceeds Average
7839  $ 120,000.00 Exceeds Average
7844  $  36,000.00 Below Average
7876  $  26,400.00 Below Average
7900  $  22,800.00 Below Average
7902  $  72,000.00 Exceeds Average
```

```
7934  $  31,200.00 Below Average
```

When you use this form, you are actually nesting an IF statement inside the ELSE part of an outer IF statement. Therefore, you need one END IF statement for each nested IF statement and one for the parent IF-ELSE statement.

### 7.10.4 IF-THEN-ELSIF-ELSE

```
  IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements ] ...]
[ ELSE
    statements ]
  END IF;
```

IF-THEN-ELSIF-ELSE provides a method of checking many alternatives in one statement. Formally it is equivalent to nested IF-THEN-ELSE-IF-THEN statements, but only one END IF is needed.

The following example uses an IF-THEN-ELSIF-ELSE statement to count the number of employees by compensation range of USD 25,000.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comp          NUMBER(8,2);
    v_lt_25K        SMALLINT := 0;
    v_25K_50K       SMALLINT := 0;
    v_50K_75K       SMALLINT := 0;
    v_75K_100K      SMALLINT := 0;
    v_ge_100K       SMALLINT := 0;
    CURSOR emp_cursor IS SELECT empno, (sal + NVL(comm,0)) * 24 FROM emp;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_empno, v_comp;
        EXIT WHEN emp_cursor%NOTFOUND;
        IF v_comp < 25000 THEN
            v_lt_25K := v_lt_25K + 1;
        ELSIF v_comp < 50000 THEN
            v_25K_50K := v_25K_50K + 1;
        ELSIF v_comp < 75000 THEN
            v_50K_75K := v_50K_75K + 1;
        ELSIF v_comp < 100000 THEN
            v_75K_100K := v_75K_100K + 1;
        ELSE
            v_ge_100K := v_ge_100K + 1;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
    DBMS_OUTPUT.PUT_LINE('Number of employees by yearly compensation');
    DBMS_OUTPUT.PUT_LINE('Less than 25,000 : ' || v_lt_25K);
    DBMS_OUTPUT.PUT_LINE('25,000 - 49,9999 : ' || v_25K_50K);
    DBMS_OUTPUT.PUT_LINE('50,000 - 74,9999 : ' || v_50K_75K);
    DBMS_OUTPUT.PUT_LINE('75,000 - 99,9999 : ' || v_75K_100K);
    DBMS_OUTPUT.PUT_LINE('100,000 and over : ' || v_ge_100K);
END;
```

The following output is generated from this program:

```
Number of employees by yearly compensation
Less than 25,000 : 2
25,000 - 49,9999 : 5
50,000 - 74,9999 : 6
75,000 - 99,9999 : 0
100,000 and over : 1
```

# 7.11 Transaction control

## 7.11.1 Overview

Under some circumstances, it is desired that all updates to a database are to occur successfully, or none is to occur if any error occurs. A set of database updates that are to all occur successfully as a single unit, or are not to occur, is said to be a transaction.

A common example in banking is a funds transfer between two accounts. The two parts of the transaction are the withdrawal of funds from one account and the deposit of the funds in another account. Both parts of this transaction must occur. Otherwise, books of the bank will be out of balance. The deposit and withdrawal are one transaction.

An SPL application that uses a style of transaction control compatible with Oracle databases can be created if the following conditions are met:

- The edb_stmt_level_tx parameter must be set to TRUE. This prevents the action of unconditionally rolling back all database updates within the BEGIN/END block if any exception occurs.
- The application must not be running in autocommit mode. If the autocommit mode is on, each successful database update is immediately committed and cannot be undone. The manner in which the autocommit mode is turned on or off is application dependent.

A transaction begins when the first SQL statement is encountered in the SPL program. All subsequent SQL statements are included as part of that transaction. The transaction ends when one of the following conditions occurs:

- An unhandled exception occurs. In this case, the effects of all database updates made during the transaction are rolled back and the transaction is aborted.

- A COMMIT statement is encountered. In this case, the effects of all database updates made during the transaction become permanent.
- A ROLLBACK statement is encountered. In this case, the effects of all database updates made during the transaction are rolled back and the transaction is aborted. If a new SQL statement is encountered, a new transaction begins.
- Control returns to the calling application such as Java and PostgreSQL. In this case, the action of the application determines whether the transaction is committed or rolled back unless the transaction is within a block in which PRAGMA AUTONOMOUS_TRANSACTION has been declared in which case the commitment or rollback of the transaction occurs independently of the calling program.

**Note:**

Unlike Oracle, DDL statements such as CREATE TABLE do not implicitly occur within their own transaction. Therefore, DDL statements do not automatically cause an immediate database commit as in Oracle, and DDL statements may be rolled back just like DML statements.

A transaction may span one or more BEGIN/END blocks, or a single BEGIN/END block may contain one or more transactions.

The following topics discuss the COMMIT and ROLLBACK statements in more detail.

## 7.11.2 COMMIT

The COMMIT statement makes all database updates made during the current transaction permanent, and ends the current transaction.

```
COMMIT [ WORK ];
```

The COMMIT statement may be used within anonymous blocks, stored procedures, or functions. Within an SPL program, it may appear in the executable section and the exception section.

In the following example, the third INSERT statement in the anonymous block results in an error. The effect of the first two INSERT statements is retained as shown by the first SELECT statement. Even after a ROLLBACK statement is issued, the two rows remain in the table as shown by the second SELECT statement. This verifies that the two rows were indeed committed.

**Note:**

You can set the edb_stmt_level_tx configuration parameter shown in the following example for the entire database by using the ALTER DATABASE statement. You can also set edb_stmt_level_tx for the entire database server by changing it in the postgresql.conf file.

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

BEGIN
    INSERT INTO dept VALUES (50, 'FINANCE', 'DALLAS');
    INSERT INTO dept VALUES (60, 'MARKETING', 'CHICAGO');
    COMMIT;
    INSERT INTO dept VALUES (70, 'HUMAN RESOURCES', 'CHICAGO');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

SQLERRM: value too long for type character varying(14)
SQLCODE: 22001

SELECT * FROM dept;

deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | FINANCE    | DALLAS
     60 | MARKETING  | CHICAGO
(6 rows)

ROLLBACK;

SELECT * FROM dept;

deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | FINANCE    | DALLAS
     60 | MARKETING  | CHICAGO
(6 rows)
```

## 7.11.3 ROLLBACK

The ROLLBACK statement undoes all database updates made during the current transaction, and ends the current transaction.

```
ROLLBACK [ WORK ];
```

The ROLLBACK statement may be used within anonymous blocks, stored procedures, or functions. Within an SPL program, it may appear in the executable section and the exception section.

In the following example, the exception section contains a ROLLBACK statement. Even though the first two INSERT statements are executed successfully, the third one results in an exception that causes the rollback of all INSERT statements in the anonymous block.

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

BEGIN
    INSERT INTO dept VALUES (50, 'FINANCE', 'DALLAS');
    INSERT INTO dept VALUES (60, 'MARKETING', 'CHICAGO');
    INSERT INTO dept VALUES (70, 'HUMAN RESOURCES', 'CHICAGO');
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

SQLERRM: value too long for type character varying(14)
SQLCODE: 22001

SELECT * FROM dept;

deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)
```

The following example uses both COMMIT and ROLLBACK. First, the following stored procedure which inserts a new employee is created.

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

CREATE OR REPLACE PROCEDURE emp_insert (
    p_empno         IN emp.empno%TYPE,
    p_ename         IN emp.ename%TYPE,
    p_job           IN emp.job%TYPE,
    p_mgr           IN emp.mgr%TYPE,
    p_hiredate      IN emp.hiredate%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm          IN emp.comm%TYPE,
    p_deptno        IN emp.deptno%TYPE
)
IS
BEGIN
    INSERT INTO emp VALUES (
        p_empno,
        p_ename,
        p_job,
        p_mgr,
        p_hiredate,
        p_sal,
        p_comm,
        p_deptno);

    DBMS_OUTPUT.PUT_LINE('Added employee...') ;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || p_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || p_job);
    DBMS_OUTPUT.PUT_LINE('Manager    : ' || p_mgr);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || p_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || p_sal);
    DBMS_OUTPUT.PUT_LINE('Commission : ' || p_comm);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || p_deptno);
    DBMS_OUTPUT.PUT_LINE('----------------------');
END;
```

Note that this procedure has no exception section so any error that may occur is propagated up to the calling program.

The following anonymous block is run. Note the use of the COMMIT statement after all calls to the emp_insert procedure and the ROLLBACK statement in the exception section.

```
BEGIN
    emp_insert(9601,'FARRELL','ANALYST',7902,'03-MAR-08',5000,NULL,40);
    emp_insert(9602,'TYLER','ANALYST',7900,'25-JAN-08',4800,NULL,40);
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('An error occurred - roll back inserts');
        ROLLBACK;
END;

Added employee...
Employee # : 9601
Name       : FARRELL
Job        : ANALYST
Manager    : 7902
Hire Date  : 03-MAR-08 00:00:00
Salary     : 5000
Commission :
Dept #     : 40
----------------------
Added employee...
Employee # : 9602
Name       : TYLER
Job        : ANALYST
Manager    : 7900
Hire Date  : 25-JAN-08 00:00:00
Salary     : 4800
Commission :
Dept #     : 40
----------------------
```

The following SELECT statement shows that employees Farrell and Tyler were added.

```
SELECT * FROM emp WHERE empno > 9600;

empno |  ename  |   job   | mgr  |      hiredate      |   sal   | comm | deptno
-------+---------+---------+------+--------------------+---------+------+--------
  9601 | FARRELL | ANALYST | 7902 | 03-MAR-08 00:00:00 | 5000.00 |      |     40
  9602 | TYLER   | ANALYST | 7900 | 25-JAN-08 00:00:00 | 4800.00 |      |     40
```

```
(2 rows)
```

Execute the following anonymous block:

```
BEGIN
    emp_insert(9603,'HARRISON','SALESMAN',7902,'13-DEC-07',5000,3000,20);
    emp_insert(9604,'JARVIS','SALESMAN',7902,'05-MAY-08',4800,4100,11);
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' ‖ SQLERRM);
        DBMS_OUTPUT.PUT_LINE('An error occurred - roll back inserts');
        ROLLBACK;
END;

Added employee...
Employee # : 9603
Name       : HARRISON
Job        : SALESMAN
Manager    : 7902
Hire Date  : 13-DEC-07 00:00:00
Salary     : 5000
Commission : 3000
Dept #     : 20
----------------------
SQLERRM: insert or update on table "emp" violates foreign key constraint "emp_ref_de
pt_fk"
An error occurred - roll back inserts
```

A SELECT statement run against the table yields the following output:

```
SELECT * FROM emp WHERE empno > 9600;

empno |  ename  |   job   | mgr  |      hiredate      |   sal   | comm | deptno
-------+---------+---------+------+--------------------+---------+------+--------
  9601 | FARRELL | ANALYST | 7902 | 03-MAR-08 00:00:00 | 5000.00 |      |     40
  9602 | TYLER   | ANALYST | 7900 | 25-JAN-08 00:00:00 | 4800.00 |      |     40
(2 rows)
```

The ROLLBACK statement in the exception section undoes the insert of employee Harrison . Note that employees Farrell and Tyler are still in the table as their inserts were made permanent by the COMMIT statement in the first anonymous block.

**Note:**

Executing a COMMIT or ROLLBACK statement in a PL/pgSQL procedure will throw an error if an Oracle-style SPL procedure exists on the runtime stack.

## 7.11.4 PRAGMA AUTONOMOUS_TRANSACTION

An SPL program is declared as an autonomous transaction when the following directive is specified in the declaration section of the SPL block:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

An autonomous transaction is an independent transaction started by a calling program. A commit or rollback of SQL statements within the autonomous transaction has no effect on the commit or rollback in any transaction of the calling program. A commit or rollback in the calling program has no effect on the commit or rollback of SQL statements in the autonomous transaction.

The following SPL programs can include PRAGMA AUTONOMOUS_TRANSACTION:

- Standalone procedures and functions
- Anonymous blocks
- Procedures and functions declared as subprograms within packages and other calling procedures, functions, and anonymous blocks
- Triggers
- Object type methods

The following issues and restrictions are related to autonomous transactions:

- Each autonomous transaction consumes a connection slot as long as it is in progress. In some cases, this may mean that the max_connections parameter in the postgresql.conf file needs to be raised.
- In most respects, an autonomous transaction behaves exactly as if it was a completely separate session, but GUCs (that is, settings established with SET) are a deliberate exception. Autonomous transactions absorb the surrounding values and can propagate values they commit to the outer transaction.
- Autonomous transactions can be nested. A maximum of 16 levels of autonomous transactions are allowed within a single session.
- Parallel query is not supported within autonomous transactions.
- The implementation of PolarDB databases compatible with Oracle of autonomous transactions is not entirely compatible with Oracle databases in that the autonomous transactions for PolarDB databases compatible with Oracle do not produce an error if an uncommitted transaction exists at the end of an SPL block.

The following set of examples illustrate the usage of autonomous transactions. This first set of scenarios show the default behavior when no autonomous transactions exist.

Before each scenario, the dept table is reset to the following initial values:

```
SELECT * FROM dept;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)
```

**Scenario 1a - No autonomous transactions with only a final COMMIT statement**

This first set of scenarios show the insertion of three rows starting just after the initial BEGIN statement of the transaction, then from within an anonymous block within the starting transaction, and finally from a stored procedure executed from within the anonymous block .

The following example shows the stored procedure:

```
CREATE OR REPLACE PROCEDURE insert_dept_70 IS
BEGIN
    INSERT INTO dept VALUES (70,'MARKETING','LOS ANGELES');
END;
```

The following example shows the PostgreSQL session:

```
BEGIN;
INSERT INTO dept VALUES (50,'HR','DENVER');
BEGIN
    INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
    insert_dept_70;
END;
COMMIT;
```

After the final commit, all three rows are inserted:

```
SELECT * FROM dept ORDER BY 1;

 deptno |   dname    |     loc
--------+------------+-------------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | HR         | DENVER
     60 | FINANCE    | CHICAGO
     70 | MARKETING  | LOS ANGELES
```

```
(7 rows)
```

### Scenario 1b - No autonomous transactions, but a final ROLLBACK statement

The next scenario shows that a final ROLLBACK statement after all inserts results in the rollback of all three insertions:

```
BEGIN;
INSERT INTO dept VALUES (50,'HR','DENVER');
BEGIN
   INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
   insert_dept_70;
END;
ROLLBACK;

SELECT * FROM dept ORDER BY 1;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)
```

### Scenario 1c - No autonomous transactions, but anonymous block ROLLBACK

A ROLLBACK statement given at the end of the anonymous block also eliminates all three prior insertions:

```
BEGIN;
INSERT INTO dept VALUES (50,'HR','DENVER');
BEGIN
   INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
   insert_dept_70;
   ROLLBACK;
END;
COMMIT;

SELECT * FROM dept ORDER BY 1;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)
```

The next set of scenarios shows the effect of using autonomous transactions with PRAGMA AUTONOMOUS_TRANSACTION in various locations.

### Scenario 2a - Autonomous transaction of anonymous block with COMMIT

The procedure remains as initially created:

```
CREATE OR REPLACE PROCEDURE insert_dept_70 IS
```

```
BEGIN
    INSERT INTO dept VALUES (70,'MARKETING','LOS ANGELES');
END;
```

PRAGMA AUTONOMOUS_TRANSACTION is given with the anonymous block along with the COMMIT statement at the end of the anonymous block.

```
BEGIN;
INSERT INTO dept VALUES (50,'HR','DENVER');
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
    insert_dept_70;
    COMMIT;
END;
ROLLBACK;
```

After the ROLLBACK statement at the end of the transaction, only the first row insertion at the very beginning of the transaction is discarded. The other two row insertions within the anonymous block with PRAGMA AUTONOMOUS_TRANSACTION have been independently committed.

```
SELECT * FROM dept ORDER BY 1;

 deptno |   dname    |     loc
--------+------------+-------------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     60 | FINANCE    | CHICAGO
     70 | MARKETING  | LOS ANGELES
(6 rows)
```

**Scenario 2b – Autonomous transaction anonymous block with COMMIT including procedure with ROLLBACK, but not an autonomous transaction procedure**

The procedure has the ROLLBACK statement at the end. However, note that PRAGMA ANONYMOUS_TRANSACTION is not included in this procedure.

```
CREATE OR REPLACE PROCEDURE insert_dept_70 IS
BEGIN
    INSERT INTO dept VALUES (70,'MARKETING','LOS ANGELES');
    ROLLBACK;
END;
```

The rollback within the procedure removes the two rows inserted within the anonymous block (deptno 60 and 70) before the final COMMIT statement within the anonymous block.

```
BEGIN;
INSERT INTO dept VALUES (50,'HR','DENVER');
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
```

```
BEGIN
    INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
    insert_dept_70;
    COMMIT;
END;
COMMIT;
```

After the final commit at the end of the transaction, the only row inserted is the first one from the beginning of the transaction. Because the anonymous block is an autonomous transaction, the rollback within the enclosed procedure has no effect on the insertion that occurs before the anonymous block is executed.

```
SELECT * FROM dept ORDER by 1;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | HR         | DENVER
(5 rows)
```

**Scenario 2c – Autonomous transaction anonymous block with COMMIT including procedure with ROLLBACK that is also an autonomous transaction procedure**

The procedure with the ROLLBACK statement at the end also has PRAGMA ANONYMOUS_TRANSACTION included. This isolates the effect of the ROLLBACK statement within the procedure.

```
CREATE OR REPLACE PROCEDURE insert_dept_70 IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO dept VALUES (70,'MARKETING','LOS ANGELES');
    ROLLBACK;
END;
```

The rollback within the procedure removes the row inserted by the procedure, but not the other row inserted within the anonymous block.

```
BEGIN;
INSERT INTO dept VALUES (50,'HR','DENVER');
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
    insert_dept_70;
    COMMIT;
END;
```

```
COMMIT;
```

After the final commit at the end of the transaction, the row inserted is the first one from the beginning of the transaction as well as the row inserted at the beginning of the anonymous block. The only insertion rolled back is the one within the procedure.

```
SELECT * FROM dept ORDER by 1;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | HR         | DENVER
     60 | FINANCE    | CHICAGO
(6 rows)
```

The following sections show examples of PRAGMA AUTONOMOUS_TRANSACTION in a couple of other SPL program types.

### Autonomous transaction trigger

The following example shows the effect of declaring a trigger with PRAGMA AUTONOMOUS _TRANSACTION.

The following table is created to log changes to the emp table:

```
CREATE TABLE empauditlog (
    audit_date      DATE,
    audit_user      VARCHAR2(20),
    audit_desc      VARCHAR2(20)
);
```

The following example shows the trigger attached to the emp table that inserts these changes into the empauditlog table. Note the inclusion of PRAGMA AUTONOMOUS _TRANSACTION in the declaration section.

```
CREATE OR REPLACE TRIGGER emp_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
    v_action        VARCHAR2(20);
BEGIN
    IF INSERTING THEN
        v_action := 'Added employee(s)';
    ELSIF UPDATING THEN
        v_action := 'Updated employee(s)';
    ELSIF DELETING THEN
        v_action := 'Deleted employee(s)';
    END IF;
    INSERT INTO empauditlog VALUES (SYSDATE, USER,
        v_action);
```

```
END;
```

The following two inserts are made into the emp table within a transaction started by the BEGIN statement:

```
BEGIN;
INSERT INTO emp VALUES (9001,'SMITH','ANALYST',7782,SYSDATE,NULL,NULL,10);
INSERT INTO emp VALUES (9002,'JONES','CLERK',7782,SYSDATE,NULL,NULL,10);
```

The following example shows the two new rows in the emp table as well as the two entries in the empauditlog table:

```
SELECT * FROM emp WHERE empno > 9000;

 empno | ename |   job   | mgr  |      hiredate      | sal | comm | deptno
-------+-------+---------+------+--------------------+-----+------+--------
  9001 | SMITH | ANALYST | 7782 | 23-AUG-18 07:12:27 |     |      |     10
  9002 | JONES | CLERK   | 7782 | 23-AUG-18 07:12:27 |     |      |     10
(2 rows)

SELECT TO_CHAR(AUDIT_DATE,'DD-MON-YY HH24:MI:SS') AS "audit date",
    audit_user, audit_desc FROM empauditlog ORDER BY 1 ASC;

     audit date     | audit_user  |    audit_desc
--------------------+-------------+-------------------
 23-AUG-18 07:12:27 | enterprisedb | Added employee(s)
 23-AUG-18 07:12:27 | enterprisedb | Added employee(s)
(2 rows)
```

But then the ROLLBACK statement is given during this session. The emp table no longer contains the two rows, but the empauditlog table still contains its two entries because the trigger implicitly performed a commit and PRAGMA AUTONOMOUS_TRANSACTION commits those changes independent from the rollback given in the calling transaction.

```
ROLLBACK;

SELECT * FROM emp WHERE empno > 9000;

 empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
(0 rows)

SELECT TO_CHAR(AUDIT_DATE,'DD-MON-YY HH24:MI:SS') AS "audit date",
    audit_user, audit_desc FROM empauditlog ORDER BY 1 ASC;

     audit date     | audit_user  |    audit_desc
--------------------+-------------+-------------------
 23-AUG-18 07:12:27 | enterprisedb | Added employee(s)
 23-AUG-18 07:12:27 | enterprisedb | Added employee(s)
(2 rows)
```

**Object type methods of autonomous transactions**

The following example shows the effect of declaring an object method with PRAGMA AUTONOMOUS_TRANSACTION.

The following object type and object type body are created. The member procedure within the object type body contains PRAGMA AUTONOMOUS_TRANSACTION in the declaration section along with COMMIT at the end of the procedure.

```
CREATE OR REPLACE TYPE insert_dept_typ AS OBJECT (
    deptno          NUMBER(2),
    dname           VARCHAR2(14),
    loc             VARCHAR2(13),
    MEMBER PROCEDURE insert_dept
);

CREATE OR REPLACE TYPE BODY insert_dept_typ AS
    MEMBER PROCEDURE insert_dept
    IS
        PRAGMA AUTONOMOUS_TRANSACTION;
    BEGIN
        INSERT INTO dept VALUES (SELF.deptno,SELF.dname,SELF.loc);
        COMMIT;
    END;
END;
```

In the following anonymous block, an insert is performed into the dept table, followed by invocation of the insert_dept method of the object, ending with a ROLLBACK statement in the anonymous block.

```
BEGIN;
DECLARE
    v_dept          INSERT_DEPT_TYP :=
                    insert_dept_typ(60,'FINANCE','CHICAGO');
BEGIN
    INSERT INTO dept VALUES (50,'HR','DENVER');
    v_dept.insert_dept;
    ROLLBACK;
END;
```

Because insert_dept has been declared as an autonomous transaction, its insert of department number 60 remains in the table, but the rollback removes the insertion of department 50.

```
SELECT * FROM dept ORDER BY 1;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     60 | FINANCE    | CHICAGO
```

```
(5 rows)
```

## 7.12 Dynamic SQL

Dynamic SQL is a technique that provides the ability to execute SQL statements that are not known until the statements are about to be executed. Up to this point, the SQL statements that have been illustrated in SPL programs have been static SQL - the full statement (with the exception of variables) must be known and coded into the program before the program itself can begin to execute. Therefore, by using dynamic SQL, the executed SQL can change during program runtime.

In addition, dynamic SQL is the only method by which data definition statements, such as CREATE TABLE, can be executed from within an SPL program.

However, note that the runtime performance of dynamic SQL will be slower than static SQL.

The EXECUTE IMMEDIATE statement is used to run SQL statements dynamically:

```
EXECUTE IMMEDIATE 'sql_expression;'
  [ INTO { variable [, ...] | record } ]
  [ USING expression [, ...] ]
```

sql_expression is a string expression containing the SQL statement to be dynamically executed. variable receives the output of the result set typically from a SELECT statement . This statement is created as a result of executing the SQL statement in sql_expression . The number, order, and type of variables must match the number, order, and be type -compatible with the fields of the result set. Alternatively, a record can be specified as long as the fields of the record match the number, order, and are type-compatible with the result set. When the INTO clause is used, exactly one row must be returned in the result set. Otherwise an exception occurs. When the USING clause is used, the value of expression is passed to a placeholder. Placeholders appear embedded within the SQL statement in sql_expression where variables may be used. Placeholders are denoted by an identifier with a colon (:) prefix - :name. The number, order, and resultant data types of the evaluated expressions must match the number, order and be type-compatible with the placeholders in sql_expression. Note that placeholders are not declared anywhere in the SPL program - they only appear in sql_expression.

The following example shows basic dynamic SQL statements as string literals:

```
DECLARE
    v_sql          VARCHAR2(50);
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE job (jobno NUMBER(3),' ||
        ' jname VARCHAR2(9))';
    v_sql := 'INSERT INTO job VALUES (100, ''ANALYST'')';
     EXECUTE IMMEDIATE v_sql;
    v_sql := 'INSERT INTO job VALUES (200, ''CLERK'')';
    EXECUTE IMMEDIATE v_sql;
END;
```

The following example illustrates the USING clause to pass values to placeholders in the SQL string:

```
DECLARE
     v_sql          VARCHAR2(50) := 'INSERT INTO job VALUES ' ||
                         '(:p_jobno, :p_jname)';
    v_jobno        job.jobno%TYPE;
    v_jname        job.jname%TYPE;
BEGIN
    V_jobno: = 300;
    v_jname := 'MANAGER';
    EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
    v_jobno := 400;
    v_jname := 'SALESMAN';
    EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
    v_jobno := 500;
    v_jname := 'PRESIDENT';
    EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
END;
```

The following example shows both the INTO and USING clauses. Note that the last execution of the SELECT statement returns the results into a record instead of individual variables.

```
DECLARE
    v_sql          VARCHAR2(60);
    v_jobno        job.jobno%TYPE;
    v_jname        job.jname%TYPE;
    r_job          job%ROWTYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('JOBNO    JNAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    v_sql := 'SELECT jobno, jname FROM job WHERE jobno = :p_jobno';
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 100;
     DBMS_OUTPUT.PUT_LINE(v_jobno || '      ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 200;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '      ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 300;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '      ' || v_jname);
     EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 400;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '      ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO r_job USING 500;
    DBMS_OUTPUT.PUT_LINE(r_job.jobno || '      ' || r_job.jname);
END;
```

The following code is the output from the previous anonymous block:

```
JOBNO    JNAME
-----    -------
100      ANALYST
200      CLERK
```

```
300     MANAGER
400     SALESMAN
500     PRESIDENT
```

You can use the BULK COLLECT clause to assemble the result set from an EXECUTE IMMEDIATE statement into a named collection.

# 7.13 Static cursors

## 7.13.1 Overview

Rather than executing a whole query at a time, it is possible to set up a cursor that encapsulates the query, and then read the query result set one row at a time. This allows the creation of SPL program logic that retrieves a row from the result set, does some processing on the data in that row, and then retrieves the next row and repeats the process .

Cursors are most often used in the context of a FOR or WHILE loop. A conditional test should be included in the SPL logic that detects when the end of the result set has been reached so the program can exit the loop.

## 7.13.2 Declare a cursor

To use a cursor, it must first be declared in the declaration topic of the SPL program. A cursor declaration appears as follows:

```
CURSOR name IS query;
```

name is an identifier that will be used to reference the cursor and its result set later in the program. query is a SQL SELECT statement that determines the result set retrievable by the cursor.

The following codes are some examples of cursor declarations:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_1 IS SELECT * FROM emp;
    CURSOR emp_cur_2 IS SELECT empno, ename FROM emp;
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    ...
```

```
END;
```

## 7.13.3 Open a cursor

Before a cursor can be used to retrieve rows, it must first be opened. This is accomplished with the OPEN statement.

```
OPEN name;
```

name is the identifier of a cursor that has been previously declared in the declaration topic of the SPL program. The OPEN statement must not be executed on a cursor that has already been and still is open.

The following code shows an OPEN statement with its corresponding cursor declaration:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
        ...
END;
```

## 7.13.4 Fetch rows from a cursor

After a cursor has been opened, rows can be retrieved from the result set of the cursor by using the FETCH statement.

```
FETCH name INTO { record | variable [, variable_2 ]... };
```

name is the identifier of a previously opened cursor. record is the identifier of a previously defined record such as using table%ROWTYPE. variable, variable_2... are SPL variables that will receive the field data from the fetched row. The fields in record or variable, variable_2 ... must match in number and order the fields returned in the SELECT list of the query given in the cursor declaration. The data types of the fields in the SELECT list must match or be implicitly convertible to the data types of the fields in record or the data types of variable, variable_2...

**Note:**

A variation of FETCH INTO using the BULK COLLECT clause exists. This variation can return multiple rows at a time into a collection.

The following code shows the FETCH statement:

```
CREATE OR REPLACE PROCEDURE cursor_example
```

```
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
        ...
END;
```

Instead of explicitly declaring the data type of a target variable, %TYPE can be used. In this way, if the data type of the database column is changed, the target variable declaration in the SPL program does not have to be changed. %TYPE will automatically pick up the new data type of the specified column.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
        ...
END;
```

If all the columns in a table are retrieved in the order defined in the table, %ROWTYPE can be used to define a record into which the FETCH statement will place the retrieved data. Each field within the record can then be accessed by using dot notation.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    FETCH emp_cur_1 INTO v_emp_rec;
    DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
    DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
        ...
```

```
END;
```

## 7.13.5 Close a cursor

After all the desired rows have been retrieved from the cursor result set, the cursor must be closed. After the cursor is closed, the result set is no longer accessible. The CLOSE statement appears as follows:

```
CLOSE name;
```

name is the identifier of a cursor that is currently open. After a cursor is closed, it must not be closed again. However, after the cursor is closed, the OPEN statement can be issued again on the closed cursor and the query result set will be rebuilt after which the FETCH statement can then be used to retrieve the rows of the new result set.

The following example illustrates the use of the CLOSE statement:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    FETCH emp_cur_1 INTO v_emp_rec;
    DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
    DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
    CLOSE emp_cur_1;
END;
```

This procedure produces the following output when invoked: Employee number 7369, SMITH is the first row of the result set.

```
EXEC cursor_example;

Employee Number: 7369
Employee Name: SMITH
```

## 7.13.6 Use %ROWTYPE with cursors

The %ROWTYPE attribute can be used to define a record that contains fields corresponding to all columns fetched from a cursor or cursor variable. Each field takes on the data type of its corresponding column. The %ROWTYPE attribute is prefixed by a cursor name or cursor variable name.

```
record cursor%ROWTYPE;
```

record is an identifier assigned to the record. cursor is an explicitly declared cursor within the current scope.

The following example shows how you can use a cursor with %ROWTYPE to get information about which employee works in which department:

```
CREATE OR REPLACE PROCEDURE emp_info
IS
    CURSOR empcur IS SELECT ename, deptno FROM emp;
    myvar          empcur%ROWTYPE;
BEGIN
    OPEN empcur;
    LOOP
        FETCH empcur INTO myvar;
        EXIT WHEN empcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( myvar.ename || ' works in department '
            || myvar.deptno );
    END LOOP;
    CLOSE empcur;
END;
```

The following output is generated from this procedure:

```
EXEC emp_info;

SMITH works in department 20
ALLEN works in department 30
WARD works in department 30
JONES works in department 20
MARTIN works in department 30
BLAKE works in department 30
CLARK works in department 10
SCOTT works in department 20
KING works in department 10
TURNER works in department 30
ADAMS works in department 20
JAMES works in department 30
FORD works in department 20
MILLER works in department 10
```

## 7.13.7 Cursor attributes

Each cursor has a set of attributes associated with it that allows the program to test the state of the cursor. These attributes are %ISOPEN, %FOUND, %NOTFOUND, and %ROWCOUNT. These attributes are described in the following topics.

### %ISOPEN

The %ISOPEN attribute is used to test whether a cursor is open.

```
cursor_name%ISOPEN
```

cursor_name is the name of the cursor. If the cursor is open, a BOOLEAN data type of TRUE is returned. Otherwise, FALSE is returned.

The following example uses %ISOPEN:

```
CREATE OR REPLACE PROCEDURE cursor_example
```

```
IS
      ...
    CURSOR emp_cur_1 IS SELECT * FROM emp;
      ...
BEGIN
      ...
   IF emp_cur_1%ISOPEN THEN
      NULL;
   ELSE
      OPEN emp_cur_1;
   END IF;
   FETCH emp_cur_1 INTO...
      ...
END;
```

### %FOUND

The %FOUND attribute is used to test whether a row is retrieved from the result set of the specified cursor after a FETCH on the cursor.

```
cursor_name%FOUND
```

cursor_name is the name of the cursor for which a BOOLEAN data type of TRUE will be returned if a row is retrieved from the result set of the cursor after a FETCH.

After the last row of the result set has been FETCHed, the next FETCH results in %FOUND returning FALSE. FALSE is also returned after the first FETCH if the result set has no rows to begin with.

Referencing %FOUND on a cursor before it is opened or after it is closed results in an INVALID_CURSOR exception being thrown.

%FOUND returns null if it is referenced when the cursor is open, but before the first FETCH.

The following example uses %FOUND:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
   v_emp_rec       emp%ROWTYPE;
   CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
   OPEN emp_cur_1;
   DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
   DBMS_OUTPUT.PUT_LINE('-----    -------');
   FETCH emp_cur_1 INTO v_emp_rec;
   WHILE emp_cur_1%FOUND LOOP
      DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
      FETCH emp_cur_1 INTO v_emp_rec;
   END LOOP;
   CLOSE emp_cur_1;
END;
```

When the previous procedure is invoked, the output appears as follows:

```
EXEC cursor_example;
```

```
EMPNO    ENAME
 -----    ------
7369    SMITH
7499    ALLEN
7521    WARD
7566    JONES
7654    MARTIN
7698    BLAKE
7782    CLARK
7788    SCOTT
7839    KING
7844    TURNER
7876    ADAMS
7900    JAMES
7902    FORD
7934    MILLER
```

**%NOTFOUND**

The %NOTFOUND attribute is the logical opposite of %FOUND.

```
cursor_name%NOTFOUND
```

cursor_name is the name of the cursor for which a BOOLEAN data type of FALSE will be returned if a row is retrieved from the result set of the cursor after a FETCH.

After the last row of the result set has been FETCHed, the next FETCH results in %NOTFOUND returning TRUE. TRUE is also returned after the first FETCH if the result set has no rows to begin with.

Referencing %NOTFOUND on a cursor before it is opened or after it is closed results in an INVALID_CURSOR exception being thrown.

%NOTFOUND returns null if it is referenced when the cursor is open, but before the first FETCH.

The following example uses %NOTFOUND:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cur_1 INTO v_emp_rec;
        EXIT WHEN emp_cur_1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
    END LOOP;
    CLOSE emp_cur_1;
```

```
END;
```

Similar to the prior example, this procedure produces the same output when invoked:

```
EXEC cursor_example;

EMPNO    ENAME
-----    ------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
```

**%ROWCOUNT**

The %ROWCOUNT attribute returns an integer showing the number of rows FETCHed so far from the specified cursor.

```
cursor_name%ROWCOUNT
```

cursor_name is the name of the cursor for which %ROWCOUNT returns the number of rows retrieved thus far. After the last row has been retrieved, %ROWCOUNT remains set to the total number of rows returned until the cursor is closed at which point %ROWCOUNT will throw an INVALID_CURSOR exception if referenced.

Referencing %ROWCOUNT on a cursor before it is opened or after it is closed results in an INVALID_CURSOR exception being thrown.

%ROWCOUNT returns 0 if it is referenced when the cursor is open, but before the first FETCH . %ROWCOUNT also returns 0 after the first FETCH when the result set has no rows to begin with.

The following example uses %ROWCOUNT:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cur_1 INTO v_emp_rec;
        EXIT WHEN emp_cur_1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(emp_cur_1%ROWCOUNT || ' rows were retrieved');
    CLOSE emp_cur_1;
END;
```

This procedure prints the total number of rows retrieved at the end of the employee list as follows:

```
EXEC cursor_example;

EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
**********************
14 rows were retrieved
```

**Summary of cursor states and attributes**

The following table summarizes the possible cursor states and the values returned by the cursor attributes.

| Cursor state | %ISOPEN | %FOUND | %NOTFOUND | %ROWCOUNT |
|---|---|---|---|---|
| Before OPEN | False | INVALID_CURSOR exception | INVALID_CURSOR exception | INVALID_CURSOR exception |
| After OPEN & Before 1st FETCH | True | Null | Null | 0 |
| After 1st Successful FETCH | True | True | False | 1 |
| After nth Successful FETCH (last row) | True | True | False | n |

| Cursor state | %ISOPEN | %FOUND | %NOTFOUND | %ROWCOUNT |
|---|---|---|---|---|
| After n+1st FETCH (after last row) | True | False | True | n |
| After CLOSE | False | INVALID_CURSOR exception | INVALID_CURSOR exception | INVALID_CURSOR exception |

## 7.13.8 Cursor FOR loop

In the cursor examples presented so far, the programming logic required to process the result set of a cursor includes a statement to open the cursor, a loop construct to retrieve each row of the result set, a test for the end of the result set, and a statement to close the cursor. The cursor FOR loop is a loop construct that eliminates the need to individually code the statements just listed.

The cursor FOR loop opens a previously declared cursor, fetches all rows in the cursor result set, and then closes the cursor.

The syntax for creating a cursor FOR loop is as follows:

```
FOR record IN cursor
LOOP
   statements
END LOOP;
```

record is an identifier assigned to an implicitly declared record with definition cursor%ROWTYPE. cursor is the name of a previously declared cursor. statements are one or more SPL statements. At least one statement must exist.

The following example shows the example from %NOTFOUND that is modified to use a cursor FOR loop:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR v_emp_rec IN emp_cur_1 LOOP
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
    END LOOP;
END;
```

The same results are achieved as shown in the following output:

```
EXEC cursor_example;

EMPNO    ENAME
-----    -------
```

```
7369    SMITH
7499    ALLEN
7521    WARD
7566    JONES
7654    MARTIN
7698    BLAKE
7782    CLARK
7788    SCOTT
7839    KING
7844    TURNER
7876    ADAMS
7900    JAMES
7902    FORD
7934    MILLER
```

### 7.13.9 Parameterized cursors

A user can also declare a static cursor that accepts parameters and can pass values for those parameters when that cursor is opened. In the following example, a parameterized cursor is created. The cursor will display the name and salary of all employees from the emp table that have a salary less than a specified value which is passed as a parameter.

```
DECLARE
    my_record       emp%ROWTYPE;
    CURSOR c1 (max_wage NUMBER) IS
        SELECT * FROM emp WHERE sal < max_wage;
BEGIN
    OPEN c1(2000);
    LOOP
        FETCH c1 INTO my_record;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Name = ' || my_record.ename || ', salary = '
            || my_record.sal);
    END LOOP;
    CLOSE c1;
END;
```

For example, if we pass the value 2000 as max_wage, we will only be shown the name and salary of all employees that have a salary less than 2000. The following data shows the result of the above query:

```
Name = SMITH, salary = 800.00
Name = ALLEN, salary = 1600.00
Name = WARD, salary = 1250.00
Name = MARTIN, salary = 1250.00
Name = TURNER, salary = 1500.00
Name = ADAMS, salary = 1100.00
Name = JAMES, salary = 950.00
Name = MILLER, salary = 1300.00
```

## 7.14 REF CURSOR and cursor variable

## 7.14.1 REF CURSOR overview

A cursor variable is a cursor that actually contains a pointer to a query result set. The result set is determined by the execution of the OPEN FOR statement by using the cursor variable.

A cursor variable is not tied to a single particular query such as a static cursor. The same cursor variable may be opened a number of times by using OPEN FOR statements containing different queries. Each time, a new result set is created from that query and made available by using the cursor variable.

REF CURSOR types may be passed as parameters to or from stored procedures and functions. The return type of a function may also be a REF CURSOR type. This provides the capability to modularize the operations on a cursor into separate programs by passing a cursor variable between programs.

## 7.14.2 Declare a cursor variable

SPL supports the declaration of a cursor variable by using both the SYS_REFCURSOR built-in data type as well as creating a type of REF CURSOR and then declaring a variable of that type. SYS_REFCURSOR is a REF CURSOR type that allows any result set to be associated with it. This is known as a weakly-typed REF CURSOR.

Only the declaration of SYS_REFCURSOR and user-defined REF CURSOR variables are different. The remaining usage such as opening the cursor, selecting into the cursor, and closing the cursor is the same across both the cursor types. For the rest of this topic , examples will primarily be making use of the SYS_REFCURSOR cursors. All you need to change in the examples to make them work for user-defined REF CURSORs is the declaration section.

**Note:**

Strongly-typed REF CURSORs require the result set to conform to a declared number and order of fields with compatible data types and can also optionally return a result set.

### Declare a SYS_REFCURSOR cursor variable

The following code is the syntax for declaring a SYS_REFCURSOR cursor variable:

```
name SYS_REFCURSOR;
```

name is an identifier assigned to the cursor variable.

The following code is an example of a SYS_REFCURSOR variable declaration:

```
DECLARE
```

```
    emp_refcur      SYS_REFCURSOR;
        ...
```

**Declare a user-defined REF CURSOR type variable**

You must perform two distinct declaration steps to use a user-defined REF CURSOR variable :

- Create a referenced cursor TYPE.
- Declare the actual cursor variable based on that TYPE.

The syntax for creating a user defined REF CURSOR type is as follows:

```
TYPE cursor_type_name IS REF CURSOR [RETURN return_type];
```

The following code is an example of a cursor variable declaration:

```
DECLARE
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    my_rec emp_cur_type;
        ...
```

## 7.14.3 Open a cursor variable

After a cursor variable is declared, it must be opened with an associated SELECT statement. The OPEN FOR statement specifies the SELECT statement to be used to create the result set.

```
OPEN name FOR query;
```

name is the identifier of a previously declared cursor variable. query is a SELECT statement that determines the result set when the statement is executed. The value of the cursor variable after the OPEN FOR statement is executed identifies the result set.

In the following example, the result set is a list of employee numbers and names from a selected department. Note that a variable or parameter can be used in the SELECT statement anywhere an expression can normally appear. In this case, a parameter is used in the equality test for department number.

```
CREATE OR REPLACE PROCEDURE emp_by_dept (
    p_deptno        emp.deptno%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
```

```
        ...
```

## 7.14.4 Fetch rows from a cursor variable

After a cursor variable is opened, rows may be retrieved from the result set by using the FETCH statement.

In the following example, a FETCH statement has been added to the previous example so now the result set is returned into two variables and then displayed. Note that the cursor attributes used to determine cursor state of static cursors can also be used with cursor variables.

```
CREATE OR REPLACE PROCEDURE emp_by_dept (
    p_deptno        emp.deptno%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
        ...
```

## 7.14.5 Close a cursor variable

**Note:**

Unlike static cursors, a cursor variable does not have to be closed before it can be re-opened again. The result set from the previously opened cursor variable will be lost.

The example is completed with the addition of the CLOSE statement:

```
CREATE OR REPLACE PROCEDURE emp_by_dept (
    p_deptno        emp.deptno%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

The following output is generated when this procedure is executed:

```
EXEC emp_by_dept(20)

EMPNO    ENAME
-----    -------
7369     SMITH
7566     JONES
7788     SCOTT
7876     ADAMS
7902     FORD
```

## 7.14.6 Usage restrictions

The restrictions on cursor variable usage are as follows:

- Comparison operators cannot be used to test cursor variables for equality, inequality, null, or not null.
- Null cannot be assigned to a cursor variable.
- The value of a cursor variable cannot be stored in a database column.
- Static cursors and cursor variables are not interchangeable. For example, a static cursor cannot be used in an OPEN FOR statement.

In addition, the following table describes the permitted parameter modes for a cursor variable used as a procedure or function parameter depending upon the operations on the cursor variable within the procedure or function.

**Table 7-3: Permitted cursor variable parameter modes**

| Operation | IN | IN OUT | OUT |
|---|---|---|---|
| OPEN | No | Yes | No |
| FETCH | Yes | Yes | No |
| CLOSE | Yes | Yes | No |

For example, if a procedure performs the OPEN FOR, FETCH, and CLOSE operations on a cursor variable declared as the formal parameter of the procedure, that parameter must be declared with IN OUT mode.

## 7.14.7 Examples

The following examples demonstrate cursor variable usage.

**Return a REF CURSOR from a function**

In the following example, the cursor variable is opened with a query that selects employees with a given job. Note that the cursor variable is specified in this RETURN statement of the function so the result set is made available to the caller of the function.

```
CREATE OR REPLACE FUNCTION emp_by_job (p_job VARCHAR2)
RETURN SYS_REFCURSOR
IS
    emp_refcur      SYS_REFCURSOR;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE job = p_job;
    RETURN emp_refcur;
END;
```

This function is invoked in the following anonymous block by assigning the return value of the function to a cursor variable that is declared in the declaration topic of the anonymous block. The result set is fetched by using this cursor variable and then it is closed.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE := 'SALESMAN';
    v_emp_refcur    SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES WITH JOB ' ‖ v_job);
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    v_emp_refcur := emp_by_job(v_job);
    LOOP
        FETCH v_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN v_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno ‖ '     ' ‖ v_ename);
    END LOOP;
    CLOSE v_emp_refcur;
END;
```

The following output is generated when the anonymous block is executed:

```
EMPLOYEES WITH JOB SALESMAN
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7844     TURNER
```

**Modularize cursor operations**

The following example illustrates how the various operations on cursor variables can be modularized into separate programs.

The following procedure opens the given cursor variable with a SELECT statement that retrieves all rows:

```
CREATE OR REPLACE PROCEDURE open_all_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp;
END;
```

This variation opens the given cursor variable with a SELECT statement that retrieves all rows but of a given department.

```
CREATE OR REPLACE PROCEDURE open_emp_by_dept (
    p_emp_refcur    IN OUT SYS_REFCURSOR,
    p_deptno        emp.deptno%TYPE
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp
        WHERE deptno = p_deptno;
END;
```

This third variation opens the given cursor variable with a SELECT statement that retrieves all rows but from a different table. Also note that the return value of the function is the opened cursor variable.

```
CREATE OR REPLACE FUNCTION open_dept (
    p_dept_refcur    IN OUT SYS_REFCURSOR
) RETURN SYS_REFCURSOR
IS
    v_dept_refcur    SYS_REFCURSOR;
BEGIN
    v_dept_refcur := p_dept_refcur;
    OPEN v_dept_refcur FOR SELECT deptno, dname FROM dept;
    RETURN v_dept_refcur;
END;
```

This procedure fetches and displays a cursor variable result set consisting of employee number and name:

```
CREATE OR REPLACE PROCEDURE fetch_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
    v_empno        emp.empno%TYPE;
    v_ename        emp.ename%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH p_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN p_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
```

```
END;
```

This procedure fetches and displays a cursor variable result set consisting of department number and name:

```
CREATE OR REPLACE PROCEDURE fetch_dept (
    p_dept_refcur   IN SYS_REFCURSOR
)
IS
    v_deptno        dept.deptno%TYPE;
    v_dname         dept.dname%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('DEPT   DNAME');
    DBMS_OUTPUT.PUT_LINE('----   ---------');
    LOOP
        FETCH p_dept_refcur INTO v_deptno, v_dname;
        EXIT WHEN p_dept_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_deptno || '     ' || v_dname);
    END LOOP;
END;
```

This procedure closes the given cursor variable:

```
CREATE OR REPLACE PROCEDURE close_refcur (
    p_refcur        IN OUT SYS_REFCURSOR
)
IS
BEGIN
    CLOSE p_refcur;
END;
```

The following anonymous block executes all the previously described programs:

```
DECLARE
    gen_refcur      SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('ALL EMPLOYEES');
    open_all_emp(gen_refcur);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('****************');

    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #10');
    open_emp_by_dept(gen_refcur, 10);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('****************');

    DBMS_OUTPUT.PUT_LINE('DEPARTMENTS');
    fetch_dept(open_dept(gen_refcur));
    DBMS_OUTPUT.PUT_LINE('*****************');

    close_refcur(gen_refcur);
END;
```

The following output is generated from the anonymous block:

```
ALL EMPLOYEES
EMPNO    ENAME
-----    -------
7369     SMITH
```

```
7499    ALLEN
7521    WARD
7566    JONES
7654    MARTIN
7698    BLAKE
7782    CLARK
7788    SCOTT
7839    KING
7844    TURNER
7876    ADAMS
7900    JAMES
7902    FORD
7934    MILLER
****************
EMPLOYEES IN DEPT #10
EMPNO    ENAME
-----    -------
7782    CLARK
7839    KING
7934    MILLER
****************
DEPARTMENTS
DEPT   DNAME
----   ---------
10     ACCOUNTING
20     RESEARCH
30     SALES
40     OPERATIONS
*****************
```

## 7.14.8 Dynamic queries with REF CURSORs

PolarDB database compatible with Oracle also supports dynamic queries by using the OPEN FOR USING statement. A string literal or string variable is supplied in the OPEN FOR USING statement to the SELECT statement.

```
OPEN name FOR dynamic_string
  [ USING bind_arg [, bind_arg_2 ] ...] ;
```

name is the identifier of a previously declared cursor variable. dynamic_string is a string literal or string variable containing a SELECT statement (without the terminating semi-colon (;)). bind_arg, bind_arg_2... are bind arguments that are used to pass variables to corresponding placeholders in the SELECT statement when the cursor variable is opened. The placeholders are identifiers prefixed by a colon character.

The following code is an example of a dynamic query using a string literal:

```
CREATE OR REPLACE PROCEDURE dept_query
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = 30' ||
        ' AND sal >= 1500';
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

The following output is generated when the procedure is executed:

```
EXEC dept_query;

EMPNO    ENAME
-----    -------
7499     ALLEN
7698     BLAKE
7844     TURNER
```

In the next example, the previous query is modified to use bind arguments to pass the query parameters:

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = :dept'
        || ' AND sal >= :sal' USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

The following output is generated:

```
EXEC dept_query(30, 1500);

EMPNO    ENAME
-----    -------
7499     ALLEN
7698     BLAKE
7844     TURNER
```

Finally, a string variable is used to pass SELECT. This provides the most flexibility.

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
```

```
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    p_query_string VARCHAR2 (100);
BEGIN
    p_query_string := 'SELECT empno, ename FROM emp WHERE ' ||
        'deptno = :dept AND sal >= :sal';
    OPEN emp_refcur FOR p_query_string USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
EXEC dept_query(20, 1500);

EMPNO    ENAME
-----    -------
7566     JONES
7788     SCOTT
7902     FORD
```

# 7.15 Collections

## 7.15.1 Overview

A collection is a set of ordered data items with the same data type. Generally, the data item is a scalar field, but may also be a user-defined type such as a record type or an object type as long as the structure and the data types that comprise each field of the user-defined type are the same for each element in the set. Each particular data item in the set is referenced by using subscript notation within a pair of parentheses.

**Note:**

Multilevel collections (that is, where the data item of a collection is another collection) are not supported.

The most commonly known type of collection is an array. In PolarDB database compatible with Oracle, the supported collection types are associative arrays (formerly called index-by-tables in Oracle), nested tables, and varrays.

The general steps for using a collection are as follows:

- A collection of the desired type must be defined. This can be done in the declaration topic of an SPL program, which results in a local type that is accessible only within that program. For nested table and varray types, this can also be done by using the CREATE

TYPE statement, which creates a persistent standalone type that can be referenced by any SPL program in the database.

- Variables of the collection type are declared. The collection associated with the declared variable is said to be uninitialized at this point if no value assignment is made as part of the variable declaration.
- Uninitialized collections of nested tables and varrays are null. A null collection does not yet exist. Generally, a COLLECTION_IS_NULL exception is thrown if a collection method is invoked on a null collection.
- Uninitialized collections of associative arrays exist but have no elements. An existing collection with no elements is called an empty collection.
- To initialize a null collection, you must either make it an empty collection or assign a non -null value to it. Generally, a null collection is initialized by using its constructor.
- To add elements to an empty associative array, you can simply assign values to its keys . For nested tables and varrays, generally its constructor is used to assign initial values to the nested table or varray. For nested tables and varrays, the EXTEND method is then used to grow the collection beyond its initial size established by the constructor.

The specific process for each collection type is described in the following topics.

## 7.15.2 Associative arrays

An associative array is a type of collection that associates a unique key with a value. The key does not have to be numeric but can be character data as well.

An associative array has the following characteristics:

- An associative array type must be defined after which array variables can be declared of that array type. Data manipulation occurs by using the array variable.
- When an array variable is declared, the associative array is created but is empty - just start assigning values to key values.
- The key can be any negative integer, positive integer, or zero if INDEX BY BINARY_INTEGER or PLS_INTEGER is specified.
- The key can be character data if INDEX BY VARCHAR2 is specified.
- The number of elements in the array has no pre-defined limit - it grows dynamically as elements are added.
- The array can be sparse - gaps may exist in the assignment of values to keys.
- An attempt to reference an array element that has not been assigned a value will result in an exception.

The TYPE IS TABLE OF ... INDEX BY statement is used to define an associative array type:

```
TYPE assoctype IS TABLE OF { datatype | rectype | objtype }
  INDEX BY { BINARY_INTEGER | PLS_INTEGER | VARCHAR2(n) };
```

assoctype is an identifier assigned to the array type. datatype is a scalar data type such as VARCHAR2 or NUMBER. rectype is a previously defined record type. objtype is a previously defined object type. n is the maximum length of a character key.

To use the array, a variable must be declared with that array type. The syntax for declaring an array variable is as follows:

```
array assoctype
```

array is an identifier assigned to the associative array. assoctype is the identifier of a previously defined array type.

An element of the array is referenced by using the following syntax:

```
array(n)[.field ]
```

array is the identifier of a previously declared array. n is the key value, type-compatible with the data type given in the INDEX BY clause. If the array type of array is defined from a record type or object type, [.field ] must reference an individual field within the record type or attribute within the object type from which the array type is defined. Alternatively, the entire record can be referenced by omitting [.field ].

The following example reads the first ten employee names from the emp table, stores them in an array, and then displays the results from the array:

```
DECLARE
    TYPE emp_arr_typ IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
    emp_arr         emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;
END;
```

The above example produces the following output:

```
SMITH
ALLEN
WARD
JONES
```

```
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
```

The previous example is now modified to use a record type in the array definition:

```
DECLARE
    TYPE emp_rec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR2(10)
    );
    TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
    emp_arr         emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno ‖ '     ' ‖
            emp_arr(j).ename);
    END LOOP;
END;
```

The following output is generated from this anonymous block:

```
EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
```

The emp%ROWTYPE attribute could be used to define emp_arr_typ instead of using the emp_rec_typ record type as shown in the following example:

```
DECLARE
    TYPE emp_arr_typ IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    emp_arr         emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END;
```

The results are the same as in the prior example.

Instead of assigning each field of the record individually, a record level assignment can be made from r_emp to emp_arr.

```
DECLARE
    TYPE emp_rec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR2(10)
    );
    TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
    emp_arr         emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END;
```

The key of an associative array can be character data as shown in the following example:

```
DECLARE
    TYPE job_arr_typ IS TABLE OF NUMBER INDEX BY VARCHAR2(9);
    job_arr         job_arr_typ;
BEGIN
    job_arr('ANALYST')   := 100;
    job_arr('CLERK')     := 200;
    job_arr('MANAGER')   := 300;
    job_arr('SALESMAN')  := 400;
    job_arr('PRESIDENT') := 500;
    DBMS_OUTPUT.PUT_LINE('ANALYST  : ' || job_arr('ANALYST'));
    DBMS_OUTPUT.PUT_LINE('CLERK    : ' || job_arr('CLERK'));
    DBMS_OUTPUT.PUT_LINE('MANAGER  : ' || job_arr('MANAGER'));
    DBMS_OUTPUT.PUT_LINE('SALESMAN : ' || job_arr('SALESMAN'));
    DBMS_OUTPUT.PUT_LINE('PRESIDENT: ' || job_arr('PRESIDENT'));
END;

ANALYST  : 100
CLERK    : 200
MANAGER  : 300
SALESMAN : 400
```

```
PRESIDENT: 500
```

## 7.15.3 Nested tables

A nested table is a type of collection that associates a positive integer with a value. A nested table has the following characteristics:

- A nested table type must be defined after which nested table variables can be declared of that nested table type. Data manipulation occurs by using the nested table variable or simply "table" for short.
- When a nested table variable is declared, the nested table initially does not exist (it is a null collection). The null table must be initialized with a constructor. You can also initialize the table by using an assignment statement where the right-hand side of the assignment is an initialized table of the same type. Note: Initialization of a nested table is mandatory in Oracle, but optional in SPL.
- The key is a positive integer.
- The constructor establishes the number of elements in the table. The EXTEND method adds additional elements to the table. Note: Usage of the constructor to establish the number of elements in the table and usage of the EXTEND method to add additional elements to the table are required in Oracle but optional in SPL.
- The table can be sparse - the assignment of values to keys may have gaps:
- An attempt to reference a table element beyond its initialized or extended size will result in a SUBSCRIPT_BEYOND_COUNT exception.

The TYPE IS TABLE statement is used to define a nested table type within the declaration section of an SPL program:

```
TYPE tbltype IS TABLE OF { datatype | rectype | objtype };
```

tbltype is an identifier assigned to the nested table type. datatype is a scalar data type such as VARCHAR2 or NUMBER. rectype is a previously defined record type. objtype is a previously defined object type.

**Note:**

You can use the CREATE TYPE statement to define a nested table type that is available to all SPL programs in the database.

To use the table, a variable must be declared of that nested table type. The syntax for declaring a table variable is as follows:

- table tbltype

  table is an identifier assigned to the nested table. tbltype is the identifier of a previously defined nested table type.

  A nested table is initialized by using the constructor of the nested table type.

- tbltype ([ { expr1 | NULL } [, { expr2 | NULL } ] [, ...] ])

  tbltype is the identifier of the constructor of the nested table type. tbltype has the same name as the nested table type. expr1, expr2, ... are expressions that are type-compatible with the element type of the table. If NULL is specified, the corresponding element is set to null. If the parameter list is empty, an empty nested table is returned, which means no elements exist in the table. If the table is defined from an object type, exprn must return an object of that object type. The object can be the return value of a function or the constructor of the object type, or the object can be an element of another nested table of the same type.

If a collection method other than EXISTS is applied to an uninitialized nested table, a COLLECTION_IS_NULL exception is thrown.

The following code is an example of a constructor for a nested table:

```
DECLARE
    TYPE nested_typ IS TABLE OF CHAR(1);
    v_nested        nested_typ := nested_typ('A','B');
```

An element of the table is referenced by using the following syntax:

```
table(n)[.element ]
```

table is the identifier of a previously declared table. n is a positive integer. If the table type of table is defined from a record type or object type, [.element ] must reference an individual field within the record type or attribute within the object type from which the nested table type is defined. Alternatively, the entire record or object can be referenced by omitting [.element ].

The following code is an example of a nested table where it is known that four elements exist:

```
DECLARE
    TYPE dname_tbl_typ IS TABLE OF VARCHAR2(14);
    dname_tbl       dname_tbl_typ;
    CURSOR dept_cur IS SELECT dname FROM dept ORDER BY dname;
    i               INTEGER := 0;
BEGIN
    dname_tbl := dname_tbl_typ(NULL, NULL, NULL, NULL);
    FOR r_dept IN dept_cur LOOP
```

```
        i := i + 1;
        dname_tbl(i) := r_dept.dname;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('DNAME');
    DBMS_OUTPUT.PUT_LINE('----------');
    FOR j IN 1..i LOOP
        DBMS_OUTPUT.PUT_LINE(dname_tbl(j));
    END LOOP;
END;
```

The above example produces the following output:

```
DNAME
----------
ACCOUNTING
OPERATIONS
RESEARCH
SALES
```

The following example reads the first ten employee names from the emp table, stores them in a nested table, and then displays the results from the table. The SPL code is written to assume that the number of employees to be returned is not known beforehand.

```
DECLARE
    TYPE emp_rec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR2(10)
    );
    TYPE emp_tbl_typ IS TABLE OF emp_rec_typ;
    emp_tbl         emp_tbl_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    emp_tbl := emp_tbl_typ();
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_tbl.EXTEND;
        emp_tbl(i) := r_emp;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_tbl(j).empno || '     ' ||
            emp_tbl(j).ename);
    END LOOP;
END;
```

Note the creation of an empty table with the constructor emp_tbl_typ() as the first statement in the executable topic of the anonymous block. The EXTEND collection method is then used to add an element to the table for each employee returned from the result set.

The output is as follows:

```
EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
```

```
7521    WARD
7566    JONES
7654    MARTIN
7698    BLAKE
7782    CLARK
7788    SCOTT
7839    KING
7844    TURNER
```

The following example shows how a nested table of an object type can be used. First, an object type is created with attributes for the department name and location.

```
CREATE TYPE dept_obj_typ AS OBJECT (
    dname          VARCHAR2(14),
    loc            VARCHAR2(13)
);
```

The following anonymous block defines a nested table type whose element consists of the dept_obj_typ object type. A nested table variable is declared, initialized, and then populated from the dept table. Finally, the elements from the nested table are displayed.

```
DECLARE
    TYPE dept_tbl_typ IS TABLE OF dept_obj_typ;
    dept_tbl        dept_tbl_typ;
    CURSOR dept_cur IS SELECT dname, loc FROM dept ORDER BY dname;
    i               INTEGER := 0;
BEGIN
    dept_tbl := dept_tbl_typ(
        dept_obj_typ(NULL,NULL),
        dept_obj_typ(NULL,NULL),
        dept_obj_typ(NULL,NULL),
        dept_obj_typ(NULL,NULL)
    );
    FOR r_dept IN dept_cur LOOP
        i := i + 1;
        dept_tbl(i).dname := r_dept.dname;
        dept_tbl(i).loc   := r_dept.loc;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('DNAME          LOC');
    DBMS_OUTPUT.PUT_LINE('----------     ----------');
    FOR j IN 1..i LOOP
        DBMS_OUTPUT.PUT_LINE(RPAD(dept_tbl(j).dname,14) || ' ' ||
            dept_tbl(j).loc);
    END LOOP;
END;
```

**Note:**

The parameters comprising the constructor dept_tbl_typ for the nested table are calls to the constructor dept_obj_typ for the object type.

The following output is generated from the anonymous block:

```
DNAME          LOC
----------     ----------
ACCOUNTING     NEW YORK
```

```
OPERATIONS     BOSTON
RESEARCH       DALLAS
SALES          CHICAGO
```

## 7.15.4 Varrays

A varray or variable-size array is a type of collection that associates a positive integer with a value. In many respects, it is similar to a nested table.

A varray has the following characteristics:

- A varray type must be defined along with a maximum size limit. After the varray type is defined, varray variables can be declared of that varray type. Data manipulation occurs by using the varray variable or simply "varray" for short. The number of elements in the varray cannot exceed the maximum size limit established in the varray type definition.
- When a varray variable is declared, the varray initially does not exist (it is a null collection). The null varray must be initialized with a constructor. You can also initialize the varray by using an assignment statement where the right-hand side of the assignment is an initialized varray of the same type.
- The key is a positive integer.
- The constructor establishes the number of elements in the varray, which must not exceed the maximum size limit. The EXTEND method can add additional elements to the varray up to the maximum size limit.
- Unlike a nested table, a varray cannot be sparse - the assignment of values to keys has no gaps.
- An attempt to reference a varray element beyond its initialized or extended size but within the maximum size limit will result in a SUBSCRIPT_BEYOND_COUNT exception.
- An attempt to reference a varray element beyond the maximum size limit or extend a varray beyond the maximum size limit will result in a SUBSCRIPT_OUTSIDE_LIMIT exception.

The TYPE IS VARRAY statement is used to define a varray type within the declaration section of an SPL program:

```
TYPE varraytype IS { VARRAY | VARYING ARRAY }(maxsize)
  OF { datatype | objtype };
```

varraytype is an identifier assigned to the varray type. datatype is a scalar data type such as VARCHAR2 or NUMBER. maxsize is the maximum number of elements permitted in varrays of that type. objtype is a previously defined object type.

The CREATE TYPE statement can be used to define a varray type that is available to all SPL programs in the database. To use the varray, a variable must be declared of that varray type. The following is the syntax for declaring a varray variable:

```
varray varraytype
```

varray is an identifier assigned to the varray. varraytype is the identifier of a previously defined varray type.

A varray is initialized by using the constructor of the varray type.

```
varraytype ([ { expr1 | NULL } [, { expr2 | NULL } ]
  [, ...] ])
```

varraytype is the identifier of the constructor of the varray type, which has the same name as the varray type. expr1, expr2, ... are expressions that are type-compatible with the element type of the varray. If NULL is specified, the corresponding element is set to null . If the parameter list is empty, an empty varray is returned, which means no elements in the varray. If the varray is defined from an object type, exprn must return an object of that object type. The object can be the return value of a function or the return value of the constructor of the object type. The object can also be an element of another varray of the same varray type.

If a collection method other than EXISTS is applied to an uninitialized varray, a COLLECTION _IS_NULL exception is thrown.

The following example shows a constructor for a varray:

```
DECLARE
    TYPE varray_typ IS VARRAY(2) OF CHAR(1);
    v_varray        varray_typ := varray_typ('A','B');
```

An element of the varray is referenced by using the following syntax:

```
varray(n)[.element ]
```

varray is the identifier of a previously declared varray. n is a positive integer. If the varray type of varray is defined from an object type, [.element ] must reference an attribute within the object type from which the varray type is defined. Alternatively, the entire object can be referenced by omitting [.element ].

The following example shows a varray where it is known that four elements exist:

```
DECLARE
    TYPE dname_varray_typ IS VARRAY(4) OF VARCHAR2(14);
    dname_varray    dname_varray_typ;
    CURSOR dept_cur IS SELECT dname FROM dept ORDER BY dname;
    i               INTEGER := 0;
BEGIN
    dname_varray := dname_varray_typ(NULL, NULL, NULL, NULL);
    FOR r_dept IN dept_cur LOOP
        i := i + 1;
        dname_varray(i) := r_dept.dname;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('DNAME');
    DBMS_OUTPUT.PUT_LINE('----------');
    FOR j IN 1..i LOOP
        DBMS_OUTPUT.PUT_LINE(dname_varray(j));
    END LOOP;
END;
```

The above example produces the following output:

```
DNAME
----------
ACCOUNTING
OPERATIONS
RESEARCH
SALES
```

# 7.16 Collection methods

## 7.16.1 COUNT

COUNT is a method that returns the number of elements in a collection. The syntax for using COUNT is as follows:

```
collection.COUNT
```

collection is the name of a collection.

For a varray, COUNT always equals LAST.

The following example shows that an associative array can be sparsely populated (that is , the sequence of assigned elements has "gaps"). COUNT includes only the elements that have been assigned a value.

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
```

```
END;
```

The following output shows that COUNT includes five populated elements:

```
COUNT: 5
```

## 7.16.2 DELETE

The DELETE method deletes entries from a collection. You can call the DELETE method in three different ways:

Use the first form of the DELETE method to remove all entries from a collection:

```
collection.DELETE
```

Use the second form of the DELETE method to remove the specified entry from a collection:

```
collection.DELETE(subscript)
```

Use the third form of the DELETE method to remove the entries that are within the range specified by first_subscript and last_subscript (including the entries for the first_subscript and the last_subscript) from a collection:

```
collection.DELETE(first_subscript, last_subscript)
```

If first_subscript and last_subscript refer to non-existent elements, elements that are in the range between the specified subscripts are deleted. If first_subscript is greater than last_subscript or if you specify a value of NULL for one of the arguments, DELETE has no effect.

Note that when you delete an entry, the subscript remains in the collection. You can re-use the subscript with an alternate entry. If you specify a subscript that does not exist in the call to the DELETE method, DELETE does not raise an exception.

The following example demonstrates how to use the DELETE method to remove the element with subscript 0 from the collection:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
    v_results       VARCHAR2(50);
    v_sub           NUMBER;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.DELETE(0);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    v_sub := sparse_arr.FIRST;
    WHILE v_sub IS NOT NULL LOOP
        IF sparse_arr(v_sub) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(v_sub) || ' ';
        END IF;
        v_sub := sparse_arr.NEXT(v_sub);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 4
Results: -100 -10 10 100
```

COUNT indicates that before the DELETE method, the collection has five elements. After the DELETE method was invoked, the collection contains four elements.

## 7.16.3 EXISTS

The EXISTS method verifies that a subscript exists within a collection. EXISTS returns TRUE if the subscript exists. If the subscript does not exist, EXISTS returns FALSE. The method takes a single argument, which is the subscript that you are testing for. The syntax is as follows:

```
collection.EXISTS(subscript)
```

collection is the name of the collection.

subscript is the value that you are testing for. If you specify a value of NULL, EXISTS returns false.

The following example verifies that subscript number 10 exists within the associative array:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('The index exists: ' ||
        CASE WHEN sparse_arr.exists(10) = TRUE THEN 'true' ELSE 'false' END);
END;

The index exists: true
```

Some collection methods raise an exception if you call them with a subscript that does not exist within the specified collection. Rather than raising an error, the EXISTS method returns a value of FALSE.

## 7.16.4 EXTEND

The EXTEND method increases the size of a collection. The EXTEND method has three variations. The first variation appends a single NULL element to a collection. The syntax for the first variation is as follows:

```
collection.EXTEND
```

collection is the name of a collection.

The following example demonstrates how to use the EXTEND method to append a single null element to a collection:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr      sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results       VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 6
Results: -100 -10 0 10 100 NULL
```

COUNT indicates that before the EXTEND method, the collection has five elements. After the EXTEND method was invoked, the collection contains six elements.

The second variation of the EXTEND method appends a specified number of elements to the end of a collection:

```
collection.EXTEND(count)
```

collection is the name of a collection.

count is the number of null elements added to the end of the collection.

The following example demonstrates how to use the EXTEND method to append multiple null elements to a collection:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr      sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results       VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND(3);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 8
Results: -100 -10 0 10 100 NULL NULL NULL
```

COUNT indicates that before the EXTEND method, the collection has five elements. After the EXTEND method was invoked, the collection contains eight elements.

The third variation of the EXTEND method appends a specified number of copies of a particular element to the end of a collection:

```
collection.EXTEND(count, index_number)
```

- collection is the name of a collection.
- count is the number of elements added to the end of the collection.
- index_number is the subscript of the element that is being copied to the collection.

The following example demonstrates how to use the EXTEND method to append multiple copies of the second element to the collection:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr      sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results       VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND(3, 2);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 8
```

```
Results: -100 -10 0 10 100 -10 -10 -10
```

COUNT indicates that before the EXTEND method, the collection has five elements. After the EXTEND method was invoked, the collection contains eight elements.

**Note:**

The EXTEND method cannot be used on a null or empty collection.

## 7.16.5 FIRST

FIRST is a method that returns the subscript of the first element in a collection. The syntax for using FIRST is as follows:

```
collection.FIRST
```

collection is the name of a collection.

The following example displays the first element of the associative array:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('FIRST element: ' || sparse_arr(sparse_arr.FIRST));
END;

FIRST element: -100
```

## 7.16.6 LAST

LAST is a method that returns the subscript of the last element in a collection. The syntax for using LAST is as follows:

```
collection.LAST
```

collection is the name of a collection.

The following example displays the last element of the associative array:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('LAST element: ' || sparse_arr(sparse_arr.LAST));
END;

LAST element: 100
```

## 7.16.7 LIMIT

LIMIT is a method that returns the maximum number of elements permitted in a collection. LIMIT is applicable only to varrays. The syntax for using LIMIT is as follows:

```
collection.LIMIT
```

collection is the name of a collection.

For an initialized varray, LIMIT returns the maximum size limit determined by the varray type definition. If the varray is uninitialized (that is, it is a null varray), an exception is thrown.

For an associative array or an initialized nested table, LIMIT returns NULL. If the nested table is uninitialized (that is, it is a null nested table), an exception is thrown.

## 7.16.8 NEXT

NEXT is a method that returns the subscript that follows a specified subscript. The method takes a single argument, which is the subscript that you are testing for.

```
collection.NEXT(subscript)
```

collection is the name of the collection.

If the specified subscript is less than the first subscript in the collection, the function returns the first subscript. If the subscript does not have a successor, NEXT returns NULL. If you specify a NULL subscript, PRIOR does not return a value.

The following example demonstrates how to use NEXT to return the subscript that follows subscript 10 in the associative array, sparse_arr:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('NEXT element: ' || sparse_arr.next(10));
END;
```

```
NEXT element: 100
```

## 7.16.9 PRIOR

The PRIOR method returns the subscript that precedes a specified subscript in a collection . The method takes a single argument, that is the subscript that you are testing for. The syntax is as follows:

```
collection.PRIOR(subscript)
```

collection is the name of the collection.

If the subscript specified does not have a predecessor, PRIOR returns NULL. If the specified subscript is greater than the last subscript in the collection, the method returns the last subscript. If you specify a NULL subscript, PRIOR does not return a value.

The following example returns the subscript that precedes subscript 100 in the associative array, sparse_arr:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr      sparse_arr_typ;
BEGIN
    sparse_arr(-100)  := -100;
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    sparse_arr(100)   := 100;
    DBMS_OUTPUT.PUT_LINE('PRIOR element: ' || sparse_arr.prior(100));
END;

PRIOR element: 10
```

## 7.16.10 TRIM

The TRIM method removes an element or elements from the end of a collection. The syntax for the TRIM method is as follows:

```
collection.TRIM[(count)]
```

collection is the name of a collection.

count is the number of elements removed from the end of the collection. PolarDB database compatible with Oracle will return an error if count is less than 0 or greater than the number of elements in the collection.

The following example demonstrates how to use the TRIM method to remove an element from the end of a collection:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr      sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.TRIM;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
END;

COUNT: 5
COUNT: 4
```

COUNT indicates that before the TRIM method, the collection has five elements. After the TRIM method was invoked, the collection contains four elements.

You can also specify the number of elements to remove from the end of the collection with the TRIM method:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr      sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results       VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.TRIM(2);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 3
Results: -100 -10 0
```

COUNT indicates that before the TRIM method, the collection has five elements. After the TRIM method was invoked, the collection contains three elements.

## 7.17 Work with collections

## 7.17.1 TABLE()

You can use the TABLE() function to transform the members of an array into a set of rows. The signature is as follows:

```
TABLE(collection_value)
```

collection_value is an expression that evaluates to a value of collection type.

The TABLE() function expands the nested contents of a collection into a table format. You can use the TABLE() function anywhere you use a regular table expression.

The TABLE() function returns a SETOF ANYELEMENT (a set of values of any type). For example, if the argument passed to this function is an array of dates, TABLE() will return SETOF dates. If the argument passed to this function is an array of paths, TABLE() will return a SETOF paths.

You can use the TABLE() function to expand the contents of a collection into table form:

```
postgres=# SELECT * FROM TABLE(monthly_balance(445.00, 980.20, 552.00));

 monthly_balance
----------------
  445.00
  980.20
  552.00
(3 rows)
```

## 7.17.2 Use the MULTISET UNION operator

The MULTISET UNION operator combines two collections to form a third collection. The signature is as follows:

```
coll_1 MULTISET UNION [ALL | DISTINCT] coll_2
```

where, coll_1 and coll_2 specify the names of the collections to combine.

Include the ALL keyword to specify that duplicate elements (elements that are present in both coll_1 and coll_2) must be represented in the result once for each time they are present in the original collections. This is the default behavior of MULTISET UNION.

Include the DISTINCT or UNIQUE keyword to specify that duplicate elements should be included in the result only once.

The following example demonstrates using the MULTISET UNION operator to combine two collections (collection_1 and collection_2) into a third collection (collection_3):

```
DECLARE
```

```
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
    collection_1    int_arr_typ;
    collection_2    int_arr_typ;
    collection_3    int_arr_typ;
    v_results       VARCHAR2(50);
BEGIN
    collection_1 := int_arr_typ(10,20,30);
    collection_2 := int_arr_typ(30,40);
    collection_3 := collection_1 MULTISET UNION ALL collection_2;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
    FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
        IF collection_3(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || collection_3(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
Results: 10 20 30 30 40
```

The resulting collection includes one entry for each element in collection_1 and collection_2 . If the DISTINCT keyword is used, the results are as follows:

```
DECLARE
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
    collection_1    int_arr_typ;
    collection_2    int_arr_typ;
    collection_3    int_arr_typ;
    v_results       VARCHAR2(50);
BEGIN
    collection_1 := int_arr_typ(10,20,30);
    collection_2 := int_arr_typ(30,40);
    collection_3 := collection_1 MULTISET UNION DISTINCT collection_2;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
    FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
        IF collection_3(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || collection_3(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 4
Results: 10 20 30 40
```

The resulting collection includes only those members with distinct values. Note in the following example that the MULTISET UNION DISTINCT operator also removes duplicate entries that are stored within the same collection:

```
DECLARE
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
    collection_1    int_arr_typ;
    collection_2    int_arr_typ;
    collection_3    int_arr_typ;
```

```
   v_results      VARCHAR2(50);
BEGIN
   collection_1 := int_arr_typ(10,20,30,30);
   collection_2 := int_arr_typ(40,50);
   collection_3 := collection_1 MULTISET UNION DISTINCT collection_2;
   DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
   FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
      IF collection_3(i) IS NULL THEN
         v_results := v_results || 'NULL ';
      ELSE
         v_results := v_results || collection_3(i) || ' ';
      END IF;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
Results: 10 20 30 40 50
```

## 7.17.3 Use the FORALL statement

Collections can be used to more efficiently process DML statements by passing all the values to be used for repetitive execution of a DELETE, INSERT, or UPDATE statement in one pass to the database server rather than re-iteratively invoking the DML statement with new values. The DML statement to be processed in such a manner is specified with the FORALL statement. In addition, one or more collections are given in the DML statement where different values are to be substituted each time the statement is executed.

```
FORALL index IN lower_bound .. upper_bound
  { insert_stmt | update_stmt | delete_stmt };
```

index is the position in the collection given in the insert_stmt, update_stmt, or delete_stmt DML statement that iterates from the integer value given as lower_bound up to and including upper_bound.

**Note:**

If an exception occurs during any iteration of the FORALL statement, all updates that occurred since the start of the execution of the FORALL statement are automatically rolled back. This behavior is not compatible with Oracle databases. Oracle allows explicit use of the COMMIT or ROLLBACK statements to control whether to commit or roll back updates that occurred prior to the exception.

The FORALL statement creates a loop - each iteration of the loop increments the index variable (you typically use the index within the loop to select a member of a collection). The number of iterations is controlled by the lower_bound .. upper_bound clause. The loop is executed once for each integer between the lower_bound and upper_bound (inclusive) and the index is incremented by one for each iteration. Example:

```
FORALL i IN 2 .. 5
```

Creates a loop that executes four times: In the first iteration, the index (i) is set to the value 2. In the second iteration, the index is set to the value 3. The loop executes for the value 5 and then terminates.

The following example creates a table (emp_copy) that is an empty copy of the emp table . The example declares a type (emp_tbl) that is an array where each element in the array is of composite type and composed of the column definitions used to create the emp table. The example also creates an index on the emp_tbl type.

t_emp is an associative array of type emp_tbl. The SELECT statement uses the BULK COLLECT INTO statement to populate the t_emp array. After the t_emp array is populated, the FORALL statement iterates through the values (i) in the t_emp array index and inserts a row for each record into emp_copy.

```
CREATE TABLE emp_copy(LIKE emp);

DECLARE

    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;

    t_emp emp_tbl;

BEGIN
    SELECT * FROM emp BULK COLLECT INTO t_emp;

    FORALL i IN t_emp.FIRST .. t_emp.LAST
     INSERT INTO emp_copy VALUES t_emp(i);

END;
```

The following example uses a FORALL statement to update the salary of three employees:

```
DECLARE
    TYPE empno_tbl  IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl    IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;
    t_empno         EMPNO_TBL;
    t_sal           SAL_TBL;
BEGIN
    t_empno(1)  := 9001;
    t_sal(1)    := 3350.00;
    t_empno(2)  := 9002;
    t_sal(2)    := 2000.00;
    t_empno(3)  := 9003;
    t_sal(3)    := 4100.00;
    FORALL i IN t_empno.FIRST..t_empno.LAST
        UPDATE emp SET sal = t_sal(i) WHERE empno = t_empno(i);
END;
```

```
SELECT * FROM emp WHERE empno > 9000;

 empno | ename  |   job   | mgr | hiredate |   sal   | comm | deptno
-------+--------+---------+-----+----------+---------+------+--------
  9001 | JONES  | ANALYST |     |          | 3350.00 |      |     40
  9002 | LARSEN | CLERK   |     |          | 2000.00 |      |     40
  9003 | WILSON | MANAGER |     |          | 4100.00 |      |     40
(3 rows)
```

The following example deletes three employees in a FORALL statement:

```
DECLARE
    TYPE empno_tbl  IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
    t_empno         EMPNO_TBL;
BEGIN
    t_empno(1)  := 9001;
    t_empno(2)  := 9002;
    t_empno(3)  := 9003;
    FORALL i IN t_empno.FIRST..t_empno.LAST
        DELETE FROM emp WHERE empno = t_empno(i);
END;

SELECT * FROM emp WHERE empno > 9000;

 empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
(0 rows)
```

## 7.17.4 Use the BULK COLLECT clause

SQL statements that return a result set consisting of a large number of rows may not be operating as efficiently as possible due to the constant context switching that must occur between the database server and the client to transfer the entire result set. You can mitigate the inefficiency by using a collection to gather the entire result set in memory which the client can then access. The BULK COLLECT clause is used to specify the aggregation of the result set into a collection.

The BULK COLLECT clause can be used with the SELECT INTO, FETCH INTO, and EXECUTE IMMEDIATE statements, and with the RETURNING INTO clause of the DELETE, INSERT, and UPDATE statements. Each of these is illustrated in the following topics:

**SELECT BULK COLLECT**

The BULK COLLECT clause can be used with the SELECT INTO statement as follows:

```
SELECT select_expressions BULK COLLECT INTO collection
  [, ...] FROM ... ;
```

If a single collection is specified, collection may be a collection of a single field or it may be a collection of a record type. If more than one collection is specified, each collection must consist of a single field. select_expressions must match in number, order, and type-compatibility all fields in the target collections.

The following example shows the use of the BULK COLLECT clause where the target collections are associative arrays consisting of a single field:

```
DECLARE
    TYPE empno_tbl    IS TABLE OF emp.empno%TYPE    INDEX BY BINARY_INTEGER;
    TYPE ename_tbl    IS TABLE OF emp.ename%TYPE    INDEX BY BINARY_INTEGER;
    TYPE job_tbl      IS TABLE OF emp.job%TYPE      INDEX BY BINARY_INTEGER;
    TYPE hiredate_tbl IS TABLE OF emp.hiredate%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl      IS TABLE OF emp.sal%TYPE      INDEX BY BINARY_INTEGER;
    TYPE comm_tbl     IS TABLE OF emp.comm%TYPE     INDEX BY BINARY_INTEGER;
    TYPE deptno_tbl   IS TABLE OF emp.deptno%TYPE   INDEX BY BINARY_INTEGER;
    t_empno           EMPNO_TBL;
    t_ename           ENAME_TBL;
    t_job             JOB_TBL;
    t_hiredate        HIREDATE_TBL;
    t_sal             SAL_TBL;
    t_comm            COMM_TBL;
    t_deptno          DEPTNO_TBL;
BEGIN
    SELECT empno, ename, job, hiredate, sal, comm, deptno BULK COLLECT
      INTO t_empno, t_ename, t_job, t_hiredate, t_sal, t_comm, t_deptno
      FROM emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB       HIREDATE    ' ||
        'SAL       ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '--------   ' || '--------  ------');
    FOR i IN 1..t_empno.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_empno(i) || '   ' ||
            RPAD(t_ename(i),8) || ' ' ||
            RPAD(t_job(i),10) || ' ' ||
            TO_CHAR(t_hiredate(i),'DD-MON-YY') || ' ' ||
            TO_CHAR(t_sal(i),'99,999.99') || ' ' ||
            TO_CHAR(NVL(t_comm(i),0),'99,999.99') || '  ' ||
            t_deptno(i));
    END LOOP;
END;

EMPNO  ENAME    JOB        HIREDATE    SAL        COMM      DEPTNO
-----  -------  ---------  ---------   --------   --------  ------
7369   SMITH    CLERK      17-DEC-80     800.00        .00  20
7499   ALLEN    SALESMAN   20-FEB-81   1,600.00     300.00  30
7521   WARD     SALESMAN   22-FEB-81   1,250.00     500.00  30
7566   JONES    MANAGER    02-APR-81   2,975.00        .00  20
7654   MARTIN   SALESMAN   28-SEP-81   1,250.00   1,400.00  30
7698   BLAKE    MANAGER    01-MAY-81   2,850.00        .00  30
7782   CLARK    MANAGER    09-JUN-81   2,450.00        .00  10
7788   SCOTT    ANALYST    19-APR-87   3,000.00        .00  20
7839   KING     PRESIDENT  17-NOV-81   5,000.00        .00  10
7844   TURNER   SALESMAN   08-SEP-81   1,500.00        .00  30
7876   ADAMS    CLERK      23-MAY-87   1,100.00        .00  20
7900   JAMES    CLERK      03-DEC-81     950.00        .00  30
7902   FORD     ANALYST    03-DEC-81   3,000.00        .00  20
```

```
7934   MILLER   CLERK     23-JAN-82   1,300.00       .00  10
```

The following example produces the same result, but uses an associative array on a record type defined with the %ROWTYPE attribute:

```
DECLARE
    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    t_emp           EMP_TBL;
BEGIN
    SELECT * BULK COLLECT INTO t_emp FROM emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ' ||
        'SAL        ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '--------   ' || '--------  ------');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || '   ' ||
            RPAD(t_emp(i).ename,8) || ' ' ||
            RPAD(t_emp(i).job,10) || ' ' ||
            TO_CHAR(t_emp(i).hiredate,'DD-MON-YY') || ' ' ||
            TO_CHAR(t_emp(i).sal,'99,999.99') || ' ' ||
            TO_CHAR(NVL(t_emp(i).comm,0),'99,999.99') || '  ' ||
            t_emp(i).deptno);
    END LOOP;
END;

EMPNO  ENAME    JOB        HIREDATE    SAL        COMM     DEPTNO
-----  -------  ---------  ---------   --------   --------  ------
7369   SMITH    CLERK      17-DEC-80     800.00        .00  20
7499   ALLEN    SALESMAN   20-FEB-81   1,600.00     300.00  30
7521   WARD     SALESMAN   22-FEB-81   1,250.00     500.00  30
7566   JONES    MANAGER    02-APR-81   2,975.00        .00  20
7654   MARTIN   SALESMAN   28-SEP-81   1,250.00   1,400.00  30
7698   BLAKE    MANAGER    01-MAY-81   2,850.00        .00  30
7782   CLARK    MANAGER    09-JUN-81   2,450.00        .00  10
7788   SCOTT    ANALYST    19-APR-87   3,000.00        .00  20
7839   KING     PRESIDENT  17-NOV-81   5,000.00        .00  10
7844   TURNER   SALESMAN   08-SEP-81   1,500.00        .00  30
7876   ADAMS    CLERK      23-MAY-87   1,100.00        .00  20
7900   JAMES    CLERK      03-DEC-81     950.00        .00  30
7902   FORD     ANALYST    03-DEC-81   3,000.00        .00  20
7934   MILLER   CLERK      23-JAN-82   1,300.00        .00  10
```

**FETCH BULK COLLECT**

The BULK COLLECT clause can be used with a FETCH statement. Instead of returning a single row at a time from the result set, FETCH BULK COLLECT will return all rows at a time from the result set into the specified collection unless restricted by the LIMIT clause.

```
FETCH name BULK COLLECT INTO collection [, ...] [ LIMIT n ];
```

If a single collection is specified, collection may be a collection of a single field or it may be a collection of a record type. If more than one collection is specified, each collection must consist of a single field. The expressions in the SELECT list of the cursor identified by name must match in number, order, and type-compatibility all fields in the target collections. If

LIMIT n is specified, the number of rows returned into the collection on each FETCH will not exceed n.

The following example uses the FETCH BULK COLLECT statement to retrieve rows into an associative array:

```
DECLARE
    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    t_emp           EMP_TBL;
    CURSOR emp_cur IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur;
    FETCH emp_cur BULK COLLECT INTO t_emp;
    CLOSE emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB       HIREDATE    ' ||
        'SAL       ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '--------   ' || '--------  ------');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || '   ' ||
            RPAD(t_emp(i).ename,8) || ' ' ||
            RPAD(t_emp(i).job,10) || ' ' ||
            TO_CHAR(t_emp(i).hiredate,'DD-MON-YY') || ' ' ||
            TO_CHAR(t_emp(i).sal,'99,999.99') || ' ' ||
            TO_CHAR(NVL(t_emp(i).comm,0),'99,999.99') || '  ' ||
            t_emp(i).deptno);
    END LOOP;
END;

EMPNO  ENAME    JOB       HIREDATE    SAL       COMM     DEPTNO
-----  -------  ---------  ---------   --------   --------  ------
7369   SMITH    CLERK      17-DEC-80     800.00       .00  20
7499   ALLEN    SALESMAN   20-FEB-81   1,600.00     300.00  30
7521   WARD     SALESMAN   22-FEB-81   1,250.00     500.00  30
7566   JONES    MANAGER    02-APR-81   2,975.00       .00  20
7654   MARTIN   SALESMAN   28-SEP-81   1,250.00   1,400.00  30
7698   BLAKE    MANAGER    01-MAY-81   2,850.00       .00  30
7782   CLARK    MANAGER    09-JUN-81   2,450.00       .00  10
7788   SCOTT    ANALYST    19-APR-87   3,000.00       .00  20
7839   KING     PRESIDENT  17-NOV-81   5,000.00       .00  10
7844   TURNER   SALESMAN   08-SEP-81   1,500.00       .00  30
7876   ADAMS    CLERK      23-MAY-87   1,100.00       .00  20
7900   JAMES    CLERK      03-DEC-81     950.00       .00  30
7902   FORD     ANALYST    03-DEC-81   3,000.00       .00  20
7934   MILLER   CLERK      23-JAN-82   1,300.00       .00  10
```

### EXECUTE IMMEDIATE BULK COLLECT

The BULK COLLECT clause can be used with a EXECUTE IMMEDIATE statement to specify a collection to receive the returned rows:

```
EXECUTE IMMEDIATE 'sql_expression;'
   BULK COLLECT INTO collection [,...]
   [USING {[bind_type] bind_argument} [, ...]}] ;
```

collection specifies the name of a collection.

bind_type specifies the parameter mode of the bind_argument.

- A bind_type of IN specifies that bind_argument contains a value that is passed to the sql_expression.
- A bind_type of OUT specifies that the bind_argument receives a value from the sql_expression.
- A bind_type of IN OUT specifies that the bind_argument is passed to sql_expression, and then stores the value returned by sql_expression.

bind_argument specifies a parameter that contains a value that is either passed to the sql_expression (specified with a bind_type of IN), or that receives a value from the sql_expression (specified with a bind_type of OUT), or both (specified with a bind_type of IN OUT).

If a single collection is specified, collection may be a collection of a single field or a collection of a record type. If more than one collection is specified, each collection must consist of a single field.

**RETURNING BULK COLLECT**

The BULK COLLECT clause can be added to the RETURNING INTO clause of a DELETE, INSERT, or UPDATE statement:

```
{ insert | update | delete }
  RETURNING { * | expr_1 [, expr_2 ] ...}
    BULK COLLECT INTO collection [, ...] ;
```

insert, update, and delete are the INSERT, UPDATE, and DELETE statements as described in INSERT, UPDATE, and DELETE respectively. If a single collection is specified, collection may be a collection of a single field or it may be a collection of a record type. If more than one collection is specified, each collection must consist of a single field. The expressions following the RETURNING keyword must match in number, order, and type-compatibility all fields in the target collections. If * is specified, all columns in the affected table are returned. ( Note that the use of * is an extension for PolarDB database compatible with Oracle and is not compatible with Oracle databases.)

The clerkemp table created by copying the emp table is used in the remaining examples in this topic.

```
CREATE TABLE clerkemp AS SELECT * FROM emp WHERE job = 'CLERK';

SELECT * FROM clerkemp;

 empno | ename  |  job  | mgr  |      hiredate      |   sal   | comm | deptno
-------+--------+-------+------+--------------------+---------+------+--------
  7369 | SMITH  | CLERK | 7902 | 17-DEC-80 00:00:00 |  800.00 |      |     20
  7876 | ADAMS  | CLERK | 7788 | 23-MAY-87 00:00:00 | 1100.00 |      |     20
  7900 | JAMES  | CLERK | 7698 | 03-DEC-81 00:00:00 |  950.00 |      |     30
  7934 | MILLER | CLERK | 7782 | 23-JAN-82 00:00:00 | 1300.00 |      |     10
(4 rows)
```

The following example increases everyone's salary by 1.5, stores the employees' numbers , names, and new salaries in three associative arrays, and finally displays the contents of these arrays:

```
DECLARE
    TYPE empno_tbl IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
    TYPE ename_tbl IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl   IS TABLE OF emp.sal%TYPE   INDEX BY BINARY_INTEGER;
    t_empno         EMPNO_TBL;
    t_ename         ENAME_TBL;
    t_sal           SAL_TBL;
BEGIN
    UPDATE clerkemp SET sal = sal * 1.5 RETURNING empno, ename, sal
        BULK COLLECT INTO t_empno, t_ename, t_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      SAL       ');
    DBMS_OUTPUT.PUT_LINE('-----  -------    --------   ');
    FOR i IN 1..t_empno.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_empno(i) || '   ' || RPAD(t_ename(i),8) ||
            ' ' || TO_CHAR(t_sal(i),'99,999.99'));
    END LOOP;
END;

EMPNO  ENAME      SAL
-----  -------    --------
7369   SMITH      1,200.00
7876   ADAMS      1,650.00
7900   JAMES      1,425.00
7934   MILLER     1,950.00
```

The following example performs the same functionality as the previous example, but uses a single collection defined with a record type to store the employees' numbers, names, and new salaries:

```
DECLARE
    TYPE emp_rec IS RECORD (
        empno       emp.empno%TYPE,
        ename       emp.ename%TYPE,
         sal        emp.sal%TYPE
    );
    TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
    t_emp           EMP_TBL;
BEGIN
    UPDATE clerkemp SET sal = sal * 1.5 RETURNING empno, ename, sal
        BULK COLLECT INTO t_emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      SAL       ');
    DBMS_OUTPUT.PUT_LINE('-----  -------    --------   ');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || '   ' ||
            RPAD(t_emp(i).ename,8) || ' ' ||
            TO_CHAR(t_emp(i).sal,'99,999.99'));
    END LOOP;
END;

EMPNO  ENAME      SAL
-----  -------    --------
```

```
7369   SMITH      1,200.00
7876   ADAMS      1,650.00
7900   JAMES      1,425.00
7934   MILLER     1,950.00
```

The following example deletes all rows from the clerkemp table and returns information on the deleted rows into an associative array, which is then displayed.

```
DECLARE
    TYPE emp_rec IS RECORD (
        empno       emp.empno%TYPE,
        ename       emp.ename%TYPE,
        job         emp.job%TYPE,
        hiredate    emp.hiredate%TYPE,
        sal         emp.sal%TYPE,
        comm        emp.comm%TYPE,
        deptno      emp.deptno%TYPE
    );
    TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
    r_emp           EMP_TBL;
BEGIN
    DELETE FROM clerkemp RETURNING empno, ename, job, hiredate, sal,
        comm, deptno BULK COLLECT INTO r_emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ' ||
        'SAL        ' || 'COMM       DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '--------   ' || '--------  ------');
    FOR i IN 1..r_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(r_emp(i).empno || '   ' ||
            RPAD(r_emp(i).ename,8) || ' ' ||
            RPAD(r_emp(i).job,10) || ' ' ||
            TO_CHAR(r_emp(i).hiredate,'DD-MON-YY') || ' ' ||
            TO_CHAR(r_emp(i).sal,'99,999.99') || ' ' ||
            TO_CHAR(NVL(r_emp(i).comm,0),'99,999.99') || '  ' ||
            r_emp(i).deptno);
    END LOOP;
END;

EMPNO  ENAME    JOB        HIREDATE    SAL        COMM       DEPTNO
-----  -------  ---------  ---------   --------   --------  ------
7369   SMITH    CLERK      17-DEC-80   1,200.00        .00  20
7876   ADAMS    CLERK      23-MAY-87   1,650.00        .00  20
7900   JAMES    CLERK      03-DEC-81   1,425.00        .00  30
7934   MILLER   CLERK      23-JAN-82   1,950.00        .00  10
```

## 7.17.5 Errors and messages

You can use the DBMS_OUTPUT.PUT_LINE statement to report messages.

```
DBMS_OUTPUT.PUT_LINE ( message );
```

message is any expression evaluating to a string.

This example displays the message on the output display of the user:

```
DBMS_OUTPUT.PUT_LINE('My name is John');
```

The special variables SQLCODE and SQLERRM contain a numeric code and a text message that describe the outcome of the last SQL statement issued. If any other error occurs in the program such as division by zero, these variables contain information pertaining to the error.

# 8 Triggers

## 8.1 Overview of triggers

A trigger is a named Structured Process Language (SPL) code block that is associated with a table and stored in the database. When a specified event occurs on the associated table, the SPL code block is executed. The trigger is considered fired when the code block is executed.

The event that causes a trigger to fire can be any combination of an insert, update, or delete carried out on the table, either directly or indirectly. If the table is the object of an SQL INSERT, UPDATE, or DELETE statement, the trigger is directly fired assuming that the corresponding insert, update, or delete event is defined as a triggering event. The events that fire the trigger are defined in the CREATE TRIGGER statement.

A trigger can be fired indirectly if a triggering event occurs on the table as a result of an event initiated on another table. For example, if a trigger is defined on a table containing a foreign key defined with the ON DELETE CASCADE clause and a row in the parent table is deleted, all children of the parent will also be deleted. If deletion is a triggering event on the child table, deletion of the children will cause the trigger to fire.

## 8.2 Types of triggers

PolarDB databases compatible with Oracle support both row-level and statement-level triggers. A row-level trigger fires once for each row that is affected by a triggering event. For example, if deletion is defined as a triggering event on a table and a single DELETE statement is executed to delete five rows from the table, then the trigger will be fired five times, once for each row.

In contrast, a statement-level trigger is fired once for each triggering statement regardless of the number of rows affected by the triggering event. In the preceding example, a single DELETE statement deletes five rows, and a statement-level trigger will be fired only once.

For statement-level triggers, the sequence of actions can be defined. The sequence refers to whether the trigger code block is executed before or after the triggering statement itself. For row-level triggers, before or after each row is affected by the triggering statement.

In a before row-level trigger, the trigger code block is executed before the triggering action is carried out on each affected row. In a before statement-level trigger, the trigger code block is executed before the action of the triggering statement is carried out.

In an after row-level trigger, the trigger code block is executed after the triggering action is carried out on each affected row. In an after statement-level trigger, the trigger code block is executed after the action of the triggering statement is carried out.

## 8.3 Create a trigger

You can use the CREATE TRIGGER statement to define and name a trigger that will be stored in the database.

### Syntax

Define a new trigger.

```
CREATE TRIGGER
```

### Synopsis

```
CREATE [ OR REPLACE ] TRIGGER name
 { BEFORE | AFTER | INSTEAD OF }
 {INSERT | UPDATE | DELETE}
   [ OR { INSERT | UPDATE | DELETE } ] [, ...]
  ON table
 [ REFERENCING { OLD AS old | NEW AS new } ...]
 [ FOR EACH ROW ]
 [ WHEN condition ]
 [ DECLARE
   [ PRAGMA AUTONOMOUS_TRANSACTION; ]
   declaration; [, ...] ]
  BEGIN
   statement; [, ...]
 [ EXCEPTION
  { WHEN exception [ OR exception ] [...] THEN
    statement; [, ...] } [, ...]
 ]
  END
```

### Description

CREATE TRIGGER defines a new trigger. CREATE OR REPLACETRIGGER creates a trigger or replaces an existing definition.

If you are using the CREATE TRIGGER statement to create a trigger, the name of the new trigger must not match any existing trigger defined on the same table. New triggers are created in the same schema as the table on which the triggering event is defined.

If you are updating the definition of an existing trigger, use the CREATEOR REPLACE TRIGGER statement.

When you use the syntax compatible with Oracle databases to create a trigger, the trigger runs as a SECURITY DEFINER function.

**Parameters**

| Parameter | Description |
|---|---|
| name | The name of the trigger that you want to create. |
| BEFORE \| AFTER | Determines whether the trigger is fired before or after the triggering event. |
| INSERT \| UPDATE \| DELETE | Defines the triggering event. |
| table | The name of the table or view on which the triggering event occurs . |
| condition | A Boolean expression that determines if the trigger will actually be executed. If condition evaluates to TRUE, the trigger is fired.<br>If the trigger definition includes the FOR EACH ROW keywords, the WHEN clause can reference columns of the old and/or new row values after you write OLD.column_name or NEW.column_name , respectively. INSERT triggers cannot reference OLD, and DELETE triggers cannot reference NEW.<br>If a trigger contains the keywords INSTEAD OF, it may not contain the WHEN clause.<br>WHEN clauses cannot contain subqueries. |

| Parameter | Description |
|---|---|
| REFERENCING { OLD AS old \| NEW AS new } ... | The REFERENCING clause to reference old rows and new rows, but restricted in that old may only be replaced by an identifier named old or any equivalent that is saved in all lowercase, for example, REFERENCING OLD AS old, REFERENCING OLD AS OLD, or REFERENCING OLD AS "old". Also, new may only be replaced by an identifier named new or any equivalent that is saved in all lowercase, for example, REFERENCING NEW AS new, REFERENCING NEW AS NEW, or REFERENCING NEW AS "new".<br>Either one or both phrases OLD AS old and NEW AS new may be specified in the REFERENCING clause, for example, REFERENCING NEW AS New OLD AS Old.<br>This clause is not compatible with Oracle databases in that identifiers other than old or new may not be used. |
| FOR EACH ROW | Determines whether the trigger should be fired once for every row affected by the triggering event or only once per SQL statement. If it is specified, the trigger is fired once for every affected row (row-level trigger). Otherwise, the trigger is a statement-level trigger. |
| PRAGMA AUTONOMOUS _TRANSACTION | The directive that sets the trigger as an autonomous transaction. |
| declaration | A variable, type, REF CURSOR, or subprogram declaration. If subprogram declarations are included, they must be declared after all other variable, type, and REF CURSOR declarations. |
| statement | A Structured Process Language (SPL) program statement. Note that a DECLARE - BEGIN - END block is considered an SPL statement. Therefore, the trigger body may contain nested blocks. |
| exception | The name of an exception condition, such as NO_DATA_FOUND and OTHERS. |

## 8.4 Trigger variables

In the trigger code block, several special variables are available for use.

**NEW**

NEW is a pseudo-record name that references the new table row for insert and update operations in row-level triggers. This variable is not applicable in statement-level triggers or in delete operations of row-level triggers.

This variable is used in the form of :NEW.column. In :NEW.column, column is the name of a column in the table on which the trigger is defined.

The initial content of :NEW.column is the value in the named column of the new row to be inserted or of the new row that is to replace the old one when it is used in a before row-level trigger. When used in an after row-level trigger, this value has been stored in the table because the action has occurred on the affected row.

In the trigger code block, :NEW.column can be used like any other variable. If a value is assigned to :NEW.column, in the code block of a before row-level trigger, the assigned value will be used in the new inserted or updated row.

**OLD**

OLD is a pseudo-record name that refers to the old table row for update and delete operations in row-level triggers. This variable is not applicable in statement-level triggers or in insert operations of row-level triggers.

This variable is used in the form of :OLD.column. In :OLD.column, column is the name of a column in the table on which the trigger is defined.

The initial content of :OLD.column is the value in the named column of the row to be deleted or of the old row that is to be replaced by the new one when it is used in a before row-level trigger. When it is used in an after row-level trigger, this value is no longer stored in the table because the action has occurred on the affected row.

In the trigger code block, :OLD.column can be used like other variables. Assigning a value to :OLD.column has no impact on the action of the trigger.

**INSERTING**

INSERTING is a conditional expression that returns TRUE if an insert operation fires the trigger. Otherwise, it returns FALSE.

**UPDATING**

UPDATING is a conditional expression that returns TRUE if an update operation fires the trigger. Otherwise, it returns FALSE.

**DELETING**

DELETING is a conditional expression that returns TRUE if a delete operation fires the trigger . Otherwise, it returns FALSE.

## 8.5 Transactions and exceptions

A trigger is always executed as part of the same transaction within which the triggering statement is being executed. When no exception occurs within the trigger code block, the effects of any triggering command within the trigger are committed only if the transaction containing the triggering statement is committed. Therefore, if the transaction is rolled back, the effect of any DML command in the trigger will also be rolled back.

If an exception occurs within the trigger code block but it is caught and handled in an exception section, the effect of any triggering commands within the trigger is still rolled back. However, the triggering statement itself is not rolled back unless the application forces a rollback of the encapsulating transaction.

If an exception within the trigger code block is not handled, the transaction that encapsulates the trigger is aborted and rolled back. Therefore, the effects of any DML commands within the trigger and the triggering statement are all rolled back.

## 8.6 Trigger examples

### 8.6.1 Before statement-level trigger

The following example is a simple before statement-level trigger that displays a message prior to an insert operation on the emp table.

```
CREATE OR REPLACE TRIGGER emp_alert_trig
    BEFORE INSERT ON emp
BEGIN
    DBMS_OUTPUT.PUT_LINE('New employees are about to be added');
END;
```

The following INSERT is constructed so that new rows are inserted upon a single execution of the command. For each row that has an employee ID between 7900 and 7999, a new row is inserted with an employee ID incremented by 1000. The following example shows the results of executing the command with three new rows inserted.

```
INSERT INTO emp (empno, ename, deptno) SELECT empno + 1000, ename, 40
    FROM emp WHERE empno BETWEEN 7900 AND 7999;
New employees are about to be added

SELECT empno, ename, deptno FROM emp WHERE empno BETWEEN 8900 AND 8999;

     EMPNO ENAME          DEPTNO
---------- ---------- ----------
      8900 JAMES              40
      8902 FORD               40
    8934 MILLER          40
```

The message "New employees are about to be added" is displayed once after the trigger is fired even though the result is that three new rows have been added.

## 8.6.2 After statement-level trigger

The following example is an after statement-level trigger. Whenever an insert, update, or delete operation occurs on the emp table, a row is added to the empauditlog table recording the date, user, and action.

```
CREATE TABLE empauditlog (
    audit_date      DATE,
    audit_user      VARCHAR2(20),
    audit_desc      VARCHAR2(20)
);
CREATE OR REPLACE TRIGGER emp_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_action        VARCHAR2(20);
BEGIN
    IF INSERTING THEN
        v_action := 'Added employee(s)';
    ELSIF UPDATING THEN
        v_action := 'Updated employee(s)';
    ELSIF DELETING THEN
        v_action := 'Deleted employee(s)';
    END IF;
    INSERT INTO empauditlog VALUES (SYSDATE, USER,
        v_action);
END;
```

In the following sequence of statements, two rows are inserted into the emp table by using two INSERT statements. The sal and comm columns of both rows are updated by using one UPDATE statement. Finally, both rows are deleted by using one DELETE statement.

```
INSERT INTO emp VALUES (9001,'SMITH','ANALYST',7782,SYSDATE,NULL,NULL,10);

INSERT INTO emp VALUES (9002,'JONES','CLERK',7782,SYSDATE,NULL,NULL,10);

UPDATE emp SET sal = 4000.00, comm = 1200.00 WHERE empno IN (9001, 9002);

DELETE FROM emp WHERE empno IN (9001, 9002);

SELECT TO_CHAR(AUDIT_DATE,'DD-MON-YY HH24:MI:SS') AS "AUDIT DATE",
    audit_user, audit_desc FROM empauditlog ORDER BY 1 ASC;

AUDIT DATE        AUDIT_USER          AUDIT_DESC
------------------ -------------------- --------------------
31-MAR-05 14:59:48 SYSTEM             Added employee(s)
31-MAR-05 15:00:07 SYSTEM             Added employee(s)
31-MAR-05 15:00:19 SYSTEM             Updated employee(s)
```

```
31-MAR-05 15:00:34 SYSTEM           Deleted employee(s)
```

The contents of the empauditlog table show the times the trigger is fired: once each for the two inserts, once for the update even though two rows are changed, and once for the deletion even though two rows are deleted.

## 8.6.3 Before row-level trigger

The following example is a before row-level trigger that calculates the commission of every new employee belonging to department 30 that is inserted into the emp table.

```
CREATE OR REPLACE TRIGGER emp_comm_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
BEGIN
    IF :NEW.deptno = 30 THEN
        :NEW.comm := :NEW.sal * .4;
    END IF;
END;
```

The list following the addition of the two employees shows that the trigger computed their commissions and inserted it as part of the new employee rows.

```
INSERT INTO emp VALUES (9005,'ROBERS','SALESMAN',7782,SYSDATE,3000.00,NULL,30);

INSERT INTO emp VALUES (9006,'ALLEN','SALESMAN',7782,SYSDATE,4500.00,NULL,30);

SELECT * FROM emp WHERE empno IN (9005, 9006);

     EMPNO ENAME      JOB             MGR HIREDATE        SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ---------- ----------
      9005 ROBERS     SALESMAN        7782 01-APR-05       3000       1200         30
      9006 ALLEN      SALESMAN        7782 01-APR-05       4500       1800         30
```

## 8.6.4 After row-level trigger

The following example is an after row-level trigger. When a new employee row is inserted , the trigger adds a new row to the jobhist table for that employee. When an existing employee row is updated, the trigger sets the enddate column of the latest jobhist row ( assumed to be the one with a null enddate) to the current date and inserts a new jobhist row with the employee's new information.

Finally, the trigger adds a row to the empchglog table with a description of the action.

```
CREATE TABLE empchglog (
    chg_date        DATE,
    chg_desc        VARCHAR2(30)
);
CREATE OR REPLACE TRIGGER emp_chg_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW
DECLARE
```

```
    v_empno        emp.empno%TYPE;
    v_deptno       emp.deptno%TYPE;
    v_dname        dept.dname%TYPE;
    v_action       VARCHAR2(7);
    v_chgdesc      jobhist.chgdesc%TYPE;
BEGIN
    IF INSERTING THEN
        v_action := 'Added';
        v_empno := :NEW.empno;
        v_deptno := :NEW.deptno;
        INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
            :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, 'New Hire');
    ELSIF UPDATING THEN
        v_action := 'Updated';
        v_empno := :NEW.empno;
        v_deptno := :NEW.deptno;
        v_chgdesc := '';
        IF NVL(:OLD.ename, '-null-') ! = NVL(:NEW.ename, '-null-') THEN
            v_chgdesc := v_chgdesc || 'name, ';
        END IF;
        IF NVL(:OLD.job, '-null-') ! = NVL(:NEW.job, '-null-') THEN
            v_chgdesc := v_chgdesc || 'job, ';
        END IF;
        IF NVL(:OLD.sal, -1) ! = NVL(:NEW.sal, -1) THEN
            v_chgdesc := v_chgdesc || 'salary, ';
        END IF;
        IF NVL(:OLD.comm, -1) ! = NVL(:NEW.comm, -1) THEN
            v_chgdesc := v_chgdesc || 'commission, ';
        END IF;
        IF NVL(:OLD.deptno, -1) ! = NVL(:NEW.deptno, -1) THEN
            v_chgdesc := v_chgdesc || 'department, ';
        END IF;
        v_chgdesc := 'Changed ' || RTRIM(v_chgdesc, ', ');
        UPDATE jobhist SET enddate = SYSDATE WHERE empno = :OLD.empno
            AND enddate IS NULL;
        INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
            :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, v_chgdesc);
    ELSIF DELETING THEN
        v_action := 'Deleted';
        v_empno := :OLD.empno;
        v_deptno := :OLD.deptno;
    END IF;

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;
```

In the first sequence of the following statements, two employees are added by using two separate INSERT statements. Then, both are updated by using a single UPDATE statement. The contents of the jobhist table show the action of the trigger for each affected row: two new hire entries for the two new employees and two changed commission records for the updated commissions on the two employees. The empchglog table also shows that the trigger is fired a total of four times, once for each action on the two rows.

```
INSERT INTO emp VALUES (9003,'PETERS','ANALYST',7782,SYSDATE,5000.00,NULL,40);

INSERT INTO emp VALUES (9004,'AIKENS','ANALYST',7782,SYSDATE,4500.00,NULL,40);

UPDATE emp SET comm = sal * 1.1 WHERE empno IN (9003, 9004);
```

```
SELECT * FROM jobhist WHERE empno IN (9003, 9004);

    EMPNO STARTDATE ENDDATE   JOB           SAL      COMM     DEPTNO CHGDESC
---------- --------- --------- --------- ---------- ---------- ---------- -------------
      9003 31-MAR-05 31-MAR-05 ANALYST        5000                  40 New Hire
      9004 31-MAR-05 31-MAR-05 ANALYST        4500                  40 New Hire
      9003 31-MAR-05           ANALYST        5000       5500       40 Changed commission
      9004 31-MAR-05           ANALYST        4500       4950       40 Changed commission

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
```

Finally, both employees are deleted by using a single DELETE statement. The empchglog table shows that the trigger has been fired twice, once for each deleted employee.

```
DELETE FROM emp WHERE empno IN (9003, 9004);

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
31-MAR-05 Deleted employee # 9003
31-MAR-05 Deleted employee # 9004
```

# 9 Object types and objects

## 9.1 Basic object concepts

This topic describes how object-oriented programming techniques can be exploited in Stored Procedure Language (SPL). Object-oriented programming as seen in programming languages such as Java and C++ centers on the concept of objects. An object is a representation of a real-world entity such as a person, place, or thing. The generic description or definition of a particular object, such as a person, is called an object type. Specific people such as "Joe" or "Sally" are objects of the object type person, or equivalently instances of the object type person, or simply person objects.

**Precautions**

The terms database objects and objects that have been used in this document up to this point are different from the object types and objects as used in this topic and other topics under "Object types and objects." The previous usage of these terms relates to the entities that can be created in a database, such as tables, views, indexes, and users. Within the context of topics that are mentioned, object types and objects refer to specific data structures supported by the SPL programming language to implement object-oriented concepts.

In Oracle, the term abstract data type (ADT) is used to describe object types in PL/SQL. The SPL implementation of object types is intended to be compatible with Oracle ADTs.

PolarDB databases compatible with Oracle do not support some features of object-oriented programming languages. This topic only describes the features that have been supported.

**Concepts**

An object type is a description or definition of some entity. This definition of an object type is characterized by two components:

- Attributes: the fields that describe particular characteristics of an object instance. For example, the attributes of a person object may include the name, address, gender, date of birth, height, weight, eye color, and occupation.
- Methods: the programs that perform some type of function or operation on or related to an object. For example, the methods of a person object may include calculating the

person's age, displaying the person's attributes, and changing the values assigned to the person's attributes.

### Attributes

Each object type must contain at least one attribute. The data type of an attribute can be one of the following types:

- A base data type, such as NUMBER and VARCHAR2
- Another object type
- A globally defined collection type (created by the CREATE TYPE statement), such as a nested table or varray

An attribute obtains its initial value when an object instance is initially created. The initial value may be NULL. Each object instance has its own set of attribute values.

### Methods

Methods are SPL procedures or functions defined within an object type. Methods can be categorized into three general types:

- Member methods: the procedures or functions that operate within the context of an object instance. Member methods have access to and can change the attributes of the object instance on which they are operating.
- Static methods: the procedures or functions that operate independently of a particular object instance. Static methods do not have access to and cannot change the attributes of an object instance.
- Constructor methods: the functions used to create an instance of an object type. A default constructor method is always provided when an object type is defined.

### Overloaded methods

In an object type, you cannot define two or more identically named methods of the same type (either a procedure or function) but with different signatures. Such methods are called overloaded methods.

A method's signature consists of the number of formal parameters, the data types of the formal parameters, and their order.

## 9.2 Object type components

You can create and store an object type in a database by using the following two constructs of the Stored Procedure Language (SPL):

- Object type specification: This is the public interface which specifies the attributes and method signatures of the object type.
- Object type body: This contains the implementation of the methods specified in the object type specification.

The following sections describe the statements used to create the object type specification and the object type body.

### Syntax of the object type specification

The syntax of the object type specification is as follows:

```
CREATE [ OR REPLACE ] TYPE name
  [ AUTHID { DEFINER | CURRENT_USER } ]
  { IS | AS } OBJECT
( { attribute { datatype | objtype | collecttype } }
    [, ...]
  [ method_spec ] [, ...]
  [ constructor ] [, ...]
) [ [ NOT ] { FINAL | INSTANTIABLE } ] ... ;
```

where, method_spec is as follows:

```
[ [ NOT ] { FINAL | INSTANTIABLE } ] ...
[ OVERRIDING ]
  subprogram_spec
```

where, subprogram_spec is as follows:

```
{ MEMBER | STATIC }
{ PROCEDURE proc_name
    [ ( [  SELF [ IN | IN OUT ] name ]
        [, parm1 [ IN | IN OUT | OUT ] datatype1
                [ DEFAULT value1 ] ]
        [, parm2 [ IN | IN OUT | OUT ] datatype2
                [ DEFAULT value2 ]
        ] ...)
    ]
|
  FUNCTION func_name
    [ ( [  SELF [ IN | IN OUT ] name ]
        [, parm1 [ IN | IN OUT | OUT ] datatype1
                [ DEFAULT value1 ] ]
        [, parm2 [ IN | IN OUT | OUT ] datatype2
                [ DEFAULT value2 ]
        ] ...)
    ]
  RETURN return_type
}
```

where, constructor is as follows:

```
CONSTRUCTOR func_name
 [ ( [  SELF [ IN | IN OUT ] name ]
     [, parm1 [ IN | IN OUT | OUT ] datatype1
          [ DEFAULT value1 ] ]
     [, parm2 [ IN | IN OUT | OUT ] datatype2
          [ DEFAULT value2 ]
     ] ...)
 ]
RETURN self AS RESULT
```

**Note:**

Currently, the OR REPLACE option cannot be used to add, delete, or modify the attributes of an existing object type. Before you can use this option, you must use the DROP TYPE statement to first delete the existing object type. However, the OR REPLACE option can be used to add, delete, or modify the methods in an existing object type.

The PostgreSQL form of the ALTER TYPE ALTER ATTRIBUTE statement can be used to change the data type of an attribute in an existing object type. However, the ALTER TYPE statement cannot be used to add or delete attributes in the object type.

name is an identifier assigned to the object type. It is optionally schema-qualified.

If the AUTHID clause is omitted or DEFINER is specified, the rights of the object type owner are used to determine access permissions on database objects. If CURRENT_USER is specified, the rights of the current user who is executing a method in the object are used to determine access permissions.

attribute is an identifier assigned to an attribute of the object type.

datatype is a base data type.

objtype is a previously defined object type.

collecttype is a previously defined collection type.

Following the closing parenthesis of the CREATE TYPE definition, [ NOT ] FINAL specifies whether a subtype can be derived from this object type. FINAL is the default value. It means that no subtypes can be derived from this object type. If you want to allow subtypes to be defined under this object type, specify NOT FINAL.

**Note:**

Even though the specification of NOT FINAL is accepted in the CREATE TYPE statement, SPL does not support the creation of subtypes.

Following the closing parenthesis of the CREATE TYPE definition, [ NOT ] INSTANTIABLE specifies whether an object instance of this object type can be created. INSTANTIABLE is the default value. It means that an instance of this object type can be created. If this object type is to be used only as a parent template from which other specialized subtypes are to be defined, specify NOT INSTANTIABLE. If NOT INSTANTIABLE is specified, NOT FINAL must also be specified. If a method in the object type contains the NOT INSTANTIABLE qualifier, the object type must be defined with NOT INSTANTIABLE and NOT FINAL.

**Note:**

Even though the specification of NOT INSTANTIABLE is accepted in the CREATE TYPE statement, SPL does not support the creation of subtypes.

method_spec denotes the specification of a member method or a static method.

Prior to the definition of a method, [ NOT ] FINAL specifies whether the method can be overridden in a subtype. NOT FINAL is the default value. It means that the method can be overridden in a subtype.

If a method overrides an identically named method in a supertype, specify OVERRIDING prior to the definition of the method. The overriding method must have the same number of identically named method parameters with the same data types and parameter modes, in the same order, and the same return type (if the method is a function) as defined in the supertype.

Prior to the definition of a method, [ NOT ] INSTANTIABLE specifies whether the object type definition provides an implementation for the method. If INSTANTIABLE is specified, the CREATE TYPE BODY statement for the object type must specify the implementation of the method. If NOT INSTANTIABLE is specified, the CREATE TYPE BODY statement for the object type must not contain the implementation of the method. In the latter case, assume that a subtype contains the implementation of the method, which overrides the method in this object type. If there are NOT INSTANTIABLE methods in the object type, the object type definition must specify NOT INSTANTIABLE and NOT FINAL following the closing parenthesis of the object type specification. The default value is INSTANTIABLE.

subprogram_spec denotes the specification of a procedure or function and begins with the specification of MEMBER or STATIC. A member subprogram must be invoked with respect to a particular object instance, while a static subprogram is not invoked with respect to an object instance.

proc_name is an identifier of a procedure. If the SELF parameter is specified, name is the object type name given in the CREATE TYPE statement. In this situation, parm1, parm2, ... are the formal parameters of the procedure. datatype1, datatype2, ... are the data types of parm1, parm2, ... respectively. IN, IN OUT, and OUT are possible parameter modes for each formal parameter. If none of them are specified, the default value is IN. value1, value2, ... are default values that may be specified for IN parameters.

You must include the CONSTRUCTOR keyword and function definition to define a constructor.

func_name is an identifier of a function. If the SELF parameter is specified, name is the object type name given in the CREATE TYPE statement. In this situation, parm1, parm2, ... are the formal parameters of the function. datatype1, datatype2, ... are the data types of parm1, parm2, ... respectively. IN, IN OUT, and OUT are possible parameter modes for each formal parameter. If none of them are specified, the default value is IN. value1, value2, ... are default values that may be specified for IN parameters. return_type is the data type of the value that the function returns.

Note the following points about an object type specification:

- There must be at least one attribute defined in the object type.
- There may be none, one, or more methods defined in the object type.
- For each member method, there is an implicit, built-in parameter named SELF, whose data type is that of the object type being defined.

  SELF refers to the object instance that is invoking the method. SELF can be explicitly declared as an IN or IN OUT parameter in the parameter list, for example, as MEMBER FUNCTION (SELF IN OUT object_type ...).

  If SELF is explicitly declared, it must be the first parameter in the parameter list. If SELF is not explicitly declared, its parameter mode defaults to IN OUT for member procedures and to IN for member functions.
- A static method cannot be overridden. OVERRIDING and STATIC cannot be specified together in method_spec.
- A static method must be instantiable. NOT INSTANTIABLE and STATIC cannot be specified together in method_spec.

### Syntax of the object type body

The syntax of the object type body is as follows:

```
CREATE [ OR REPLACE ] TYPE BODY name
  { IS | AS }
  method_spec [...]
  [constructor] [...]
END;
```

where, method_spec is as follows:

```
subprogram_spec
```

where, subprogram_spec is as follows:

```
{ MEMBER | STATIC }
{ PROCEDURE proc_name
    [ ( [  SELF [ IN | IN OUT ] name ]
        [, parm1 [ IN | IN OUT | OUT ] datatype1
                 [ DEFAULT value1 ] ]
        [, parm2 [ IN | IN OUT | OUT ] datatype2
                 [ DEFAULT value2 ]
        ] ...)
    ]
{ IS | AS }
  [ PRAGMA AUTONOMOUS_TRANSACTION; ]
  [ declarations ]
  BEGIN
    statement; ...
[ EXCEPTION
    WHEN ... THEN
      statement; ...]
  END;
|
  FUNCTION func_name
    [ ( [  SELF [ IN | IN OUT ] name ]
        [, parm1 [ IN | IN OUT | OUT ] datatype1
                 [ DEFAULT value1 ] ]
        [, parm2 [ IN | IN OUT | OUT ] datatype2
                 [ DEFAULT value2 ]
        ] ...)
    ]
  RETURN return_type
{ IS | AS }
  [ PRAGMA AUTONOMOUS_TRANSACTION; ]
  [ declarations ]
  BEGIN
    statement; ...
[ EXCEPTION
    WHEN ... THEN
      statement; ...]
  END;
```

where, constructor is as follows:

```
  CONSTRUCTOR func_name
    [ ( [  SELF [ IN | IN OUT ] name ]
        [, parm1 [ IN | IN OUT | OUT ] datatype1
                 [ DEFAULT value1 ] ]
```

```
      [, parm2 [ IN | IN OUT | OUT ] datatype2
            [ DEFAULT value2 ]
      ] ...)
    ]
  RETURN self AS RESULT
{ IS | AS }
  [ declarations ]
  BEGIN
    statement; ...
[ EXCEPTION
    WHEN ... THEN
      statement; ...]
  END;
```

name is an identifier assigned to the object type. It is optionally schema-qualified.

method_spec denotes the implementation of an instantiable method that is specified in the CREATE TYPE statement.

If INSTANTIABLE is specified or omitted in method_spec of the CREATE TYPE statement, there must be a method_spec for this method in the CREATE TYPE BODY statement.

If NOT INSTANTIABLE is specified in method_spec of the CREATE TYPE statement, there must be no method_spec for this method in the CREATE TYPE BODY statement.

subprogram_spec denotes the specification of a procedure or function and begins with the specification of MEMBER or STATIC. The same qualifier as that specified in subprogram_spec of the CREATE TYPE statement must be used.

proc_name is an identifier of a procedure specified in the CREATE TYPE statement. The parameter declarations have the same meaning as described for the CREATE TYPE statement, and must be specified in the CREATE TYPE BODY statement in the same manner as specified in the CREATE TYPE statement.

You must include the CONSTRUCTOR keyword and function definition to define a constructo r.

func_name is an identifier of a function specified in the CREATE TYPE statement. The parameter declarations have the same meaning as described for the CREATE TYPE statement, and must be specified in the CREATE TYPE BODY statement in the same manner as specified in the CREATE TYPE statement. return_type is the data type of the value that the function returns and must match return_type given in the CREATE TYPE statement.

PRAGMA AUTONOMOUS_TRANSACTION is the directive that sets the procedure or function as an autonomous transaction.

declarations are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations.

statement is an SPL program statement.

## 9.3 Create an object type

You can use the CREATE TYPE statement to create an object type specification, and the CREATE TYPE BODY statement to create an object type body. This topic provides examples to illustrate the CREATE TYPE and CREATE TYPE BODY statements.

The following example creates the addr_object_type object type that contains attributes but no methods:

```
CREATE OR REPLACE TYPE addr_object_type AS OBJECT (
    street          VARCHAR2(30),
    city            VARCHAR2(20),
    state           CHAR(2),
    zip             NUMBER(5)
);
```

Since there are no methods in this object type, an object type body is not required. This example creates a composite type, which allows you to treat related objects as a single attribute.

### Member methods

A member method is a function or procedure that is defined within an object type and only can be invoked by using an instance of that type. Member methods have access to and can change the attributes of the object instance on which they are operating.

The following example creates the emp_obj_type object type:

```
CREATE OR REPLACE TYPE emp_obj_type AS OBJECT (
    empno           NUMBER(4),
    ename           VARCHAR2(20),
    addr            ADDR_OBJ_TYPE,
    MEMBER PROCEDURE display_emp(SELF IN OUT emp_obj_type)
);
```

The object type emp_obj_type contains a member method named display_emp. display_emp uses a SELF parameter, which passes the object instance on which the method is invoked.

The data type of a SELF parameter is the same as that of the object type being defined . A SELF parameter always references the instance that is invoking the method. A SELF parameter is the first parameter in a member procedure or function regardless of whether it is explicitly declared in the parameter list.

The following example defines an object type body for emp_obj_type:

```
CREATE OR REPLACE TYPE BODY emp_obj_type AS
    MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_type)
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Employee No   : ' || empno);
        DBMS_OUTPUT.PUT_LINE('Name          : ' || ename);
        DBMS_OUTPUT.PUT_LINE('Street        : ' || addr.street);
        DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || addr.city || ', ' ||
            addr.state || ' ' || LPAD(addr.zip,5,'0'));
    END;
END;
```

You can also use the SELF parameter in an object type body. To illustrate how the SELF parameter is used in the CREATE TYPE BODY statement, you can rewrite the preceding object type body as follows:

```
CREATE OR REPLACE TYPE BODY emp_obj_type AS
    MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_type)
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Employee No   : ' || SELF.empno);
        DBMS_OUTPUT.PUT_LINE('Name          : ' || SELF.ename);
        DBMS_OUTPUT.PUT_LINE('Street        : ' || SELF.addr.street);
        DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || SELF.addr.city || ', ' ||
            SELF.addr.state || ' ' || LPAD(SELF.addr.zip,5,'0'));
    END;
END;
```

Both versions of the emp_obj_type body are equivalent.

**Static methods**

Like a member method, a static method belongs to an object type. A static method, however, is not invoked by an instance of the object type but by using the name of the object type. For example, to invoke a static function named get_count and defined within the emp_obj_type object type, you can write as follows:

```
emp_obj_type.get_count();
```

A static method does not have access to and cannot change the attributes of an object instance. It does not typically work with an instance of the object type.

The following object type specification includes a static function get_dname and a member procedure display_dept:

```
CREATE OR REPLACE TYPE dept_obj_type AS OBJECT (
    deptno          NUMBER(2),
    STATIC FUNCTION get_dname(p_deptno IN NUMBER) RETURN VARCHAR2,
    MEMBER PROCEDURE display_dept
);
```

The object type body for dept_obj_type defines a static function named get_dname and a member procedure named display_dept.

```
CREATE OR REPLACE TYPE BODY dept_obj_type AS
    STATIC FUNCTION get_dname(p_deptno IN NUMBER) RETURN VARCHAR2
    IS
        v_dname     VARCHAR2(14);
    BEGIN
        CASE p_deptno
            WHEN 10 THEN v_dname := 'ACCOUNTING';
            WHEN 20 THEN v_dname := 'RESEARCH';
            WHEN 30 THEN v_dname := 'SALES';
            WHEN 40 THEN v_dname := 'OPERATIONS';
            ELSE v_dname := 'UNKNOWN';
        END CASE;
        RETURN v_dname;
    END;
    MEMBER PROCEDURE display_dept
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Dept No    : ' || SELF.deptno);
        DBMS_OUTPUT.PUT_LINE('Dept Name  : ' ||
            dept_obj_type.get_dname(SELF.deptno));
    END;
END;
```

Within the static function get_dname, references to SELF are not allowed. Since a static function is invoked independently of an object instance, it has no implicit access to any object attribute.

The member procedure display_dept can access the deptno attribute of the object instance passed in the SELF parameter. You do not need to explicitly declare the SELF parameter in the display_dept parameter list.

The last DBMS_OUTPUT.PUT_LINE statement in the display_dept procedure includes a call to the static function get_dname, which is qualified by its object type name dept_obj_type.

## Constructor methods

A constructor method is a function that creates an instance of an object type, typically by assigning values to the members of the object. An object type may define several constructors to accomplish different tasks. A constructor method is a member function invoked with a SELF parameter and its name matches the name of the object type.

For example, if you define an object type named address, each constructor is named address. You may overload a constructor by creating one or more different constructor functions with the same name but with different parameter types.

The SPL compiler provides a default constructor for each object type. The default constructor is a member function. Its name matches the name of the object type and its parameter list matches the object type members in order. The following example creates an object type named address:

```
CREATE TYPE address AS OBJECT
(
  street_address VARCHAR2(40),
  postal_code    VARCHAR2(10),
  city           VARCHAR2(40),
  state          VARCHAR2(2)
)
```

The SPL compiler provides a default constructor with the following signature:

```
CONSTRUCTOR FUNCTION address
(
  street_address VARCHAR2(40),
  postal_code    VARCHAR2(10),
  city           VARCHAR2(40),
  state          VARCHAR2(2)
)
```

The body of the default constructor sets each member to NULL.

If you want to create a custom constructor, declare the constructor by using the keyword constructor in the CREATE TYPE statement and define it in the CREATE TYPE BODY statement. For example, if you want to create a custom constructor for the address object type that computes the city and state given a street_address and postal_code, write as follows:

```
CREATE TYPE address AS OBJECT
(
  street_address VARCHAR2(40),
  postal_code    VARCHAR2(10),
  city           VARCHAR2(40),
  state          VARCHAR2(2),

  CONSTRUCTOR FUNCTION address
   (
     street_address VARCHAR2,
     postal_code VARCHAR2
    ) RETURN self AS RESULT
)
 CREATE TYPE BODY address AS
  CONSTRUCTOR FUNCTION address
   (
     street_address VARCHAR2,
     postal_code VARCHAR2
    ) RETURN self AS RESULT
  IS
    BEGIN
      self.street_address := street_address;
      self.postal_code := postal_code;
      self.city := postal_code_to_city(postal_code);
      self.state := postal_code_to_state(postal_code);
      RETURN;
    END;
END;
```

If you want to create an instance of an object type, you can invoke one of the constructor methods for that object type. For example:

```
DECLARE
  cust_addr address := address('100 Main Street', 02203');
BEGIN
  DBMS_OUTPUT.PUT_LINE(cust_addr.city);  -- displays Boston
  DBMS_OUTPUT.PUT_LINE(cust_addr.state); -- displays MA
END;
```

Custom constructors are typically used to compute member values when they are given incomplete information. The preceding example computes the values for city and state when a postal code is provided.

Custom constructors are also used to enforce business rules that restrict the state of an object. For example, if you define an object type to represent a payment, you can use a custom constructor to ensure that no object of the object type payment can be created with an amount that is NULL, negative, or zero. The default constructor sets payment.amount to NULL. Therefore, you must create a custom constructor whose signature matches the default constructor to prohibit NULL amounts.

## 9.4 Create an object instance

If you want to create an instance of an object type, you must declare a variable of the object type and then initialize the declared object variable. The syntax for declaring an object variable is as follows:

```
object obj_type
```

where, object is the identifier assigned to the object variable, and obj_type is the identifier of the previously defined object type.

After you declare an object variable, you must invoke a constructor method to initialize the object with values. Use the following syntax to invoke the constructor method:

```
[NEW] obj_type ({expr1 | NULL} [, {expr2 | NULL} ] [, ...])
```

where, obj_type is the identifier of the object type's constructor method, and the constructo r method has the same name as the previously declared object type.

expr1, expr2, ... are expressions that are type-compatible with the first attribute of the object type, the second attribute of the object type, and so on. If an attribute is of an object type, the corresponding expression can be NULL, an object initialization expression, or any expression that returns the object type.

The following anonymous block declares and initializes a variable:

```
DECLARE
    v_emp          EMP_OBJ_TYPE;
BEGIN
    v_emp := emp_obj_type (9001,'JONES',
        addr_obj_type('123 MAIN STREET','EDISON','NJ',08817));
END;
```

The variable v_emp is declared with a previously defined object type named EMP_OBJ_TY PE. The body of the block initializes the variable by using the emp_obj_type and addr_obj_t ype constructors.

You can include the NEW keyword when you create an instance of an object in the body of a block. The NEW keyword invokes the object constructor whose signature matches the parameters provided.

The following example declares two variables named mgr and emp. Both the variables are of EMP_OBJ_TYPE. mgr is initialized in the declaration, while emp is initialized to NULL in the declaration and is assigned a value in the body.

```
DECLARE
    mgr  EMP_OBJ_TYPE := (9002,'SMITH',NULL);
    emp  EMP_OBJ_TYPE;
BEGIN
    emp := NEW EMP_OBJ_TYPE (9003,'RAY',NULL);
END;
```

In PolarDB databases compatible with Oracle, you can use the following alternate syntax in place of the constructor method:

```
[ ROW ] ({ expr1 | NULL } [, { expr2 | NULL } ] [, ...])
```

ROW is an optional keyword if you specify two or more expressions within the parenthesis -enclosed, comma-delimited list. If you only specify one expression, you must specify the ROW keyword.

## 9.5 Reference an object

After an object variable is created and initialized, you can reference its individual attributes by using the dot notation of the following form:

```
object.attribute
```

where, object is the identifier assigned to the object variable, and attribute is the identifier of an object type attribute.

If the attribute is of an object type, you must reference it in the following form:

```
object.attribute.attribute_inner
```

attribute_inner is an identifier belonging to the object type to which attribute references in its definition of object.

The following example expands upon the preceding anonymous block to display the values assigned to the emp_obj_type object:

```
DECLARE
    v_emp           EMP_OBJ_TYPE;
BEGIN
    v_emp := emp_obj_type(9001,'JONES',
        addr_obj_type('123 MAIN STREET','EDISON','NJ',08817));
    DBMS_OUTPUT.PUT_LINE('Employee No   : ' || v_emp.empno);
    DBMS_OUTPUT.PUT_LINE('Name          : ' || v_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Street        : ' || v_emp.addr.street);
    DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || v_emp.addr.city || ', ' ||
        v_emp.addr.state || ' ' || LPAD(v_emp.addr.zip,5,'0'));
END;
```

The following information is the output from this anonymous block:

```
Employee No   : 9001
Name          : JONES
Street        : 123 MAIN STREET
City/State/Zip: EDISON, NJ 08817
```

Methods are called in a similar way as attributes.

After an object variable is created and initialized, you can call its member procedures or functions by using the dot notation of the following form:

```
object.prog_name
```

where, object is the identifier assigned to the object variable, and prog_name is the identifier of the procedure or function.

You cannot call static procedures or functions by using an object variable. Instead, you can call them by using an object type name.

```
object_type.prog_name
```

where, object_type is the identifier assigned to the object type, and prog_name is the identifier of the procedure or function.

The results of the preceding anonymous block can be duplicated by calling the member procedure display_emp.

```
DECLARE
    v_emp          EMP_OBJ_TYPE;
BEGIN
    v_emp := emp_obj_type(9001,'JONES',
        addr_obj_type('123 MAIN STREET','EDISON','NJ',08817));
    v_emp.display_emp;
END;
```

The following information is the output from this anonymous block:

```
Employee No   : 9001
Name          : JONES
Street        : 123 MAIN STREET
City/State/Zip: EDISON, NJ 08817
```

The following anonymous block creates an instance of dept_obj_type and calls the member procedure display_dept:

```
DECLARE
    v_dept          DEPT_OBJ_TYPE := dept_obj_type (20);
BEGIN
    v_dept.display_dept;
END;
```

The following information is the output from this anonymous block:

```
Dept No    : 20
Dept Name  : RESEARCH
```

You can directly call the static function defined in dept_obj_type by qualifying it with the object type name as follows:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(dept_obj_type.get_dname(20));
END;
```

```
RESEARCH
```

## 9.6 Delete an object type

The following example shows the syntax for deleting an object type.

```
DROP TYPE objtype;
```

objtype is the identifier of the object type that you want to delete. If the definition of objtype contains attributes that are object types or collection types, these nested object types or collection types must be deleted last.

If an object type body is defined for the object type, the DROP TYPE statement deletes the object type body as well as the object type specification. If you want to recreate the complete object type, both the CREATE TYPE and CREATE TYPE BODY statements must be reissued.

The following example deletes the emp_obj_typ and the addr_obj_typ object types created earlier in this topic. emp_obj_typ must be deleted first because it contains addr_obj_typ within its definition as an attribute.

```
DROP TYPE emp_obj_typ;
DROP TYPE addr_obj_typ;
```

The syntax for deleting an object type body, but not the object type specification is as follows:

```
DROP TYPE BODY objtype;
```

The object type body can be recreated by issuing the CREATE TYPE BODY statement.

The following example deletes only the object type body of the dept_obj_typ.

```
DROP TYPE BODY dept_obj_typ;
```

# 10 dblink_ora

## 10.1 Overview of dblink_ora

dblink_ora provides an OCI-based database link that allows you to run SELECT, INSERT, UPDATE or DELETE statements on the data stored in an Oracle system from within a PolarDB database compatible with Oracle. OCI is short for Oracle Call Interface.

If you want to enable Oracle connectivity, download Oracle's freely available OCI drivers from http://www.oracle.com/technetwork/database/database-technologies/instant-client/overview/index.html.

### Connect to an Oracle database

If the Oracle Instant Client that you download does not include the libclntsh.so library, you must create a symbolic link named libclntsh.so that points to the downloaded version. Navigate to the Instant Client directory and run the following command:

```
ln -s libclntsh.so.version libclntsh.so
```

where, version is the version number of the libclntsh.so library. For example:

```
ln -s libclntsh.so.12.1 libclntsh.so
```

Before you create a link to an Oracle server, you must tell the PolarDB database compatible with Oracle where to find the OCI driver.

Set the LD_LIBRARY_PATH environment variable on Linux or PATH on Windows to the lib directory of the Oracle client installation directory.

For Windows only, you can also set the value of the oracle_home configuration parameter in the postgresql.conf file. The value specified in the oracle_home configuration parameter overrides the Windows PATH environment variable.

The LD_LIBRARY_PATH environment variable on Linux and the PATH environment variable or oracle_home configuration parameter on Windows must be set properly each time you start the PolarDB database compatible with Oracle.

When you use a Linux service script to start the PolarDB database compatible with Oracle , make sure that LD_LIBRARY_PATH is set within the service script so it is in effect when the script invokes the pg_ctl utility to start the PolarDB database compatible with Oracle.

For Windows only, if you want to set the oracle_home configuration parameter in the postgresql.conf file, edit the file by adding the following line:

```
oracle_home = 'lib_directory '
```

Substitute the name of the Windows directory that contains oci.dll for lib_directory.

After you set the oracle_home configuration parameter, you must restart the server for the changes to take effect. You can restart the server from the Windows Services console.

## 10.2 dblink_ora functions and procedures

dblink_ora supports the following functions and procedures:

**dblink_ora_connect()**

The dblink_ora_connect() function establishes a connection to an Oracle database with user-specified connection information. This function comes in two forms. The signature of the first form is as follows:

```
dblink_ora_connect(conn_name, server_name, service_name, user_name, password,
port, asDBA)
```

where,

- conn_name specifies the name of the link.
- server_name specifies the name of the host.
- service_name specifies the name of the service.
- user_name specifies the name you use to connect to the server.
- password specifies the password associated with the username.
- port specifies the port number.

If you want to request SYSDBA permissions on the Oracle server, asDBA is True. This parameter is optional. If it is omitted, the value is FALSE.

The first form of dblink_ora_connect() returns a TEXT value.

The signature of the second form of the dblink_ora_connect() function is as follows:

```
dblink_ora_connect(foreign_server_name, asDBA)
```

where,

foreign_server_name specifies the name of a foreign server.

If you want to request SYSDBA permissions on the Oracle server, asDBA is True. This parameter is optional. If it is omitted, the value is FALSE.

The second form of the dblink_ora_connect() function allows you to use the connection properties of a predefined foreign server when you establish a connection to the server.

Before you invoke the second form of the dblink_ora_connect() function, use the CREATE SERVER statement to store the connection properties for the link to a system table. When you call the dblink_ora_connect() function, substitute the server name specified in the CREATE SERVER statement for the name of the link.

The second form of dblink_ora_connect() returns a TEXT value.

**dblink_ora_status()**

The dblink_ora_status() function returns the database connection status. The signature of dblink_ora_status() is as follows:

```
dblink_ora_status(conn_name)
```

where,

conn_name specifies the name of the link.

If the specified connection is active, the function returns a TEXT value of OK.

**dblink_ora_disconnect()**

The dblink_ora_disconnect() function closes a database connection. The signature of dblink_ora_disconnect() is as follows:

```
dblink_ora_disconnect(conn_name)
```

where,

conn_name specifies the name of the link.

The function returns a TEXT value.

**dblink_ora_record()**

The dblink_ora_record() function retrieves information from a database. The signature of dblink_ora_record() is as follows:

```
dblink_ora_record(conn_name, query_text)
```

where,

- conn_name specifies the name of the link.

- query_text specifies the text of the SQL SELECT statement that will be invoked on the Oracle server.

The function returns a SETOF record.

### dblink_ora_call()

The dblink_ora_call() function executes a non-SELECT statement on an Oracle database and returns a result set. The signature of dblink_ora_call() is as follows:

```
dblink_ora_call(conn_name, command, iterations)
```

where,

- conn_name specifies the name of the link.
- command specifies the text of the SQL statement that will be invoked on the Oracle server.
- iterations specifies the number of times the statement is executed.

The function returns a SETOF record.

### dblink_ora_exec()

The dblink_ora_exec() procedure executes a DML or DDL statement in a remote database. The signature of dblink_ora_exec() is as follows:

```
dblink_ora_exec(conn_name, command)
```

where,

- conn_name specifies the name of the link.
- command specifies the text of the SQL INSERT, UPDATE, or DELETE statement that will be invoked on the Oracle server.

The function returns a VOID.

### dblink_ora_copy()

The dblink_ora_copy() function copies an Oracle table to a table in a PolarDB database compatible with Oracle. The dblink_ora_copy() function returns a BIGINT value that represents the number of rows copied. The signature of dblink_ora_copy() is as follows:

```
dblink_ora_copy(conn_name, command, schema_name, table_name, truncate, count)
```

where,

- conn_name specifies the name of the link.

- command specifies the text of the SQL SELECT statement that will be invoked on the Oracle server.
- schema_name specifies the name of the target schema.
- table_name specifies the name of the target table.
- truncate specifies whether the server needs to truncate the table prior to copying. Specify TRUE to indicate that the server needs to truncate the table. This parameter is optional. If it is omitted, the value is FALSE.
- count instructs the server to report status information every n records, where n is the number specified. During the execution of the function, the PolarDB database compatible with Oracle raises a notice of severity INFO with each iteration of the count. For example, if FeedbackCount is 10, dblink_ora_copy() raises a notice every 10 records. This parameter is optional. If it is omitted, the value is 0.

## 10.3 Call dblink_ora functions

You can use the dblink_ora_connect() function to establish a connection.

```
SELECT dblink_ora_connect('acctg', 'localhost', 'xe', 'hr', 'pwd', 1521);
```

This example connects to a service named xe running on port 1521 on the localhost with a username of hr and a password of pwd. You can use the connection name acctg to reference this connection when calling other dblink_ora functions.

The following statement uses the dblink_ora_copy() function over a connection named edb_conn. It copies the empid and deptno columns from a table named ora_acctg on an Oracle server to a table named as_acctg located in the public schema of a PolarDB cluster compatible with Oracle. The TRUNCATE option is enforced, and a feedback count of 3 is specified.

```
edb=# SELECT dblink_ora_copy('edb_conn','select empid, deptno FROM ora_acctg', '
public', 'as_acctg', true, 3);
INFO:  Row: 0
INFO:  Row: 3
INFO:  Row: 6
INFO:  Row: 9
INFO:  Row: 12

 dblink_ora_copy
-----------------
 12
```

```
(1 row)
```

The following statement uses the dblink_ora_record() function and the acctg connection to retrieve information from the Oracle server:

```
SELECT * FROM dblink_ora_record( 'acctg', 'SELECT first_name from employees') AS t1(id
VARCHAR);
```

This statement retrieves a list that includes all of the entries in the first_name column of the employees table.

# 11 Data types

## 11.1 Data types

The following table describes the built-in general-purpose data types.

**Table 11-1: Data types**

| Name | Alias | Description |
|---|---|---|
| BLOB | LONG RAW, RAW(n), BYTEA | Binary data |
| BOOLEAN | | Logical Boolean (true/false) |
| CHAR [ (n) ] | CHARACTER [ (n) ] | Fixed-length character string of n characters |
| CLOB | LONG, LONG VARCHAR | Long character string |
| DATE | TIMESTAMP(0) | Date and time to the second |
| DOUBLE PRECISION | FLOAT, FLOAT(25) - FLOAT(53) | Double precision floating-point number |
| INTEGER | INT, BINARY INTEGER, PLS INTEGER | Signed four-byte integer |
| NUMBER | DEC, DECIMAL, NUMERIC | Exact numeric with optional decimal places |
| NUMBER(p [, s ]) | DEC(p [, s ]), DECIMAL(p [, s ]), NUMERIC(p [, s ]) | Exact numeric of maximum precision, p, and optional scale, s |
| REAL | FLOAT(1) - FLOAT(24) | Single precision floating-point number |
| TIMESTAMP [ (p) ] | | Date and time with optional, fractional second precision, p |
| TIMESTAMP [ (p) ] WITH TIME ZONE | | Date and time with optional, fractional second precision, p, and with time zone |
| VARCHAR2(n) | CHAR VARYING(n), CHARACTER VARYING(n), VARCHAR(n) | Variable-length character string with a maximum length of n characters |

| Name | Alias | Description |
|---|---|---|
| XMLTYPE | | XML data |

The following topics describe the data types in details.

## 11.2 Numeric type

Numeric types consist of four-byte integers, four-byte and eight-byte floating-point numbers, and fixed-precision decimals. The following table lists the available types.

**Table 11-2: Numeric types**

| Name | Storage size | Description | Range |
|---|---|---|---|
| BINARY INTEGER | 4 bytes | Signed integer, Alias for INTEGER | -2,147,483,648 to +2, 147,483,647 |
| DOUBLE PRECISION | 8 bytes | Variable-precision, inexact | 15 decimal digits precision |
| INTEGER | 4 bytes | Usual choice for integer | -2,147,483,648 to +2, 147,483,647 |
| NUMBER | Variable | User-specified precision, exact | Up to 1000 digits of precision |
| NUMBER(p [, s ] ) | Variable | Exact numeric of maximum precision, p, and optional scale , s | Up to 1000 digits of precision |
| PLS INTEGER | 4 bytes | Signed integer, Alias for INTEGER | -2,147,483,648 to +2, 147,483,647 |
| REAL | 4 bytes | Variable-precision, inexact | 6 decimal digits precision |
| ROWID | 8 bytes | Signed 8 bit integer. | -9223372036 854775808 to 9223372036 854775807 |

The following sections describe the types in details.

### Integer type

The INTEGER type stores whole numbers without fractional components between the values of -2,147,483,648 and +2,147,483,647. Attempts to store values outside of the allowed range will result in an error.

Columns of the ROWID type holds fixed-length binary data that describes the physical address of a record. ROWID is an unsigned, four-byte INTEGER that stores whole numbers without fractional components between the values of 0 and 4,294,967,295. Attempts to store values outside of the allowed range will result in an error.

### Arbitrary precision number

The NUMBER type can store practically an unlimited number of digits of precision and perform calculations exactly. It is recommended for storing monetary amounts and other quantities where exactness is required. However, the NUMBER type is very slow compared to the floating-point types described in the next section.

The scale of a NUMBER is the count of decimal digits in the fractional part, to the right of the decimal point. The precision of a NUMBER is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Both the precision and the scale of the NUMBER type can be configured. You can use the following syntax to declare a column of type NUMBER:

```
NUMBER(precision, scale)
```

The precision must be positive, the scale zero or positive. The following syntax

```
NUMBER(precision)
```

selects a scale of 0. Specifying NUMBER without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas NUMBER columns with a declared scale will coerce input values to that scale. The SQL standard requires a default scale of 0, for example, coercion to integer precision. For maximum portability, it is best to specify the precision and scale explicitly.

If the precision or scale of a value is greater than the declared precision or scale of a column, the system will attempt to round the value. If the value cannot be rounded to satisfy the declared limits, an error is raised.

### Floating-point type

The REAL and DOUBLE PRECISION data types are inexact, variable-precision numeric types . In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations,

so that storing and printing back out a value may show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed further here, except for the following points:

If you require exact storage and calculations such as for monetary amounts, use the NUMBER type instead.

If you want to do complicated calculations by using these types for anything important, especially if you rely on certain behavior in boundary cases such as infinity and underflow, you must evaluate the implementation carefully.

Comparing two floating-point values for equality may or may not work as expected. On most platforms, the REAL type has a range of at least 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The DOUBLE PRECISION type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding may take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

POLARDB compatible with Oracle also supports the SQL standard notations FLOAT and FLOAT(p) for specifying inexact numeric types. Here, p specifies the minimum acceptable precision in binary digits. POLARDB compatible with Oracle accepts FLOAT(1) to FLOAT(24) as selecting the REAL type, while FLOAT(25) to FLOAT(53) as selecting DOUBLE PRECISION. Values of p that exceed the allowed range draw an error. FLOAT with no precision specified is taken as DOUBLE PRECISION type.

## 11.3 Character type

This topic introduces the general-purpose character types available in POLARDB compatible with Oracle.

**Table 11-3: Character types**

| Name | Description |
|---|---|
| CHAR[(n)] | Fixed-length character string, blank-padded to the size specified by n |
| CLOB | Large variable-length up to 1 GB |
| LONG | Variable unlimited length. |
| NVARCHAR(n) | Variable-length national character string, with limit. |
| NVARCHAR2(n) | Variable-length national character string, with limit. |
| STRING | Alias for VARCHAR2. |
| VARCHAR(n) | Variable-length character string, with limit ( considered deprecated, but supported for compatibility) |
| VARCHAR2(n) | Variable-length character string, with limit |

**Note:**

In the preceding table, n is a positive integer. These types can store strings up to n characters in length. An attempt to assign a value that exceeds the length of n will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length.

### CHAR

If you do not specify a value for n, the default value will be 1. If the string to be assigned is shorter than n, values of the CHAR type will be space-padded to the specified width (n), and will be stored and displayed that way.

Padding spaces are semantically insignificant. That is, trailing spaces are disregarded when comparing two values of type CHAR, and the spaces will be removed when a CHAR value is converted to one of the other string types.

If you explicitly cast an over-length value to a CHAR(n) type, the value will be truncated to n characters without raising an error as specified by the SQL standard.

### VARCHAR, VARCHAR2, NVARCHAR, and NVARCHAR2

If the string to be assigned is shorter than n, values of type VARCHAR, VARCHAR2, NVARCHAR, and NVARCHAR2 will store the shorter string without padding.

**Note:**

Trailing spaces are semantically significant in VARCHAR values.

If you explicitly cast a value to a VARCHAR type, an over-length value will be truncated to n characters without raising an error as specified by the SQL standard.

### CLOB

You can store a large character string in a CLOB type. CLOB is semantically equivalent to VARCHAR2 except no length limit is specified. We recommend that you use a CLOB type if the maximum string length is not known.

**Note:**

The longest possible character string that can be stored in a CLOB type is about 1 GB.

### Note

The storage requirement for data of these types is the actual string plus 1 byte if the string is less than 127 bytes, or 4 bytes if the string is 127 bytes or greater. In the case of CHAR, the padding also requires storage. Long strings are compressed by the system automatically, so the physical requirement on disk may be less. Long values are stored in background tables so they do not interfere with rapid access to the shorter column values.

The database character set determines the character set used to store textual values.

## 11.4 Binary data

This topic introduces the data types that allow storage of binary strings.

**Table 11-4: Binary Large Object**

| Name | Storage size | Description |
|---|---|---|
| BINARY | The length of the binary string. | Fixed-length binary string, with a length between 1 and 8300. |
| BLOB | The actual binary string plus 1 byte if the binary string is less than 127 bytes, or 4 bytes if the binary string is 127 bytes or greater. | Variable-length binary string |
| VARBINARY | The length of the binary string | Variable-length binary string , with a length between 1 and 8300. |

A binary string is a sequence of octets or bytes. Binary strings are distinguished from characters strings by two characteristics: First, binary strings allow storing octets of value zero and other non-printable octets that exceed the range 32 to 126. Second, operations on binary strings process the actual bytes, whereas the encoding and processing of character strings depends on local settings.

## 11.5 Date and time type

This topic introduces the date and time types supported by POLARDB compatible with Oracle.

**Note:**

The following discussion of the date and time types assumes that the edb_redwood_date configuration parameter has been set to TRUE whenever a table is created or altered.

**Table 11-5: Date/Time Types**

| Name | Storage size | Description | Low value | High value | Resolution |
|---|---|---|---|---|---|
| DATE | 8 bytes | Date and time | 4713 BC | 5874897 AD | 1 second |

| Name | Storage size | Description | Low value | High value | Resolution |
|---|---|---|---|---|---|
| INTERVAL DAY TO SECOND [(p)] | 12 bytes | Period of time | -178000000 years | 178000000 years | 1 microsecond / 14 digits |
| INTERVAL YEAR TO MONTH | 12 bytes | Period of time | -178000000 years | 178000000 years | 1 microsecond / 14 digits |
| TIMESTAMP [(p)] | 8 bytes | Date and time | 4713 BC | 5874897 AD | 1 microsecond |
| TIMESTAMP [(p)] WITH TIME ZONE | 8 bytes | Date and time with time zone | 4713 BC | 5874897 AD | 1 microsecond |

When DATE appears as the data type of a column in the data definition language (DDL) statements, CREATE TABLE or ALTER TABLE, it is translated to TIMESTAMP(0) at the time the table definition is stored in the database. Therefore, a time component will also be stored in the column along with the date.

When DATE appears as a data type of a variable in an SPL declaration section, or the data type of a formal parameter in an SPL procedure or an SPL function, or the return type of an SPL function, it is always translated to TIMESTAMP(0) and thus can handle a time component if present.

TIMESTAMP accepts an optional precision value p which specifies the number of fractional digits retained in the seconds field. The valid values of p is from 0 to 6.The default value is 6 .

When TIMESTAMP values are stored as double precision floating-point numbers by default , the effective limit of precision can be less than 6. TIMESTAMP values are stored as seconds before or after midnight January 1, 2000. Microsecond precision is achieved for dates within a few years of 2000-01-01, but the precision degrades for dates further away. When TIMESTAMP values are stored as eight-byte integers (a compile-time option), microsecond precision is available over the full range of values. However, eight-byte integer timestamps have a more limited range of dates than dates listed in the preceding table. It is from 4713 BC up to 294276 AD.

TIMESTAMP (p) WITH TIME ZONE is similar to TIMESTAMP (p), but includes the time zone as well.

### INTERVAL type

INTERVAL values specify a period of time. Values of the INTERVAL type are composed of fields that describe the value of the data. The following table lists the fields allowed in an INTERVAL type:

| Field name | INTERVAL values allowed |
|---|---|
| YEAR | Integer value (positive or negative) |
| MONTH | 0 through 11 |
| DAY | Integer value (positive or negative) |
| HOUR | 0 through 23 |
| MINUTE | 0 through 59 |
| SECOND | 0 through 59.9(p) where 9(p) is the precision of fractional seconds |

The fields must be displayed in descending order, from YEARS to MONTHS, and from DAYS to HOURS, MINUTES and then SECONDS.

POLARDB compatible with Oracle supports two INTERVAL types compatible with Oracle databases.

- The first variation supported by POLARDB compatible with Oracle is INTERVAL DAY TO SECOND [(p)]. INTERVAL DAY TO SECOND [(p)] stores a time interval in days, hours, minutes and seconds.

  **Note:**

p specifies the precision of the second field.

POLARDB compatible with Oracle interprets the value:

-
  ```
  INTERVAL '1 2:34:5.678' DAY TO SECOND(3)
  ```

  as 1 day, 2 hours, 34 minutes, 5 seconds and 678 thousandths of a second.

-
  ```
  INTERVAL '1 23' DAY TO HOUR
  ```

  as 1 day and 23 hours.

-
  ```
  INTERVAL '2:34' HOUR TO MINUTE
  ```

  as 2 hours and 34 minutes.

-
  ```
  INTERVAL '2:34:56.129' HOUR TO SECOND(2)
  ```

  as 2 hours, 34 minutes, 56 seconds and 13 thousandths of a second.

  > **Note:**
  > Note that the fractional second is rounded up to 13 because of the specified precision.

- The second variation supported by POLARDB compatible with Oracle that is compatible with Oracle databases is INTERVAL YEAR TO MONTH. This variation stores a time interval in years and months.

  POLARDB compatible with Oracle interprets the value:

  -
    ```
    INTERVAL '12-3' YEAR TO MONTH
    ```

    as 12 years and 3 months.

  -
    ```
    INTERVAL '456' YEAR(2)
    ```

    as 12 years and 3 months.

  -
    ```
    INTERVAL '300' MONTH
    ```

    as 25 years.

**Date and time input**

Date and time input is accepted in ISO 8601 SQL-compatible format, the Oracle default dd-MON-yy format, and a number of other formats provided that there is no ambiguity as to which component is the year, month, and day. However, we recommend that you use the TO_DATE function to avoid ambiguities.

Any date or time literal input needs to be enclosed in single quotation marks (') in the format of text strings. The following SQL standard syntax is also accepted:

```
type 'value' type
```

**Note:**

- type is either DATE or TIMESTAMP.
- value is a date and time text string.

- **Date**

  The following table describes some input formats for dates, all of which equate to January 8, 1999.

| Example |
|---|
| January 8, 1999 |
| 1999-01-08 |
| 1999-Jan-08 |
| Jan-08-1999 |
| 08-Jan-1999 |
| 08-Jan-99 |
| Jan-08-99 |
| 19990108 |
| 990108 |

  The date values can be assigned to a DATE or TIMESTAMP column or variable. The hour , minute, and seconds fields will be set to zero if the date value is not appended with a time value.

- **Time**

  Some examples of the time component of a date or timestamp are shown in the following table.

**Table 11-6: Time input**

| Example | Description |
|---|---|
| 04:05:06.789 | ISO 8601 |
| 04:05:06 | ISO 8601 |

| Example | Description |
|---|---|
| 04:05 | ISO 8601 |
| 040506 | ISO 8601 |
| 04:05 AM | Same as 04:05; AM does not affect value |
| 04:05 PM | Same as 16:05; input hour must be <= 12 |

- **Timestamp**

  Valid input for timestamps consists of a concatenation of a date and a time. The date portion of the timestamp can be formatted based on the preceding table. The time portion of the timestamp can be formatted based on the preceding table.

  The following example uses the default format of Oracle.

  ```
  08-JAN-99 04:05:06
  ```

  The following example uses the ISO 8601 standard.

  ```
  1999-01-08 04:05:06
  ```

### Date and time output

The default output format of the date and time types will be either (dd-MON-yy) referred to as the Redwood date style, compatible with Oracle databases, or (yyyy-mm-dd) referred to as the ISO 8601 format, depending upon the application interface to the database. Applications that use JDBC such as SQL Interactive always present the date in ISO 8601 form . Other applications such as psql present the date in Redwood form.

The following table lists examples of the output formats for the Redwood and ISO 8601 styles.

**Table 11-7: Date/time output styles**

| Description | Example |
|---|---|
| Redwood style | 31-DEC-05 07:37:16 |
| ISO 8601/SQL standard | 1997-12-17 07:37:16 |

### Internals

POLARDB compatible with Oracle uses Julian dates for all date and time calculations. Julian dates correctly predict or calculate any date after 4713 BC based on the assumption that the length of the year is 365.2425 days.

## 11.6 Boolean type

POLARDB compatible with Oracle provides the standard SQL type BOOLEAN. BOOLEAN can have one of only two states: TRUE or FALSE. A third state, UNKNOWN, is represented by the SQL NULL value.

**Table 11-8: Boolean type**

| Name | Storage size | Description |
|---|---|---|
| BOOLEAN | 1 byte | Logical Boolean (true/false) |

**Note:**

- The valid value for representing the true state is TRUE.
- The valid value for representing the false state is FALSE.

## 11.7 XML type

The XMLTYPE data type is used to store XML data. The advantage over storing XML data in a character field is that it checks whether the input values are well-formed, and there are support functions to perform type-safe operations.

As defined by the XML standard, the XML type can store well-formed documents and content fragments, which are defined by the production XMLDecl? Content in the XML standard. This means that content fragments can have more than one top-level element or character node.

**Note:**

Oracle does not support the storage of content fragments in XMLTYPE columns.

### Examples

The following example shows the creation and insertion of a row into a table with an XMLTYPE column.

```
CREATE TABLE books (
   content       XMLTYPE
);

INSERT INTO books VALUES (XMLPARSE (DOCUMENT '<? xml version="1.0"? ><book><title>
Manual</title><chapter>...</chapter></book>'));

SELECT * FROM books;
```

```
              content
-----------------------------------------------------------
 <book><title>Manual</title><chapter>...</chapter></book>
(1 row)
```

# 12 SQL Commands

## 12.1 Overview

This topic describes all SQL commands that are supported by both PolarDB and Oracle databases. You can run the SQL commands in Oracle database and PolarDB databases compatible with Oracle.

**Note:**

- PolarDB databases compatible with Oracle support other commands that are described in this topic. These commands may not have equivalent Oracle commands. They can provide similar or identical functions to Oracle SQL commands by using different syntax.
- This topic does not describe the complete syntax, options, and functions that are available for each command. In most cases, the syntax, options, and functions that are incompatible with the Oracle database are omitted.
- The PolarDB database documentation provides the document command feature that may not be compatible with Oracle databases.

## 12.2 ALTER INDEX

Modifies an index.

### Syntax

PolarDB databases compatible with Oracle support two variants of the ALTER INDEX command. You can use the first variant to rename an index:

```
ALTER INDEX name RENAME TO new_name
```

You can use the second variant to reconstruct an index.

```
ALTER INDEX name REBUILD
```

### Description

You can use the ALTER INDEX command to modify an index. The RENAME clause allows you to change the name of an index. The REBUILD clause allows you to reconstruct an index and replaces the previous copy of the index with an updated version based on the index table.

You can call the PostgreSQL REINDEX command when using the REBUILD clause. For more information about how to use the REBUILD clause, see the PostgreSQL documentation.

The ALTER INDEX command does not affect stored data.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the index. The name can be schema-qualified. |
| new_name | The new name of the index. |

### Examples

The following example shows how to change the name of an index from name_idx to empname_idx:

```
ALTER INDEX name_idx RENAME TO empname_idx;
```

The following example shows how to reconstruct an index named empname_idx:

```
ALTER INDEX empname_idx REBUILD;
```

## 12.3 ALTER PROCEDURE

### Syntax

```
ALTER PROCEDURE procedure_name options [RESTRICT]
```

### Description

You can use the ALTER PROCEDURE command to specify whether a stored procedure is a SECURITY INVOKER or SECURITY DEFINER.

### Parameters

| Parameter | Description |
|---|---|
| procedure_name | The name of the stored procedure. The name can be schema-qualified. |

| Parameter | Description |
|---|---|
| options | • [EXTERNAL] SECURITY DEFINER<br>Specifies that the server runs the stored procedure by using the privileges of the user who has created the stored procedure. The EXTERNAL keyword is supported for compatibility and is ignored.<br>• [EXTERNAL] SECURITY INVOKER<br>Specifies that the server runs the stored procedure by using the privileges of the user who is calling the stored procedure. The EXTERNAL keyword is supported for compatibility and is ignored.<br>The RESTRICT keyword is supported for compatibility and can be ignored. |

### Examples

The following command specifies that the server runs the update_balance stored procedure by using the privileges of the user who is calling the stored procedure.

```
ALTER PROCEDURE update_balance SECURITY INVOKER;
```

## 12.4 ALTER PROFILE

Modifies a configuration file.

### Syntax

```
ALTER PROFILE profile_name RENAME TO new_name;

ALTER PROFILE profile_name
     LIMIT {parameter value}[...] ;
```

### Description

You can use the ALTER PROFILE command to modify a user-specified configuration file. PolarDB databases compatible with Oracle support the following two types of syntax:

- ALTER PROFILE...RENAME TO: changes the name of a configuration file.

- ALTER PROFILE...LIMIT: modifies the limits that are associated with a configuration file.

You can include the LIMIT clause and one or more space-delimited parameter/value pairs to specify the rules that are enforced by PolarDB databases compatible with Oracle. You can also use the ALTER PROFILE...RENAME TO command to change the name of a configuration file.

**Parameters**

| Parameter | Description |
|---|---|
| profile_name | The name of the configuration file. |
| new_name | The new name of the configuration file. |
| parameter | The parameters in the configuration file. |
| value | The values of the parameters. |

PolarDB databases compatible with Oracle support the following parameter values:

- FAILED_LOGIN_ATTEMPTS specifies the maximum number of failed logon attempts before the server locks the account for the period that is specified by the PASSWORD_LOCK_TIME parameter. Valid values:
  - An INTEGER value greater than 0.
  - DEFAULT: the value of the FAILED_LOGIN_ATTEMPTS parameter that is specified in the DEFAULT configuration file.
  - UNLIMITED: The number of failed logon attempts is unlimited.
- PASSWORD_LOCK_TIME: specifies the required period before the server unlocks an account that has been locked due to excessive logon attempts. Valid values:
  - A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, you must specify a decimal value. For example, you can use the value 4.5 to specify 4 days and 12 hours.
  - DEFAULT: the value of the PASSWORD_LOCK_TIME parameter that is specified in the DEFAULT configuration file.
  - UNLIMITED: The account is locked until it is unlocked by a database superuser.
- PASSWORD_LIFE_TIME: specifies the number of days that the current password can be used before the user is prompted to provide a new password. When using the PASSWORD_LIFE_TIME clause, you can include the PASSWORD_GRACE_TIME clause to specify the number of days after the password expires until connections from the role are rejected. If you do not specify the PASSWORD_GRACE_TIME parameter, the password

expires on the day that is specified by the default value of the PASSWORD_GRACE_TIME parameter. The user is not allowed to run any command until a new password is provided. Valid values:

  - A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, you must specify a decimal value. For example, you can use the value 4.5 to specify 4 days and 12 hours.
  - DEFAULT: the value of the PASSWORD_LIFE_TIME parameter that is specified in the DEFAULT configuration file.
  - UNLIMITED: The password never expires.

- PASSWORD_GRACE_TIME: specifies the grace period after the password expires until the user is required to change the password. After a specified period ends, the user is allowed to connect the server and cannot run any command until the expired password is updated. Valid values:

  - A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, you must specify a decimal value. For example, you can use the value 4.5 to specify 4 days and 12 hours.
  - DEFAULT: the value of the PASSWORD_GRACE_TIME parameter that is specified in the DEFAULT configuration file.
  - UNLIMITED: The grace period is unlimited.

- PASSWORD_REUSE_TIME: specifies the number of days a user must wait before reusing a password. You must use the PASSWORD_REUSE_TIME parameter with the PASSWORD_REUSE_MAX parameter. If you specify a finite value for one parameter and specify UNLIMITED for the other parameter, previous passwords cannot be reused. If you specify UNLIMITED for both parameters, no limit is imposed on password reuse. Valid values:

  - A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, you must specify a decimal value. For example, you can use the value 4.5 to specify 4 days and 12 hours.
  - DEFAULT: the value of the PASSWORD_REUSE_TIME parameter that is specified in the DEFAULT configuration file.
  - UNLIMITED: No limit is imposed on password reuse.

- PASSWORD_REUSE_MAX: specifies the number of password changes that must occur before a password can be reused. You need to use the PASSWORD_REUSE_TIME parameter with the PASSWORD_REUSE_MAX parameter. If you specify a finite value for one parameter and specify UNLIMITED for the other parameter, previous passwords cannot be reused. If you specify UNLIMITED for both parameters, no limit is imposed on password reuse. Valid values:

  - An INTEGER value greater than 0.
  - DEFAULT: the value of the PASSWORD_REUSE_MAX parameter that is specified in the DEFAULT configuration file.
  - UNLIMITED: No limit is imposed on password reuse.

- PASSWORD_VERIFY_FUNCTION: specifies password complexity. Valid values:
  - The name of a PL/SQL function.
  - DEFAULT: the value of the PASSWORD_VERIFY_FUNCTION parameter that is specified in the DEFAULT configuration file.
  - NULL
- PASSWORD_ALLOW_HASHED: specifies whether to allow using an encrypted password. If you set the value to TRUE, the system allows you to change the password by specifying the hash-calculated encrypted password on the client. However, if you set the value to FALSE, you must specify a password in plain-text for verification. Otherwise, an error occurs when the server receives the encrypted password. Valid values:
  - A BOOLEAN value TRUE/ON/YES/1 or FALSE/OFF/NO/0.
  - DEFAULT: the value of the PASSWORD_ALLOW_HASHED parameter that is specified in the DEFAULT configuration file.

**Note:**

The PASSWORD_ALLOW_HASHED parameter is not supported by Oracle.

## Examples

The following example shows how to modify a configuration file named acctg_profile:

```
ALTER PROFILE acctg_profile
     LIMIT FAILED_LOGIN_ATTEMPTS 3 PASSWORD_LOCK_TIME 1;
```

acctg_profile calculates the number of failed connection attempts when a logon role attempts to connect to the server. The configuration file specifies that if a user does not use the correct password for verification in three attempts, the account is locked for one day.

In the following example, the name of the configuration file is changed from acctg_profile to payables_profile:

```
ALTER PROFILE acctg_profile RENAME TO payables_profile;
```

## 12.5 ALTER QUEUE

PolarDB databases compatible with Oracle provide the syntax of the ALTER QUEUE SQL command that is not provided by Oracle. You can use this command with the DBMS_AQADM package.

### Description

You can use the ALTER QUEUE command to modify a queue if you have the aq_administrator_role privilege. This command has the following four types of syntax based on functions.

### Change the name of a queue

You can use the first type of syntax to change the name of a queue. The syntax is as follows :

```
ALTER QUEUE queue_name RENAME TO new_name
```

**Table 12-1: Parameters**

| Parameter | Description |
|---|---|
| queue_name | The name of the queue. The name can be schema-qualified. |
| RENAME TO | The RENAME TO clause that is used to rename the queue. The clause is followed by a new name of the queue. |
| new_name | The new name of the queue. |

### Modify parameters of a queue

You can use the second type of syntax to modify parameters of a queue.

```
ALTER QUEUE queue_name SET [ ( { option_name option_value  } [,SET option_name
```

**Table 12-2: Parameters**

| Parameter | Description |
|---|---|
| queue_name | The name of the queue. The name can be schema-qualified. |

To specify parameters to be modified, you must include the SET clause and option_name/option_value pairs.

```
option_name option_value
```

The names and values of one or more options that are associated with the new queue. If you provide duplicate option names, the server returns an error.

- If the value of the option_name parameter is retries, you must provide an integer that indicates the number of dequeuing attempts.
- If the value of the option_name parameter is retrydelay, you must provide a double-precision value that indicates the delay in seconds.
- If the value of the option_name parameter is retention, you must provide a double-precision value that indicates the retention period in seconds.

### Enable or disable enqueuing and dequeuing

You can use the third type of syntax to enable or disable enqueuing and dequeuing for a queue.

```
ALTER QUEUE queue_name ACCESS { START | STOP } [ FOR { enqueue | dequeue } ] [ NOWAIT
 ]
```

**Table 12-3: Parameters**

| Parameter | Description |
|---|---|
| queue_name | The name of the queue. The name can be schema-qualified. |
| ACCESS | To enable or disable enqueuing and dequeuing for a queue, you must include the ACCESS clause. |

| Parameter | Description |
|---|---|
| START \| STOP | The required state of the queue. |
| FOR enqueue\|dequeue | Specifies whether to enable the enqueuing or dequeuing feature for the queue. |
| NOWAIT | Specifies that the server does not wait for the completion of outstanding transactions before changing the state of the queue. The NOWAIT keyword can be used only if you specify STOP in the ACCESS clause. If you specify START in the ACCESS clause, the server returns an error. |

### Add or remove callback details of a queue

You can use the fourth type of syntax to add or remove callback details of a specified queue.

```
ALTER QUEUE queue_name { ADD | DROP } CALL TO location_name [ WITH callback_option
 ]
```

| Parameter | Description |
|---|---|
| queue_name | The name of the queue. The name can be schema-qualified. |
| ADD \| DROP | Specifies whether to add or remove the callback details of a queue. |
| location_name | The name of the callback stored procedure. |
| callback_option | A valid value of the lback_option parameter is context. You must specify a RAW value when including the callback_option parameter. |

### Examples

In the following example, the name of a queue is changed from work_queue_east to work_order:

```
ALTER QUEUE work_queue_east RENAME TO work_order;
```

The following example shows how to modify a queue named work_order. The number of retries is set to 100, the interval between retries is set to 2 seconds, and the retention period of dequeued messages is set to 10 seconds.

```
ALTER QUEUE work_order SET (retries 100, retrydelay 2, retention 10);
```

The following examples show how to enable enqueuing and dequeuing for a queue named work_order:

```
ALTER QUEUE work_order ACCESS START;
ALTER QUEUE work_order ACCESS START FOR enqueue;
ALTER QUEUE work_order ACCESS START FOR dequeue;
```

The following examples show how to disable enqueuing and dequeuing for a queue named work_order:

```
ALTER QUEUE work_order ACCESS STOP NOWAIT;
ALTER QUEUE work_order ACCESS STOP FOR enqueue;
ALTER QUEUE work_order ACCESS STOP FOR dequeue;
```

## 12.6 ALTER QUEUE TABLE

Modifies a queue table.

### Syntax

You can use the following syntax to modify the name of a queue table:

```
ALTER QUEUE TABLE name RENAME TO new_name
```

### Description

You can use the ALTER QUEUE TABLE command to modify a queue table if you are a superuser or a user who has the aq_administrator_role privilege.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the queue table. The name can be schema-qualified. |

| Parameter | Description |
|---|---|
| new_name | The new name of the queue table. |

### Example

Change the name of a queue table from wo_table_east to work_order_table:

```
ALTER QUEUE TABLE wo_queue_east RENAME TO work_order_table;
```

## 12.7 ALTER ROLE... IDENTIFIED BY

Changes the password that is associated with a database role.

### Syntax

```
ALTER ROLE role_name IDENTIFIED BY password
        [REPLACE prev_password]
```

### Description

You can use the ALTER ROLE... IDENTIFIED BY command to change the password if you are a role without the CREATEROLE privilege. If you use an unauthorized role and PASSWORD_VERIFY_FUNCTION is not NULL in the configuration file, you must include the REPLACE clause and previous password. If a non-superuser uses the REPLACE clause, the server compares the provided password with the existing password. If the passwords do not match, an error occurs.

A database superuser can use this command to change the password that is associated with any role. If a superuser includes the REPLACE clause, this clause is ignored and a non-matching value for the previous password does not generate an error.

If the role whose password is to be changed has the SUPERUSER attribute, only a superuser can run the ALTER ROLE... IDENTIFIED BY command. A role with the CREATEROLE attribute can use this command to change the password that is associated with a non-superuser role .

### Parameters

| Parameter | Description |
|---|---|
| role_name | The name of the role whose password is to be changed. |
| password | The new password of the role. |

| Parameter | Description |
|---|---|
| prev_password | The previous password of the role. |

**Example**

Change the password of the role:

```
ALTER ROLE john IDENTIFIED BY xyRP35z REPLACE 23PJ74a;
```

## 12.8 ALTER ROLE

Manages database link and DBMS_RLS privileges.

PolarDB databases compatible with Oracle provide the syntax of the ALTER ROLE SQL command that is not provided by Oracle. This syntax is useful when you assign privileges of creating and deleting database links that are compatible with Oracle databases, and the DBMS_RLS privilege for fine-grained access control.

**CREATE DATABASE LINK**

A user who has the `CREATE DATABASE LINK` privilege can create a private database link. You can use the following `ALTER ROLE` command to grant a role the privilege to create a private database link:

```
ALTER ROLE role_name
    WITH [CREATEDBLINK | CREATE DATABASE LINK]
```

This command has the same effect as the following command:

```
GRANT CREATE DATABASE LINK to role_name
```

You can use the following command to revoke the privilege:

```
ALTER ROLE role_name
   WITH [NOCREATEDBLINK | NO CREATE DATABASE LINK]
```

**Note:**

The `CREATEDBLINK` and `NOCREATEDBLINK` syntax will be discarded. We recommend that you use the `CREATE DATABASE LINK` and `NO CREATE DATABASE LINK` syntax.

### CREATE PUBLIC DATABASE LINK

A user who has the CREATE PUBLIC DATABASE LINK privilege can create a public database link. You can use the following ALTER ROLE command to grant a role the privilege to create a public database link:

```
ALTER ROLE role_name
  WITH [CREATEPUBLICDBLINK | CREATE PUBLIC DATABASE LINK]
```

This command has the same effect as the following command:

```
GRANT CREATE PUBLIC DATABASE LINK to role_name
```

You can use the following command to revoke the privilege:

```
ALTER ROLE role_name
  WITH [NOCREATEPUBLICDBLINK | NO CREATE PUBLIC DATABASE LINK]
```

**Note:**

The CREATEPUBLICDBLINK and NOCREATEPUBLICDBLINK syntax will be discarded. We recommend that you use the CREATE PUBLIC DATABASE LINK and NO CREATE PUBLIC DATABASE LINK syntax.

### DROP PUBLIC DATABASE LINK

A user who has the DROP PUBLIC DATABASE LINK privilege can delete a public database link. You can use the following ALTER ROLE command to grant a role the privilege to delete a public database link:

```
ALTER ROLE role_name
   WITH [DROPPUBLICDBLINK | DROP PUBLIC DATABASE LINK]
```

This command has the same effect as the following command:

```
GRANT DROP PUBLIC DATABASE LINK to role_name
```

You can use the following command to revoke the privilege:

```
ALTER ROLE role_name
    WITH [NODROPPUBLICDBLINK | NO DROP PUBLIC DATABASE LINK]
```

**Note:**

The DROPPUBLICDBLINK and NODROPPUBLICDBLINK syntax will be discarded. We recommend that you use the DROP PUBLIC DATABASE LINK and NO DROP PUBLIC DATABASE LINK syntax.

### EXEMPT ACCESS POLICY

A user who has the EXEMPT ACCESS POLICY privilege is exempt from fine-grained access control (DBMS_RLS) policies. A user who has the EXEMPT ACCESS POLICY privilege can view or modify any row in a table that is limited by a DBMS_RLS policy. You can use the following ALTER ROLE command to grant a role the EXEMPT ACCESS POLICY privilege so that the role is exempt from defined DBMS_RLS policies.

```
ALTER ROLE role_name
    WITH [POLICYEXEMPT | EXEMPT ACCESS POLICY]
```

This command has the same effect as the following command:

```
GRANT EXEMPT ACCESS POLICY TO role_name
```

You can use the following command to revoke the privilege:

```
ALTER ROLE role_name
  WITH [NOPOLICYEXEMPT | NO EXEMPT ACCESS POLICY]
```

**Note:**

The POLICYEXEMPT and NOPOLICYEXEMPT syntax will be discarded. We recommend that you use the EXEMPT ACCESS POLICY and NO EXEMPT ACCESS POLICY syntax.

## 12.9 ALTER SEQUENCE

Modifies the definition of a sequence generator.

### Syntax

```
ALTER SEQUENCE name [ INCREMENT BY increment ]
  [ MINVALUE minvalue ] [ MAXVALUE maxvalue ]
  [ CACHE cache | NOCACHE ] [ CYCLE ]
```

### Description

You can use the ALTER SEQUENCE command to modify the parameters of a sequence generator. Any parameter that is not specified in the ALTER SEQUENCE command retains its prior setting.

**Note:**

To prevent blocking concurrent transactions that retrieve numbers from the same sequence, rollback does not occur when you run the ALTER SEQUENCE command. The changes take effect immediately and are irreversible.

The ALTER SEQUENCE command does not immediately affect NEXTVAL results in backends (other than the current backend) that have preallocated (cached) sequence values. The system uses cached values before detecting the changed sequence parameters. The current backend is affected immediately.

**Parameters**

| **Parameter** | **Description** |
|---|---|
| name | The name of the sequence to be modified. The name can be schema-qualified. |
| increment | The INCREMENT BY increment clause is optional. A positive value indicates an ascending sequence, and a negative value indicates a descending sequence. If you do not specify this parameter, the old increment value is retained. |
| minvalue | The MINVALUE minvalue clause is optional and specifies the minimum value that a sequence can generate. If you do not specify this parameter, the current minimum value is retained. Note: The NO MINVALUE keyword can be used to specify the default values 1 and-263-1 for ascending and descending orders, respectively. However, this keyword is not compatible with Oracle databases. |
| maxvalue | The MAXVALUE maxvalue clause is optional and specifies the maximum value for the sequence. If you do not specify this parameter, the current maximum value is retained. Note: The NO MAXVALUE keyword can be used to specify the default values 263-1 and -1 for ascending and descending orders, respectively. However, this keyword is not compatible with Oracle databases. |

| Parameter | Description |
|---|---|
| cache | The CACHE cache clause is optional and specifies the number of sequence numbers to be preallocated and stored in memory for fast access. The minimum value is 1, indicating that only one value NOCACHE can be generated at a time. If you do not specify this parameter, the previous cached value is retained. |
| CYCLE | Allows a sequence to wrap around when the ascending sequence reaches the maximum value or descending sequence reaches the minimum value. If the constraint is reached, the next number generated is the value that is specified by the minvalue or maxvalue parameter. If you do not specify this parameter, the previous cycle is retained. Note: The NO CYCLE keyword can be used to specify that the sequence does not recycle. However, this keyword is not compatible with Oracle databases. |

### Example

Modify the increment and cached value of a sequence named serial:

```
ALTER SEQUENCE serial INCREMENT BY 2 CACHE 5;
```

## 12.10 ALTER SESSION

Modifies a runtime parameter.

### Syntax

```
ALTER SESSION SET name = value
```

### Description

You can use the ALTER SESSION command to modify a runtime parameter. ALTER SESSION only changes the value that is used by the current session. Certain parameters are provided only to be compatible with the Oracle syntax and have no impact on the running behavior of PolarDB databases compatible with Oracle. Other parameters change the runtime parameters of PolarDB databases compatible with Oracle.

**Parameters**

| Parameter | Description |
|---|---|
| name | The name of the runtime parameter. The following table lists available parameters. |
| value | The new value of the parameter. |

You can use the ALTERSESSION command to modify the following parameters:

| Parameter | Description |
|---|---|
| NLS_DATE_FORMAT (string) | The display format of the date and time values and the rules for interpreting ambiguous data input values. This parameter has the same effect as the datestyle parameter. |
| NLS_LANGUAGE (string) | The language in which messages are displayed. This parameter has the same effect as the lc_messages parameter. |
| NLS_LENGTH_SEMANTICS (string) | Valid values: BYTE and CHAR. Default value : BYTE. This parameter is provided only for syntax compatibility and has no effect in PolarDB databases compatible with Oracle. |
| OPTIMIZER_MODE (string) | The default query optimization mode. Valid values: ALL_ROWS, CHOOSE, FIRST_ROWS , FIRST_ROWS_10, FIRST_ROWS_100, and FIRST_ROWS_1000. Default value: CHOOSE. This parameter is implemented in PolarDB databases compatible with Oracle. |
| QUERY_REWRITE_ENABLED (string) | Valid values: TRUE, FALSE, and FORCE. Default value: FALSE. This parameter is provided only for syntax compatibility and has no effect in PolarDB databases compatible with Oracle. |
| QUERY_REWRITE_INTEGRITY (string) | Valid values: ENFORCED, TRUSTED, and STALE_TOLERATED. Default value: ENFORCED . This parameter is provided only for syntax compatibility and has no effect in PolarDB databases compatible with Oracle. |

### Examples

Set the language to English (United States) in UTF-8-encoding. Note: In this example, the value en_US.UTF-8 must use the format that you specified for PolarDB databases compatible with Oracle. This format is not compatible with Oracle databases.

```
ALTER SESSION SET NLS_LANGUAGE = 'en_US.UTF-8';
```

Set the date display format:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'dd/mm/yyyy';
```

## 12.11 ALTER TABLE

Changes the definition of a table.

### Syntax

```
ALTER TABLE name
  action [, ...]
ALTER TABLE name
  RENAME COLUMN column TO new_column
ALTER TABLE name
  RENAME TO new_name
```

The action clause has the following types of syntax:

```
ADD column type [ column_constraint [ ... ] ]
DROP COLUMN column
ADD table_constraint
DROP CONSTRAINT constraint_name [ CASCADE ]
```

### Description

You can use the ALTER TABLE command to change the definition of a table. This command has the following clauses:

- ADD column type: adds a new column to the table by using the same syntax as the CREATE TABLE command.
- DROP COLUMN: deletes a column from the table. Indexes and table constraints that involve the column are automatically deleted.
- ADD table_constraint: adds a new constraint to the table by using the same syntax as the CREATE TABLE command.
- DROP CONSTRAINT: deletes the constraints of the table. Table constraints do not need unique names and a specified name can match multiple constraints. All matched constraints are deleted.

- RENAME: changes the name of a table or an individual column in the table. You can also use this type of syntax to change the name of an index, sequence, or view. The stored data is not affected.

Only the owner of a table can use the ALTER TABLE command.

**Note:**

When you use the ADD COLUMN clause, all rows in the table are initialized with the default value of the column. If no DEFAULT clause is specified, the value is null. To add a column with non-null default values, you must rewrite the table. Rewriting a large table is time-consuming and requires twice the disk space. To add a CHECK or NOT NULL constraint, you must scan the table to verify that existing rows meet the constraint.

The DROP COLUMN clause does not physically remove the column, but makes columns invisible to SQL operations. Subsequent insert and update operations in the table store null values for the column. Therefore, deleting a column is fast, but does not immediatel y reduce the disk space that is occupied by the table because the space that is occupied by the deleted column is not reclaimed. The space is reclaimed after existing rows are updated.

You are not allowed to modify any portion of the system directory table. For more information about valid parameters, see the CREATE TABLE topic.

**Parameters**

| Parameter | Description |
|---|---|
| name | The name of the table to be modified. The name can be schema-qualified. |
| column | The name of the new or existing column. |
| new_column | The new name of the existing column. |
| new_name | The new name of the table. |
| type | The data type of the new column. |
| table_constraint | The new constraint of the table. |
| constraint_name | The name of the existing constraint to be deleted. |
| CASCADE | If you specify the CASCADE parameter, the objects that depend on the deleted constraints are automatically deleted. |

### Examples

Add a column of the VARCHAR2 data type to a table:

```
ALTER TABLE emp ADD address VARCHAR2(30);
```

Delete a column from a table:

```
ALTER TABLE emp DROP COLUMN address;
```

Rename an existing column:

```
ALTER TABLE emp RENAME COLUMN address TO city;
```

Rename an existing table:

```
ALTER TABLE emp RENAME TO employee;
```

Add a CHECK constraint to a table:

```
ALTER TABLE emp ADD CONSTRAINT sal_chk CHECK (sal > 500);
```

Delete a CHECK constraint from a table:

```
ALTER TABLE emp DROP CONSTRAINT sal_chk;
```

## 12.12 ALTER TABLESPACE

Changes the definition of a tablespace.

### Syntax

```
ALTER TABLESPACE name RENAME TO newname
```

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the tablespace. |
| newname | The new name of the tablespace. The new name cannot start with the pg_ prefix. This prefix is reserved for system tablespace names. |

### Example

Change the name of a tablespace from empspace to employee_space:

```
ALTER TABLESPACE empspace RENAME TO employee_space;
```

## 12.13 ALTER USER... IDENTIFIED BY

Changes a database user account.

### Syntax

```
ALTER USER role_name IDENTIFIED BY password REPLACE prev_password
```

### Description

You can use the ALTER USER... IDENTIFIED BY command to change the password if you are a role without the CREATEROLE privilege. An unauthorized role must include the REPLACE clause and previous password if PASSWORD_VERIFY_FUNCTION is not NULL in the configuration file. If a non-superuser uses the REPLACE clause, the server compares the provided password with the current password. If the passwords do not match, an error occurs.

### Parameters

| Parameter | Description |
|---|---|
| role_name | The name of the role whose password is to be changed. |
| password | The new password of the role. |
| prev_password | The previous password of the role. |

### Example

Change a user password:

```
ALTER USER john IDENTIFIED BY xyRP35z REPLACE 23PJ74a;
```

## 12.14 CALL

### Syntax

```
CALL procedure_name '('[argument_list]')'
```

### Description

You can use the CALL statement to call a stored procedure. To use the CALL statement, you must have the EXECUTE privilege on the stored procedure that is called.

### Parameters

| Parameter | Description |
|---|---|
| procedure_name | The name of the stored procedure. The name can be schema-qualified. |
| argument_list | A comma-separated list of parameters that are required by the stored procedure. Note that each member in the list corresponds to a parameter that is required by the stored procedure. Each parameter can be an IN, OUT, or INOUT parameter. |

### Examples

The CALL statement has different types of syntax based on the parameters that are required by the stored procedure:

```
CALL update_balance();
CALL update_balance(1,2,3);
```

## 12.15 COMMENT

Defines or modifies the comment of an object.

### Syntax

```
COMMENT ON
{
  TABLE table_name |
  COLUMN table_name.column_name
} IS 'text'
```

### Description

You can use the COMMENT command to store comments about database objects. To modify a comment of an object, you need to issue a new COMMENT command for the object. Only one comment string can be stored for each object. To delete a comment, specify an empty string (two consecutive single quotation marks with no space) for the text parameter. A comment is automatically deleted when the object is deleted.

**Note:**

Currently, no security mechanism is provided for comments. Any user who connects to the database can view all comments of the objects in the database. Do not include important security information in comments.

### Parameters

| Parameter | Description |
|---|---|
| table_name | The name of the table to be commented. The table name can be schema-qualified. |
| table_name.column_name | The name of the column to be commented in the table. The table name can be schema-qualified. |
| text | The new comment. |

### Examples

Attach a comment to a table named emp:

```
COMMENT ON TABLE emp IS 'Current employee information';
```

Attach a comment to the empno column of the emp table:

```
COMMENT ON COLUMN emp.empno IS 'Employee identification number';
```

Delete the comments:

```
COMMENT ON TABLE emp IS '';
COMMENT ON COLUMN emp.empno IS '';
```

## 12.16 COMMIT

Commits the current transaction.

### Syntax

```
COMMIT [ WORK ]
```

### Description

You can use the Commit command to commit the current transaction. All changes that are made by the transaction are visible to others and retained even if an exception occurs.

**Note:**

You can use the ROLLBACK command to abort the transaction. Issuing the COMMIT command outside the transaction does not cause damage.

When you run the COMMIT command in a PL/pgSQL procedure, an error occurs if an Oracle-style SPL stored procedure exists on the runtime stack.

### Parameters

| Parameter | Description |
|---|---|
| WORK | Optional. This keyword has no effect. |

### Example

Commit the current transaction and permanently store the changes:

```
COMMIT;
```

## 12.17 CREATE DATABASE

Creates a database.

### Syntax

```
CREATE DATABASE name
```

### Description

You cannot use the CREATE DATABASE command in a transaction block.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the database to be created. |

### Example

Create a database:

```
CREATE DATABASE employees;
```

## 12.18 CREATE [PUBLIC] DATABASE LINK

Creates a database link.

### Syntax

```
CREATE [ PUBLIC ] DATABASE LINK name
  CONNECT TO { CURRENT_USER |
             username IDENTIFIED BY 'password'}
  USING { postgres_fdw 'fdw_connection_string' |
       [ oci ] 'oracle_connection_string' }
```

### Description

The CREATE DATABASE LINK command creates a database link. A database link is an object that allows a reference to a table or view in a remote database within a DELETE, INSERT, SELECT, or UPDATE command. To reference a database link, you can append @dblink to the name of the table or view that is referenced in an SQL command. dblink is the name of the database link.

Database links can be public or private. A public database link can be used by all users. A private database link can be used only by the owner of the database link. If you specify the PUBLIC option, a public database link is created. If you do not specify the PUBLIC option, a private database link is created.

When you use the CREATE DATABASE LINK command, the database link name and the specified connection attributes are stored in the system table named pg_catalog.edb_dblink. A database link is defined in an edb_dblink entry. The database that contains the edb_dblink entry is called the local database. The server and database whose connection attributes are defined in the edb_dblink entry is called the remote database.

If an SQL command contains a reference to a database link, the SQL command must be issued when it is connected to the local database. When the SQL command is executed, the remote database is authenticated and connected to access the table or view to which the @dblink reference is appended.

**Note:**

- A database link cannot be used to access a remote database within a secondary database server. Secondary database servers are used for high availability, load balancing, and replication.
- For more information about high availability, load balancing, and replication for PostgreSQL database servers, see the PostgreSQL documentation.

**Parameters**

| Parameter | Description |
|---|---|
| PUBLIC | Specifies that the created database link is public. A public database link can be used by all users. If you do not specify this parameter, the database link is private and can be used only by the owner of the database link. |
| name | The name of the database link. |
| username | The username that is used for connecting to the remote database. |
| CURRENT_USER | Specifies that PolarDB uses the user mapping associated with the role that is using the link when establishing a connection to the remote server. |
| password | The password for the username. |
| postgres_fdw | Specifies the postgres_fdw foreign data wrapper as the connection to a remote PolarDB database. If postgres_fdw is not installed on the database, use the CREATE EXTENSION command to install postgres_fdw. For more information, see the CREATE EXTENSION command in the PostgreSQL documentation. |
| fdw_connection_string | The connection information for the postgres_fdw foreign data wrapper. |

| Parameter | Description |
|---|---|
| oci | A connection to a remote Oracle database. This is the default behavior of the PolarDB database. |
| oracle_connection_string | The connection information for an oci connection. |

**Description**

To create a non-public database link, you must have the CREATE DATABASE LINK privilege. To create a public database link, you must have the CREATE PUBLIC DATABASE LINK privilege .

- **Prepare an Oracle instant client for oci-dblink**

  To use oci-dblink, you must download and install an Oracle instant client on the host running the PolarDB database in which the database link is to be created.

  You can download an instant client from the following site: http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html

- **Oracle instant client for Linux**

  **Note:**
  The following instructions apply to Linux hosts running PolarDB databases compatible with Oracle.

  Make sure that the libaio library (the Linux-native asynchronous I/O facility) is installed on the Linux host running the PolarDB database compatible with Oracle.

  You can run the following command to install the libaio library:

```
yum install libaio
```

  If the Oracle instant client that you have downloaded does not include the file named libclntsh.so (without a version number suffix), you must create a symbolic link named libclntsh.so. This symbolic link must point to the downloaded version of the library file. Navigate to the instant client directory and run the following command:

```
ln -s libclntsh.so.version libclntsh.so
```

The version parameter indicates the version number of the libclntsh.so library. Example:

```
ln -s libclntsh.so.12.1 libclntsh.so
```

When you run an SQL command that references a database link to a remote Oracle database, the PolarDB database compatible with Oracle must know where the Oracle instant client library resides on the PolarDB host.

The LD_LIBRARY_PATH environment variable must include the path to the Oracle client installation directory that contains the libclntsh.so file. For example, the installation directory that contains libntsh. so is /tmp/instantclient.

```
export LD_LIBRARY_PATH=/tmp/instantclient:$LD_LIBRARY_PATH
```

**Note:**
The LD_LIBRARY_PATH environment variable setting must be effective when you call the pg_ctl utility to start or restart the PolarDB database compatible with Oracle.

If you are running the current session as the user account (such as enterprisedb) that invokes pg_ctl to start or restart PolarDB database compatible with Oracle, you must set LD_LIBRARY_PATH before calling pg_ctl.

You can set LD_LIBRARY_PATH in the ~enterprisedb/.bash_profile file. The ~enterprisedb/.bash_profile file refers to the .bash_profile file under the home directory of the enterprisedb user account. This ensures that LD_LIBRARY_PATH is set when you log on to the database as enterprisedb.

However, if you use a Linux service script with the systemctl or service command to start or restart the PolarDB database, you must set LD_LIBRARY_PATH in the service script. This ensures that the variable setting is effective when the script calls the pg_ctl utility.

The script file that needs to include the LD_LIBRARY_PATH setting depends on the version of the PolarDB database compatible with Oracle, the Linux system on which it is installed, and whether it is installed by using the graphical installer or an RPM package.

- **Oracle instant client for Windows**

**Note:**

The following instructions apply to Windows hosts running PolarDB databases compatible with Oracle.

When you run an SQL command that references a database link to a remote Oracle database, the PolarDB database compatible with Oracle must know where the Oracle instant client library resides on the PolarDB host.

Set the Windows PATH system environment variable to include the Oracle client installation directory that contains the oci.dll file.

You can also set the value of the oracle_home parameter in the postgresql.conf file. The value specified in the oracle_home parameter overwrites the Windows PATH environment variable.

To set the oracle_home parameter in the postgresql.conf file, edit the file and add the following line:

```
oracle_home = 'lib_directory '
```

Replace lib_directory with the name of the Windows directory that contains oci.dll. Example:

```
oracle_home = 'C:/tmp/instantclient_10_2'
```

After setting the PATH environment variable or the oracle_home parameter, you must restart the server for the changes to take effect. Restart the server from the Windows Services console.

**Note:**

If tnsnames.ora is configured in failover mode and a client:server failure occurs, a connection between the client and a secondary server is established. When the primary server resumes, the client retains the connection to a secondary server until a new session is established. The new client connections to the primary server are automatically established. If the primary and secondary servers are out of synchronization, the client that connects to the secondary server and the client that connects to the primary server may have different database views.

## Examples

### Create an oci-dblink database link

The following example shows how to use the CREATE DATABASE LINK command to create a database link named chicago. This database link connects a PolarDB cluster compatible with Oracle to an Oracle server through an oci-dblink connection. The connection information indicates that Apsara PolarDB logs on to Oracle as the admin user and the password is mypassword. The oci option specifies that this is an oci-dblink connection to the PolarDB database compatible with Oracle. The connection string '//127.0.0.1/acctg' specifies the server address and database name.

```
CREATE DATABASE LINK chicago
  CONNECT TO admin IDENTIFIED BY 'mypassword'
  USING oci '//127.0.0.1/acctg';
```

**Note:**

You can specify a hostname in the connection string in place of an IP address.

**Create a postgres_fdw database link**

The following example shows how to use the CREATE DATABASE LINK command to create a database link named bedford. This database link connects a PolarDB cluster compatible with Oracle to another PolarDB cluster compatible with Oracle by using a postgres_fdw foreign data wrapper connection. The connection information indicates that the PolarDB database compatible with Oracle logs on as the user admin with the password mypassword. The postgres_fdw option specifies that this is a postgres_fdw connection to the PolarDB database compatible with Oracle. The connection string 'host=127.0.0.1 port=5444 dbname=marketing' specifies the server address and database name.

```
CREATE DATABASE LINK bedford
  CONNECT TO admin IDENTIFIED BY 'mypassword'
  USING postgres_fdw 'host=127.0.0.1 port=5444 dbname=marketing';
```

**Note:**

You can specify a hostname in the connection string in place of an IP address.

**Use a database link**

The following examples show how to use a database link to connect to a PolarDB database compatible with Oracle. The examples assume that a copy of the emp table of the PolarDB sample application is created in an Oracle database. The examples also assume that a PolarDB cluster compatible with Oracle with the sample application is receiving connections at port 5443.

Create a public database link named oralink to an Oracle database named xe. The database address is 127.0.0.1 and port 1521 is used. Use the username (edb) and password (password) to connect to the Oracle database.

```
CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb
IDENTIFIED BY 'password' USING '//127.0.0.1:1521/xe';
```

In the Oracle database that uses the database link oralink, issue a SELECT command on the emp table.

```
SELECT * FROM emp@oralink;

 empno | ename  |    job    | mgr  |      hiredate      | sal  | comm | deptno
-------+--------+-----------+------+--------------------+------+------+--------
  7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800 |      |     20
  7499 | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600 |  300 |     30
  7521 | WARD   | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250 |  500 |     30
  7566 | JONES  | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975 |      |     20
  7654 | MARTIN | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250 | 1400 |     30
  7698 | BLAKE  | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850 |      |     30
  7782 | CLARK  | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450 |      |     10
  7788 | SCOTT  | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000 |      |     20
  7839 | KING   | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000 |      |     10
  7844 | TURNER | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500 |    0 |     30
  7876 | ADAMS  | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100 |      |     20
  7900 | JAMES  | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950 |      |     30
  7902 | FORD   | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000 |      |     20
  7934 | MILLER | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300 |      |     10
(14 rows)
```

Create a private database link named fdwlink to connect to the PolarDB database compatible with Oracle named edb. The database runs on host 192.168.2.22 and port 5444 . Use the username (enterprisedb) and password (password) to connect to the PolarDB database compatible with Oracle.

```
CREATE DATABASE LINK fdwlink CONNECT TO enterprisedb IDENTIFIED BY 'password'
USING postgres_fdw 'host=192.168.2.22 port=5444 dbname=edb';
```

Display attributes of the oralink and fdwlink database links from the local edb_dblink system table.

```
SELECT lnkname, lnkuser, lnkconnstr FROM pg_catalog.edb_dblink;

 lnkname |   lnkuser    |               lnkconnstr
---------+--------------+----------------------------------------
 oralink | edb          | //127.0.0.1:1521/xe
 fdwlink | enterprisedb |
(2 rows)
```

Join the emp table from the Oracle database with the dept table from the PolarDB database.

```
SELECT d.deptno, d.dname, e.empno, e.ename, e.job, e.sal, e.comm FROM emp@oralink
e, dept@fdwlink d WHERE e.deptno = d.deptno ORDER BY 1, 3;

 deptno |   dname    | empno | ename  |    job    | sal  | comm
--------+------------+-------+--------+-----------+------+------
     10 | ACCOUNTING |  7782 | CLARK  | MANAGER   | 2450 |
     10 | ACCOUNTING |  7839 | KING   | PRESIDENT | 5000 |
     10 | ACCOUNTING |  7934 | MILLER | CLERK     | 1300 |
     20 | RESEARCH   |  7369 | SMITH  | CLERK     |  800 |
     20 | RESEARCH   |  7566 | JONES  | MANAGER   | 2975 |
     20 | RESEARCH   |  7788 | SCOTT  | ANALYST   | 3000 |
     20 | RESEARCH   |  7876 | ADAMS  | CLERK     | 1100 |
     20 | RESEARCH   |  7902 | FORD   | ANALYST   | 3000 |
     30 | SALES      |  7499 | ALLEN  | SALESMAN  | 1600 |  300
     30 | SALES      |  7521 | WARD   | SALESMAN  | 1250 |  500
     30 | SALES      |  7654 | MARTIN | SALESMAN  | 1250 | 1400
     30 | SALES      |  7698 | BLAKE  | MANAGER   | 2850 |
     30 | SALES      |  7844 | TURNER | SALESMAN  | 1500 |    0
     30 | SALES      |  7900 | JAMES  | CLERK     |  950 |
(14 rows)
```

**Pushdown for an oci database link**

When you use the oci-dblink to run SQL statements on a remote Oracle database, the statements may be pushed down to a foreign server for processing.

Pushdown is the occurrence of processing on the foreign server rather than the local client where the SQL statement was issued. The foreign server is also known as the remote server. Pushdown can improve performance because the data is processed on the remote server before being returned to the local client.

Pushdown applies to statements with the standard SQL join operations, such as inner join, left outer join, right outer join, and full outer join. Pushdown still occurs even when a sort is specified on the resulting data set.

To perform pushdown, specific basic conditions must be met. The tables involved in the join operation must belong to the same foreign server and use the identical connection information to the foreign server. In other words, the connection information must be consistent with the definition of the database link defined in the CREATE DATABASE LINK command.

To determine whether an SQL statement can be pushed down, run the EXPLAIN command to display the execution plan.

For more information about the EXPLAIN command, see the PostgreSQL documentation.

The following examples use the database link created as follows:

```
CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password' USING
 '//192.168.2.23:1521/xe';
```

The following example shows the execution plan of an inner join:

```
EXPLAIN (verbose,costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM dept@
oralink d, emp@oralink e WHERE d.deptno = e.deptno ORDER BY 1, 3;

                                   QUERY PLAN
------------------------------------------------------------------------------------
 Foreign Scan
   Output: d.deptno, d.dname, e.empno, e.ename
   Relations: (_dblink_dept_1 d) INNER JOIN (_dblink_emp_2 e)
   Remote Query: SELECT r1.deptno, r1.dname, r2.empno, r2.ename FROM (dept r1 INNER
 JOIN emp r2 ON ((r1.deptno = r2.deptno))) ORDER BY r1.deptno ASC NULLS LAST, r2.
empno ASC NULLS LAST
(4 rows)
```

Note that the INNER JOIN operation occurs in the Foreign Scan section. The output of this join is as follows:

```
 deptno |   dname    | empno | ename
--------+------------+-------+--------
     10 | ACCOUNTING |  7782 | CLARK
     10 | ACCOUNTING |  7839 | KING
     10 | ACCOUNTING |  7934 | MILLER
     20 | RESEARCH   |  7369 | SMITH
     20 | RESEARCH   |  7566 | JONES
     20 | RESEARCH   |  7788 | SCOTT
     20 | RESEARCH   |  7876 | ADAMS
     20 | RESEARCH   |  7902 | FORD
     30 | SALES      |  7499 | ALLEN
     30 | SALES      |  7521 | WARD
     30 | SALES      |  7654 | MARTIN
     30 | SALES      |  7698 | BLAKE
     30 | SALES      |  7844 | TURNER
     30 | SALES      |  7900 | JAMES
(14 rows)
```

The following example shows the execution plan of a left outer join:

```
EXPLAIN (verbose,costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM dept@
oralink d LEFT OUTER JOIN emp@oralink e ON d.deptno = e.deptno ORDER BY 1, 3;

                                   QUERY PLAN
------------------------------------------------------------------------------------
 Foreign Scan
   Output: d.deptno, d.dname, e.empno, e.ename
   Relations: (_dblink_dept_1 d) LEFT JOIN (_dblink_emp_2 e)
   Remote Query: SELECT r1.deptno, r1.dname, r2.empno, r2.ename FROM (dept r1 LEFT
 JOIN emp r2 ON ((r1.deptno = r2.deptno))) ORDER BY r1.deptno ASC NULLS LAST, r2.
empno ASC NULLS LAST
```

```
(4 rows)
```

The output of this join is as follows:

```
 deptno |   dname    | empno | ename
--------+------------+-------+--------
     10 | ACCOUNTING |  7782 | CLARK
     10 | ACCOUNTING |  7839 | KING
     10 | ACCOUNTING |  7934 | MILLER
     20 | RESEARCH   |  7369 | SMITH
     20 | RESEARCH   |  7566 | JONES
     20 | RESEARCH   |  7788 | SCOTT
     20 | RESEARCH   |  7876 | ADAMS
     20 | RESEARCH   |  7902 | FORD
     30 | SALES      |  7499 | ALLEN
     30 | SALES      |  7521 | WARD
     30 | SALES      |  7654 | MARTIN
     30 | SALES      |  7698 | BLAKE
     30 | SALES      |  7844 | TURNER
     30 | SALES      |  7900 | JAMES
     40 | OPERATIONS |       |
(15 rows)
```

In the following example, the entire processing is not pushed down because the emp joined table resides locally instead of on the same foreign server.

```
EXPLAIN (verbose,costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM dept@
oralink d LEFT OUTER JOIN emp e ON d.deptno = e.deptno ORDER BY 1, 3;

                          QUERY PLAN
----------------------------------------------------------------------
 Sort
   Output: d.deptno, d.dname, e.empno, e.ename
   Sort Key: d.deptno, e.empno
   ->  Hash Left Join
         Output: d.deptno, d.dname, e.empno, e.ename
         Hash Cond: (d.deptno = e.deptno)
         ->  Foreign Scan on _dblink_dept_1 d
               Output: d.deptno, d.dname, d.loc
               Remote Query: SELECT deptno, dname, NULL FROM dept
         ->  Hash
               Output: e.empno, e.ename, e.deptno
               ->  Seq Scan on public.emp e
                     Output: e.empno, e.ename, e.deptno
(13 rows)
```

The output of this join is the same as that of the previous left outer join example.

**Create a foreign table from a database link**

**Note:**

The stored procedure described in this section is incompatible with Oracle databases.

After creating a database link, you can create a foreign table based on the database link. The foreign table can be used to access the remote table by referencing the foreign table with its name instead of using the database link syntax. Using the database link requires appending @dblink to the table or view name referenced in the SQL command. dblink is the name of the database link.

This technique can be used for either an oci-dblink connection for remote Oracle access or a postgres_fdw connection for remote Postgres access.

The following example shows how to create a foreign table to access a remote Oracle table .

First, create a database link as previously described. Run the following command to create a database link named oralink for connecting to the Oracle database:

```
CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password' USING
 '//127.0.0.1:1521/xe';
```

The following query shows the database link:

```
SELECT lnkname, lnkuser, lnkconnstr FROM pg_catalog.edb_dblink;

 lnkname | lnkuser |    lnkconnstr
---------+---------+---------------------
 oralink | edb     | //127.0.0.1:1521/xe
(1 row)
```

When you create the database link, a foreign server is created for the PolarDB database. The following query displays the foreign server:

```
SELECT srvname, srvowner, srvfdw, srvtype, srvoptions FROM pg_foreign_server;

 srvname | srvowner | srvfdw | srvtype |         srvoptions
---------+----------+--------+---------+------------------------------
 oralink |       10 |  14005 |         | {connstr=//127.0.0.1:1521/xe}
(1 row)
```

For more information about foreign servers, see the CREATE SERVER command in the PostgreSQL documentation.

Run the following commands to create the foreign table:

```
CREATE FOREIGN TABLE emp_ora (
    empno          NUMERIC(4),
    ename          VARCHAR(10),
    job            VARCHAR(9),
    mgr            NUMERIC(4),
    hiredate       TIMESTAMP WITHOUT TIME ZONE,
    sal            NUMERIC(7,2),
    comm           NUMERIC(7,2),
    deptno         NUMERIC(2)
)
  SERVER oralink
  OPTIONS (table_name 'emp', schema_name 'edb'
);
```

Note the following in the CREATE FOREIGN TABLE command:

- The name specified in the SERVER clause at the end of the CREATE FOREIGN TABLE command is the name of the foreign server. In this example, the name is oralink, as shown in the srvname column in the query for pg_frontend_server.
- The table name and schema name are specified in the OPTIONS clause by the table and schema options.
- The column names specified in the CREATE FOREIGN TABLE command must be the same as the column names in the remote table.
- CONSTRAINT clauses may not be accepted or enforced on the foreign table because they are assumed to have been defined on the remote table.

For more information about the CREATE FOREIGN TABLE command, see the PostgreSQL documentation.

The following is a query on the foreign table:

```
SELECT * FROM emp_ora;

 empno | ename  |    job    | mgr  |      hiredate      |   sal   |  comm   | deptno
-------+--------+-----------+------+--------------------+---------+---------+--------
  7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800.00 |         |     20
  7499 | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600.00 |  300.00 |     30
  7521 | WARD   | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250.00 |  500.00 |     30
  7566 | JONES  | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975.00 |         |     20
  7654 | MARTIN | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 |     30
  7698 | BLAKE  | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850.00 |         |     30
  7782 | CLARK  | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450.00 |         |     10
  7788 | SCOTT  | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000.00 |         |     20
  7839 | KING   | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000.00 |         |     10
  7844 | TURNER | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500.00 |    0.00 |     30
  7876 | ADAMS  | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100.00 |         |     20
  7900 | JAMES  | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950.00 |         |     30
  7902 | FORD   | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000.00 |         |     20
  7934 | MILLER | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300.00 |         |     10
(14 rows)
```

In contrast, the following is a query on the same remote table, but the database link rather than the foreign table is used.

```
SELECT * FROM emp@oralink;

 empno | ename  |    job    | mgr  |      hiredate      | sal  | comm | deptno
-------+--------+-----------+------+--------------------+------+------+--------
  7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800 |      |     20
  7499 | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600 |  300 |     30
  7521 | WARD   | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250 |  500 |     30
  7566 | JONES  | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975 |      |     20
  7654 | MARTIN | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250 | 1400 |     30
  7698 | BLAKE  | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850 |      |     30
  7782 | CLARK  | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450 |      |     10
  7788 | SCOTT  | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000 |      |     20
  7839 | KING   | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000 |      |     10
  7844 | TURNER | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500 |    0 |     30
  7876 | ADAMS  | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100 |      |     20
  7900 | JAMES  | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950 |      |     30
  7902 | FORD   | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000 |      |     20
  7934 | MILLER | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300 |      |     10
(14 rows)
```

**Note:**

For backward compatibility reasons, USING libpq rather than USING postgres_fdw can be written to the database. However, the libpq connector lacks many important optimizations that are provided by the postgres_fdw connector. We recommend that you use the postgres_fdw connector whenever possible. The libpq option is deprecated and may be deleted in the future version of PolarDB database compatible with Oracle.

## 12.19 CREATE FUNCTION

Creates a function.

### Syntax

```
CREATE [ OR REPLACE ] FUNCTION name [ (parameters) ]
  RETURN data_type
   [
       IMMUTABLE
     | STABLE
     | VOLATILE
     | DETERMINISTIC
     | [ NOT ] LEAKPROOF
     | CALLED ON NULL INPUT
     | RETURNS NULL ON NULL INPUT
     | STRICT
     | [ EXTERNAL ] SECURITY INVOKER
     | [ EXTERNAL ] SECURITY DEFINER
     | AUTHID DEFINER
     | AUTHID CURRENT_USER
     | PARALLEL { UNSAFE | RESTRICTED | SAFE }
     | COST execution_cost
     | ROWS result_rows
     | SET configuration_parameter
       { TO value | = value | FROM CURRENT }
   ...]
{ IS | AS }
    [ PRAGMA AUTONOMOUS_TRANSACTION; ]
    [ declarations ]
  BEGIN
    statements
```

```
END [ name ];
```

### Description

CREATE FUNCTION creates a function. CREATE OR REPLACE FUNCTION either creates a new function or replaces an existing definition.

If you specify a schema name, the function is created in the specified schema. Otherwise, the function is created in the current schema. The name of the new function cannot be the same as an existing function that has the same input argument types in the same schema . However, functions with different input argument types can share a name. This is called overloading. Overloading of functions is a feature of PolarDB databases compatible with Oracle. Overloading of stored standalone functions is incompatible with Oracle databases.

To update the definition of an existing function, you can use the CREATE OR REPLACE FUNCTION statement. You cannot use the statement to change the name or argument types of a function. If you have tried, a new distinct function is created. In addition, you cannot use the CREATE OR REPLACE FUNCTION statement to change the return type of an existing function. To change the return type of an existing function, you must delete the function and create the function again. When using the OUT parameters, you cannot change the types of OUT parameters unless you delete the function.

The user that creates the function becomes the owner of the function.

PolarDB databases compatible with Oracle support function overloading. The same name can be used for several different functions if they have distinct input (IN, IN OUT) argument data types.

### Parameters

| Parameter | Description |
|---|---|
| name | The identifier of the function. |
| parameters | A list of parameter values. |
| data_type | The data type of the value returned by the RETURN statement of the function. |
| declarations | Variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations . |

| Parameter | Description |
|---|---|
| statements | The SPL program statements. The BEGIN - END block can contain an EXCEPTION section. |
| IMMUTABLE \| STABLE \| VOLATILE | These attributes are used to inform the query optimizer about the behavior of the function. You can specify only one of them. VOLATILE is the default behavior.<br>• IMMUTABLE indicates that the function does not modify the database and always returns the same result if the same argument value is specified. The function does not perform database lookups or use information that is excluded from the argument list. If this clause is included, a call to the function with all-constant arguments can be immediately replaced with the function value.<br>• STABLE indicates that the function does not modify the database and that the function returns the same result for the same argument value within a single table scan. In this case, the result can change across SQL statements. This attribute is suitable for functions that depend on database lookups and parameter variables such as the current time zone.<br>• VOLATILE indicates that the function value changes within a single table scan. In this case, no optimizations can be made. Note that functions with negative effects must be classified as a volatile function, even if the results are predictable. This prevents calls from being removed due to optimization. |

| Parameter | Description |
|---|---|
| DETERMINISTIC | DETERMINISTIC is a synonym for IMMUTABLE . A DETERMINISTIC function does not modify the database and always returns the same result if the same argument value is specified. The function does not perform database lookups or use information that is excluded from the argument list. If this clause is included, a call to the function with all-constant arguments can be immediately replaced with the function value. |
| [ NOT ] LEAKPROOF | A LEAKPROOF function has no negative effects and reveals no information about the values used to call the function. |
| CALLED ON NULL INPUT \| RETURNS NULL ON NULL INPUT \| STRICT | • CALLED ON NULL INPUT is the default value. It indicates that the stored procedure is called when some arguments are NULL. If necessary, the author is responsible for checking NULL values and making proper responses.<br>• RETURNS NULL ON NULL INPUT or STRICT indicates that the stored procedure returns NULL whenever some arguments are NULL. If these clauses are specified , the stored procedure is not executed when NULL arguments exist. A NULL result is returned automatically. |
| [ EXTERNAL ] SECURITY DEFINER | SECURITY DEFINER specifies that the function executes with the privileges of the user that created it. This is the default value . The EXTERNAL keyword is allowed for SQL conformance but it is optional. |
| [ EXTERNAL ] SECURITY INVOKER | The SECURITY INVOKER clause indicates that the function executes with the privileges of the user that calls it. The EXTERNAL keyword is allowed for SQL conformance but it is optional. |

| Parameter | Description |
|---|---|
| AUTHID DEFINER \| AUTHID CURRENT_USER | • The AUTHID DEFINER clause is a synonym for [EXTERNAL] SECURITY DEFINER. If the AUTHID clause is omitted or AUTHID DEFINER is specified, the rights of the function owner are used to determine access privileges to database objects.<br>• The AUTHID CURRENT_USER clause is a synonym for [EXTERNAL] SECURITY INVOKER. If AUTHID CURRENT_USER is specified, the rights of the current user executing the function are used to determine access privileges. |
| PARALLEL { UNSAFE \| RESTRICTED \| SAFE } | The PARALLEL clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.<br>• If this parameter is set to UNSAFE, the function cannot be executed in the parallel mode. If such a function exists in an SQL statement, a serial execution plan is enforced. If the PARALLEL clause is omitted, this is the default setting.<br>• If this parameter is set to RESTRICTED, the function can be executed in the parallel mode, but the execution is restricted to the parallel group leader. If the qualificat ion for a particular relation has content that is parallel restricted, the relation is not selected for parallel execution.<br>• If this parameter is set to SAFE, the function can be executed in the parallel mode without restrictions. |
| COST execution_cost | execution_cost is a positive value that indicates the estimated execution cost of the function. The unit is cpu_operator_cost . If the function returns a set, this is the cost of each returned row. Larger values cause the planner to try to avoid evaluating the function more often than necessary. |

| Parameter | Description |
|---|---|
| ROWS result_rows | result_rows is a positive value that indicates the estimated number of rows that the planner expects the function to return. This value can be used only when the function is declared to return a set. The default value is 1,000 rows. |
| SET configuration_parameter { TO value \| = value \| FROM CURRENT } | The SET clause causes the specified configuration parameter to be set to the specified value when the function is entered, and then restored to its prior value when the function exits. SET FROM CURRENT saves the current value of the parameter as the value to be applied when the function is entered.<br>If a SET clause is attached to a function, the effects of a SET LOCAL command executed inside the function for the same variable are restricted to the function. The configurat ion parameter is restored to its prior value when the function exits. An ordinary SET command without LOCAL overrides the SET clause. This is similar to a previous SET LOCAL command. The effects of such a command persist after the function exits, unless the current transaction is rolled back. |
| PRAGMA AUTONOMOUS_TRANSACTION | PRAGMA AUTONOMOUS_TRANSACTION is the directive that sets the function as an autonomous transaction. |

**Note:**

The STRICT, LEAKPROOF, PARALLEL, COST, ROWS, and SET keywords provide extended functionality for PolarDB databases compatible with Oracle. However, these keywords are not supported by Oracle databases.

### Examples

The emp_comp function accepts two numbers as inputs and returns a computed value. The SELECT command is used to describe how to use the function.

```
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal           NUMBER,
    p_comm          NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;

SELECT ename "Name", sal "Salary", comm "Commission", emp_comp(sal, comm)
    "Total Compensation"  FROM emp;

  Name  | Salary  | Commission | Total Compensation
--------+---------+------------+--------------------
 SMITH  |  800.00 |            |           19200.00
 ALLEN  | 1600.00 |     300.00 |           45600.00
 WARD   | 1250.00 |     500.00 |           42000.00
 JONES  | 2975.00 |            |           71400.00
 MARTIN | 1250.00 |    1400.00 |           63600.00
 BLAKE  | 2850.00 |            |           68400.00
 CLARK  | 2450.00 |            |           58800.00
 SCOTT  | 3000.00 |            |           72000.00
 KING   | 5000.00 |            |          120000.00
 TURNER | 1500.00 |       0.00 |           36000.00
 ADAMS  | 1100.00 |            |           26400.00
 JAMES  |  950.00 |            |           22800.00
 FORD   | 3000.00 |            |           72000.00
 MILLER | 1300.00 |            |           31200.00
(14 rows)
```

The sal_range function returns the number of employees whose salary falls in the specified range. The following anonymous block calls the function multiple times and the default value of the arguments are used in the first two calls.

```
CREATE OR REPLACE FUNCTION sal_range (
    p_sal_min       NUMBER DEFAULT 0,
    p_sal_max       NUMBER DEFAULT 10000
) RETURN INTEGER
IS
    v_count         INTEGER;
BEGIN
    SELECT COUNT(*) INTO v_count FROM emp
        WHERE sal BETWEEN p_sal_min AND p_sal_max;
    RETURN v_count;
END;

BEGIN
    DBMS_OUTPUT.PUT_LINE('Number of employees with a salary: ' ||
        sal_range);
    DBMS_OUTPUT.PUT_LINE('Number of employees with a salary of at least '
        || '$2000.00: ' || sal_range(2000.00));
    DBMS_OUTPUT.PUT_LINE('Number of employees with a salary between '
        || '$2000.00 and $3000.00: ' || sal_range(2000.00, 3000.00));
```

```
END;

Number of employees with a salary: 14
Number of employees with a salary of at least $ 2000.00: 6
Number of employees with a salary between $ 2000.00 and $ 3000.00: 5
```

The following example shows how to use the AUTHID CURRENT_USER clause and STRICT keyword in a function declaration:

```
CREATE OR REPLACE FUNCTION dept_salaries(dept_id int) RETURN NUMBER
  STRICT
  AUTHID CURRENT_USER
BEGIN
  RETURN QUERY (SELECT sum(salary) FROM emp WHERE deptno = id);
END;
```

The STRICT keyword is included to instruct the server to return NULL if an input parameter passed is NULL. If a NULL value is passed, the function is not executed.

The dept_salaries function executes with the privileges of the role that is calling the function. If the current user does not have sufficient privileges to execute the SELECT statement to query the emp table (to display employee salaries), the function reports an error. To instruct the server to use the privileges associated with the role that defined the function, replace the AUTHID CURRENT_USER clause with the AUTHID DEFINER clause.

## 12.20 CREATE INDEX

Creates an index.

### Syntax

```
CREATE [ UNIQUE ] INDEX name ON table
  ( { column | ( expression ) } )
  [ TABLESPACE tablespace ]
```

### Description

CREATE INDEX constructs an index (name) on the specified table. Indexes are used to improve database performance. However, inappropriate use can result in unfavorable performance.

The key fields for the index are specified as column names or expressions written in parentheses. Multiple fields can be specified to create multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of a table row. This function can be used for quick data access based on some conversions of the basic data. For example, an index computed on UPPER(col) allows the WHERE UPPER(col) = 'JIM' clause to use an index.

PolarDB databases compatible with Oracle provide the B-tree index method. The B-tree index method is an implementation of Lehman-Yao high-concurrency B-trees.

Indexes are not used for IS NULL clauses by default.

All functions and operators used in an index definition must be immutable. Their results must depend only on their arguments and never on external influence such as the contents of another table or the current time. This restriction ensures that the behavior of the index is properly defined. To use a user-defined function in an index expression, you must mark the function as immutable when you create it.

If you create an index on a partition table, the CREATE INDEX command propagates indexes to the partitions of the table.

**Note:**

You can specify up to 32 fields in a multicolumn index.

**Parameters**

| Parameter | Description |
|---|---|
| UNIQUE | Causes the system to check for duplicate values in the table when the index is created if data already exists and each time data is added. If an attempt to insert or update data results in duplicate entries, an error is generated. |
| name | The name of the index to be created. The index name cannot contain a schema name. The index is always created in the same schema as its parent table. |
| table | The name of the table to be indexed. The name can be schema-qualified. |
| column | The name of a column in the table. |

| Parameter | Description |
|---|---|
| expression | An expression based on one or more columns of the table. The expression is enclosed in parentheses in most cases, as shown in the syntax. However, if the expression has the form of a function call, the parentheses can be omitted. |
| tablespace | The tablespace in which to create the index. If this parameter is not specified, default_tablespace is used. If default_tablespace is an empty string, the default tablespace of the database is used. |

### Example

Create a B-tree index on the ename column in the emp table:

```
CREATE INDEX name_idx ON emp (ename);
```

Create an index that is the same as the preceding one, but place it in the index_tblspc tablespace:

```
CREATE INDEX name_idx ON emp (ename) TABLESPACE index_tblspc;
```

## 12.21 CREATE MATERIALIZED VIEW

Creates a materialized view.

### Syntax

```
CREATE MATERIALIZED VIEW name     [build_clause][create_mv_refresh] AS subquery
```

where build_clause is:

```
BUILD {IMMEDIATE | DEFERRED}
```

where create_mv_refresh is:

```
REFRESH [COMPLETE] [ON DEMAND]
```

### Description

CREATE MATERIALIZED VIEW defines a view of a query that is not updated each time the view is referenced in a query. By default, the view is populated when the view is created. You can include the BUILD DEFERRED keywords to delay the population of the view.

A materialized view can be schema-qualified. If you specify a schema name when running the CREATE MATERIALIZED VIEW command, the view is created in the specified schema. The view name must be different from the names of all other views, tables, sequences, and indexes in the same schema.

**Note:**

Materialized views are read-only. The server does not allow an INSERT, UPDATE, or DELETE operation on a view.

Access to tables referenced in the view is determined by privileges of the view owner. The user of a view must have privileges to call all functions used by the view.

For more information about the Postgres REFRESH MATERIALIZED VIEW command, see the PostgreSQL documentation.

**Parameters**

| Parameter | Description |
|---|---|
| name | The name of the view to be created. The name can be schema-qualified. |
| subquery | A SELECT statement that specifies the contents of the view. For more informatio n about valid queries, see the SELECT command. |
| build_clause | Include a build_clause to specify when the view is populated. You can specify BUILD IMMEDIATE or BUILD DEFERRED.<br>• BUILD IMMEDIATE instructs the server to immediately populate the view. This is the default behavior.<br>• BUILD DEFERRED instructs the server to populate the view at a later time (during a REFRESH operation). |

| Parameter | Description |
|---|---|
| create_mv_refresh | Include the create_mv_refresh clause to specify when the content of a materialized view is updated. The clause contains the REFRESH keyword followed by COMPLETE and/or ON DEMAND, where:<br>• COMPLETE instructs the server to discard the current content and reload the materialized view by executing the defining query of the view when the materialized view is refreshed.<br>• ON DEMAND instructs the server to refresh the materialized view on demand by calling the DBMS_MVIEW package or by calling the Postgres REFRESH MATERIALIZED VIEW statement. This is the default behavior. |

### Examples

The following statement creates a materialized view named dept_30:

```
CREATE MATERIALIZED VIEW dept_30 BUILD IMMEDIATE AS SELECT * FROM emp WHERE
deptno = 30;
```

The view contains information retrieved from the emp table about all employees that work in department 30.

## 12.22 CREATE PACKAGE

Creates a package specification.

### Syntax

```
CREATE [ OR REPLACE ] PACKAGE name
[ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS }
  [ declaration; ] [, ...]
  [ { PROCEDURE proc_name
      [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
        [, ...]) ];
      [ PRAGMA RESTRICT_REFERENCES(name,
        { RNDS | RNPS | TRUST | WNDS | WNPS } [, ... ] ); ]
    |
      FUNCTION func_name
      [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
        [, ...]) ]
      RETURN rettype [ DETERMINISTIC ];
      [ PRAGMA RESTRICT_REFERENCES(name,
      { RNDS | RNPS | TRUST | WNDS | WNPS } [, ... ] ); ]
   }
  ] [, ...]
  END [ name ]
```

**Description**

CREATE PACKAGE creates a package specification. CREATE OR REPLACE TRIGGER either creates a new package specification or replaces an existing specification.

If you specify a schema name, the package is created in the specified schema. Otherwise, the package is created in the current schema. The name of the new package cannot be the same as an existing package in the same schema. If you want to update the definition of an existing package, you can use the CREATE OR REPLACE PACKAGE command.

The user that creates the stored procedure is the owner of the package.

**Parameters**

| Parameter | Description |
|---|---|
| name | The name of the package to be created. The name can be schema-qualified. |
| DEFINER \| CURRENT_USER | The privileges that determine whether access is allowed to database objects referenced in the package. DEFINER indicates the privileges of the package owner. CURRENT_USER indicates the privileges of the current user executing a program in the package. The default value is DEFINER. |
| declaration | A public variable, type, cursor, or REF CURSOR declaration. |
| proc_name | The name of a public stored procedure. |
| argname | The name of an argument. |
| IN \| IN OUT \| OUT | The argument mode. |
| argtype | The data types of the program arguments. |
| DEFAULT value | The default value of an input argument. |
| func_name | The name of a public function. |
| rettype | The return data type. |

| Parameter | Description |
|---|---|
| DETERMINISTIC | DETERMINISTIC is a synonym for IMMUTABLE . A DETERMINISTIC stored procedure cannot modify the database and always returns the same result if the same argument value is specified. The stored procedure does not perform database lookups or use information that is excluded from the argument list. If this clause is included, a call to the stored procedure with all-constant arguments is immediately replaced with the stored procedure value. |
| RNDS \| RNPS \| TRUST \| WNDS \| WNPS | The keywords are accepted for compatibility and are ignored. |

### Examples

The package specification (empinfo) contains three public components: a public variable, a public stored procedure, and a public function.

```
CREATE OR REPLACE PACKAGE empinfo
IS
    emp_name        VARCHAR2(10);
    PROCEDURE get_name (
        p_empno     NUMBER
    );
    FUNCTION display_counter
    RETURN INTEGER;
END;
```

## 12.23 CREATE PACKAGE BODY

Creates a package body.

### Syntax

```
CREATE [ OR REPLACE ] PACKAGE BODY name
{ IS | AS }
  [ declaration; ] [, ...]
  [ { PROCEDURE proc_name
      [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
        [, ...]) ]
      [ STRICT ]
      [ LEAKPROOF ]
      [ PARALLEL { UNSAFE | RESTRICTED | SAFE } ]
      [ COST execution_cost ]
      [ ROWS result_rows ]
      [ SET config_param { TO value | = value | FROM CURRENT } ]
    { IS | AS }
        program_body
      END [ proc_name ];
    |
      FUNCTION func_name
      [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
        [, ...]) ]
      RETURN rettype [ DETERMINISTIC ]
      [ STRICT ]
      [ LEAKPROOF ]
      [ PARALLEL { UNSAFE | RESTRICTED | SAFE } ]
      [ COST execution_cost ]
      [ ROWS result_rows ]
      [ SET config_param { TO value | = value | FROM CURRENT } ]
    { IS | AS }
        program_body
      END [ func_name ];
    }
  ] [, ...]
  [ BEGIN
      statement; [, ...] ]
  END [ name ]
```

**Description**

CREATE PACKAGE BODY creates a package body. CREATE OR REPLACEPACKAGE BODY creates a new package body or replaces an existing body.

If you specify a schema name, the package body is created in the specified schema. Otherwise, the package body is created in the current schema. The name of the new package body must match an existing package specification in the same schema. The name of the new package body cannot be the same as an existing package body in the same schema. If you want to update the definition of an existing package body, you can use the CREATE OR REPLACE PACKAGE BODY command.

**Parameters**

| Parameter | Description |
|---|---|
| name | The name of the package body to be created. The name can be schema-qualified . |
| declaration | A private variable, type, cursor, or REF CURSOR declaration. |
| proc_name | The name of a public stored procedure or private stored procedure. If proc_name with the same signature exists in the package specification, the stored procedure is public . Otherwise, the stored procedure is private. |
| argname | The name of an argument. |
| IN \| IN OUT \| OUT | The argument mode. |

| Parameter | Description |
|---|---|
| argtype | The data types of the program arguments. |
| DEFAULT value | The default value of an input argument. |
| STRICT | The STRICT keyword specifies that the function is not executed when a NULL parameter is used to call the function. On the contrary, the function returns NULL. |
| LEAKPROOF | The LEAKPROOF keyword specifies that the function does not reveal information about arguments, other than through a return value. |
| PARALLEL { UNSAFE \| RESTRICTED \| SAFE } | The PARALLEL clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.<br>• If this parameter is set to UNSAFE, the stored procedure or function cannot be executed in the parallel mode. If such a stored procedure or function exists in an SQL statement, a serial execution plan is enforced. If the PARALLEL clause is omitted, this is the default setting.<br>• If this parameter is set to RESTRICTED, the stored procedure or function can be executed in the parallel mode, but the execution is restricted to the parallel group leader. If the qualification for a particular relation has content that is parallel restricted, the relation is not selected for parallel execution.<br>• If this parameter is set to SAFE, the stored procedure or function can be executed in the parallel mode without restrictions. |
| execution_cost | execution_cost is a positive value that indicates the estimated execution cost of the function. The unit is cpu_operator_cost. If the function returns a set, this is the cost of each returned row. The default value is 0 .0025. |

| Parameter | Description |
|---|---|
| result_rows | result_rows is a positive value that indicates the estimated number of rows that the planner expects the function to return. The default value is 1000. |
| SET | You can use the SET clause to specify a parameter value for the duration of the function:<br>• config_param specifies the parameter name.<br>• value specifies the parameter value.<br>• FROM CURRENT ensures that the parameter value is restored when the function ends. |
| program_body | The pragma, declarations, and SPL statements that comprise the body of the function or stored procedure.<br>The pragma can be PRAGMA AUTONOMOUS _TRANSACTION to set the function or stored procedure as an autonomous transaction.<br>The declarations can include variable, type , REF CURSOR, and subprogram declaratio ns. If subprogram declarations are included , they must be declared after all other variable, type, and REF CURSOR declaratio ns. |
| func_name | The name of a public or private function. If func_name with the same signature exists in the package specification, the function is public. Otherwise, the function is private. |
| rettype | The return data type. |

| Parameter | Description |
|---|---|
| DETERMINISTIC | You can use DETERMINISTIC to specify that the function always returns the same result if the same argument value is specified. A DETERMINISTIC function does not modify the database.<br>**Note:**<br>• The DETERMINISTIC keyword is equivalent to the PostgreSQL IMMUTABLE option.<br>• If you have specified the DETERMINISTIC keyword for a public function in the package body, you must also specify this keyword for the function declaration in the package specification. For private functions, no function declarations are included in the package specification. |
| statement | An SPL program statement. If a package is referenced for the first time, the statements in the package initialization section are executed once for each session. |

**Note:**

The STRICT, LEAKPROOF, PARALLEL, COST, ROWS, and SET keywords provide extended functionality for PolarDB databases compatible with Oracle. However, these keywords are not supported by Oracle databases.

### Examples

The following is the package body for the empinfo package.

```
CREATE OR REPLACE PACKAGE BODY empinfo
IS
    v_counter       INTEGER;
    PROCEDURE get_name (
        p_empno     NUMBER
    )
    IS
    BEGIN
        SELECT ename INTO emp_name FROM emp WHERE empno = p_empno;
        v_counter := v_counter + 1;
    END;
    FUNCTION display_counter
    RETURN INTEGER
    IS
    BEGIN
        RETURN v_counter;
    END;
BEGIN
    v_counter := 0;
    DBMS_OUTPUT.PUT_LINE('Initialized counter');
END;
```

The following two anonymous blocks execute the stored procedure and function in the empinfo package and display the public variable.

```
BEGIN
    empinfo.get_name(7369);
    DBMS_OUTPUT.PUT_LINE('Employee Name    : ' || empinfo.emp_name);
    DBMS_OUTPUT.PUT_LINE('Number of queries: ' || empinfo.display_counter);
END;

Initialized counter
Employee name: SMITH
Number of queries: 1

BEGIN
    empinfo.get_name(7900);
    DBMS_OUTPUT.PUT_LINE('Employee Name    : ' || empinfo.emp_name);
    DBMS_OUTPUT.PUT_LINE('Number of queries: ' || empinfo.display_counter);
END;

Employee name: JAMES
Number of queries: 2
```

## 12.24 CREATE PROCEDURE

Creates a stored procedure.

### Syntax

```
CREATE [OR REPLACE] PROCEDURE name [ (parameters) ]
  [
       IMMUTABLE
      | STABLE
      | VOLATILE
      | DETERMINISTIC
      | [ NOT ] LEAKPROOF
      | CALLED ON NULL INPUT
      | RETURNS NULL ON NULL INPUT
      | STRICT
      | [ EXTERNAL ] SECURITY INVOKER
      | [ EXTERNAL ] SECURITY DEFINER
      | AUTHID DEFINER
      | AUTHID CURRENT_USER
      | PARALLEL { UNSAFE | RESTRICTED | SAFE }
      | COST execution_cost
      | ROWS result_rows
      | SET configuration_parameter
       { TO value | = value | FROM CURRENT }
  ...]
{ IS | AS }
    [ PRAGMA AUTONOMOUS_TRANSACTION; ]
    [ declarations ]
  BEGIN
    statements
  END [ name ];
```

**Description**

CREATE PROCEDURE creates a stored procedure. CREATE OR REPLACE PROCEDURE either creates a new stored procedure or replaces an existing definition.

If you specify a schema name, the stored procedure is created in the specified schema. Otherwise, the stored procedure is created in the current schema. The name of the new stored procedure cannot be the same as an existing stored procedure that has the same input argument types in the same schema. However, stored procedures of different input argument types can share a name. This is called overloading. Overloading of stored procedures is a feature of PolarDB database compatibles with Oracle. Overloading of standalone stored procedures is incompatible with Oracle databases.

To update the definition of an existing stored procedure, you can use the CREATE OR REPLACE PROCEDURE statement. You cannot use the statement to change the name or argument types of a stored procedure. If you have tried, a new distinct stored procedure is created. When using the OUT parameters, you cannot change the types of OUT parameters unless you delete the stored procedure.

**Parameters**

| Parameter | Description |
|---|---|
| name | The identifier of the stored procedure. |
| parameters | A list of parameter values. |
| declarations | Variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations . |
| statements | The SPL program statements. The BEGIN - END block can contain an EXCEPTION section. |

| Parameter | Description |
|---|---|
| IMMUTABLE \| STABLE \| VOLATILE | These attributes are used to inform the query optimizer about the behavior of the stored procedure. You can specify only one of them. VOLATILE is the default behavior.<br>• IMMUTABLE indicates that the stored procedure does not modify the database and always returns the same result if the same argument value is specified. The stored procedure does not perform database lookups or use information that is excluded from the argument list. If this clause is included, a call to the stored procedure with all-constant arguments can be immediately replaced with the stored procedure value.<br>• STABLE indicates that the stored procedure does not modify the database and that the stored procedure returns the same result for the same argument value within a single table scan. In this case, the result can change across SQL statements. This attribute is suitable for stored procedures that depend on database lookups and parameter variables such as the current time zone.<br>• VOLATILE indicates that the stored procedure value changes within a single table scan. In this case, no optimizations can be made. Note that functions with negative effects must be classified as a volatile function, even if the results are predictable. This prevents calls from being removed due to optimization. |

| Parameter | Description |
|---|---|
| DETERMINISTIC | DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC stored procedure does not modify the database and always returns the same result if the same argument value is specified. The stored procedure does not perform database lookups or use information that is excluded from the argument list. If this clause is included, a call to the stored procedure with all-constant arguments can be immediately replaced with the stored procedure value. |
| [ NOT ] LEAKPROOF | A LEAKPROOF stored procedure has no negative effects and reveals no information about the values used to call the stored procedure. |
| CALLED ON NULL INPUT \| RETURNS NULL ON NULL INPUT \| STRICT | • CALLED ON NULL INPUT is the default value. It indicates that the stored procedure is called when some arguments are NULL. If necessary, the author is responsible for checking NULL values and making proper responses.<br>• RETURNS NULL ON NULL INPUT or STRICT indicates that the stored procedure returns NULL whenever some arguments are NULL. If these clauses are specified, the stored procedure is not executed when NULL arguments exist. A NULL result is returned automatically. |
| [ EXTERNAL ] SECURITY DEFINER | SECURITY DEFINER specifies that the stored procedure executes with the privileges of the user that created it. This is the default value. The EXTERNAL keyword is allowed for SQL conformance but it is optional. |
| [ EXTERNAL ] SECURITY INVOKER | The SECURITY INVOKER clause indicates that the stored procedure executes with the privileges of the user that calls it. The EXTERNAL keyword is allowed for SQL conformance but it is optional. |

| Parameter | Description |
|---|---|
| AUTHID DEFINER \| AUTHID CURRENT_USER | • The AUTHID DEFINER clause is a synonym for [EXTERNAL] SECURITY DEFINER. If the AUTHID clause is omitted or AUTHID DEFINER is specified, the rights of the stored procedure owner are used to determine access privileges to database objects.<br>• The AUTHID CURRENT_USER clause is a synonym for [EXTERNAL] SECURITY INVOKER. If AUTHID CURRENT_USER is specified, the rights of the current user executing the stored procedure are used to determine access privileges. |
| PARALLEL { UNSAFE \| RESTRICTED \| SAFE } | The PARALLEL clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.<br>• If this parameter is set to UNSAFE, the stored procedure cannot be executed in the parallel mode. If such a stored procedure exists in an SQL statement, a serial execution plan is enforced. If the PARALLEL clause is omitted, this is the default setting.<br>• If this parameter is set to RESTRICTED, the stored procedure can be executed in the parallel mode, but the execution is restricted to the parallel group leader. If the qualification for a particular relation has content that is parallel restricted, the relation is not selected for parallel execution.<br>• If this parameter is set to SAFE, the stored procedure can be executed in the parallel mode without restrictions. |

| Parameter | Description |
|---|---|
| COST execution_cost | execution_cost is a positive value that indicates the estimated execution cost of the stored procedure. The unit is cpu_operator_cost. If the stored procedure returns a set, this is the cost of each returned row. Larger values cause the planner to try to avoid evaluating the stored procedure more often than necessary. |
| ROWS result_rows | result_rows is a positive value that indicates the estimated number of rows that the planner expects the stored procedure to return. This value can be used only when the stored procedure is declared to return a set. The default value is 1000 rows. |
| SET configuration_parameter { TO value \| = value \| FROM CURRENT } | The SET clause causes the specified configuration parameter to be set to the specified value when the stored procedure is entered, and then restored to its prior value when the stored procedure exits. SET FROM CURRENT saves the current value of the parameter as the value to be applied when the stored procedure is entered.<br>If a SET clause is attached to a stored procedure, the effects of a SET LOCAL command executed inside the stored procedure for the same variable are restricted to the stored procedure. The configuration parameter is restored to its prior value when the stored procedure exits. When the stored procedure exits, the configuration parameter is restored to its prior value. An ordinary SET command without LOCAL overrides the SET clause. This is similar to a previous SET LOCAL command. The effects of such a command persist after the stored procedure exits, unless the current transaction is rolled back. |

| Parameter | Description |
|---|---|
| PRAGMA AUTONOMOUS_TRANSACTION | PRAGMA AUTONOMOUS_TRANSACTION is the directive that sets the stored procedure as an autonomous transaction. |

**Note:**

- The STRICT, LEAKPROOF, PARALLEL, COST, ROWS, and SET keywords provide extended functionality for PolarDB databases compatible with Oracle. However, these keywords are not supported by Oracle databases.
- The IMMUTABLE, STABLE, STRICT, LEAKPROOF, COST, ROWS and PARALLEL { UNSAFE | RESTRICTED | SAFE } attributes are supported only by stored procedures of PolarDB database compatible with Oracle.
- Stored procedures are created as SECURITY DEFINERS by default. Stored procedures defined in plpgsql are created as SECURITY INVOKERS.

**Examples**

The following stored procedure lists the employees in the emp table:

```
CREATE OR REPLACE PROCEDURE list_emp
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur IS
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_cur;
END;

EXEC list_emp;

EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
```

```
7876    ADAMS
7900    JAMES
7902    FORD
7934    MILLER
```

The following stored procedure uses IN OUT and OUT arguments to return the number, name, and job of an employee. First, the search is based on the specified employee number. If no results are found, the specified name is used. An anonymous block calls the stored procedure.

```
CREATE OR REPLACE PROCEDURE emp_job (
    p_empno         IN OUT emp.empno%TYPE,
    p_ename         IN OUT emp.ename%TYPE,
    p_job           OUT    emp.job%TYPE
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
BEGIN
    SELECT ename, job INTO v_ename, v_job FROM emp WHERE empno = p_empno;
    p_ename := v_ename;
    p_job   := v_job;
    DBMS_OUTPUT.PUT_LINE('Found employee # ' || p_empno);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        BEGIN
            SELECT empno, job INTO v_empno, v_job FROM emp
                WHERE ename = p_ename;
            p_empno := v_empno;
            p_job   := v_job;
            DBMS_OUTPUT.PUT_LINE('Found employee ' || p_ename);
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE('Could not find an employee with ' ||
                    'number, ' || p_empno || ' nor name, '  || p_ename);
                p_empno := NULL;
                p_ename := NULL;
                p_job   := NULL;
        END;
END;

DECLARE
    v_empno      emp.empno%TYPE;
    v_ename      emp.ename%TYPE;
    v_job        emp.job%TYPE;
BEGIN
    v_empno := 0;
    v_ename := 'CLARK';
    emp_job(v_empno, v_ename, v_job);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
END;

Found employee CLARK
Employee No: 7782
Name       : CLARK
```

```
Job       : MANAGER
```

The following example shows how to use the AUTHID DEFINER and SET clauses in a procedure declaration. The update_salary stored procedure grants the privileges of the role that defined the stored procedure to the role that is calling the stored procedure:

```
CREATE OR REPLACE PROCEDURE update_salary(id INT, new_salary NUMBER)
  SET SEARCH_PATH = 'public' SET WORK_MEM = '1MB'
  AUTHID DEFINER IS
BEGIN
  UPDATE emp SET salary = new_salary WHERE emp_id = id;
END;
```

You can use the SET clause to set the search path of the stored procedure to public and set the working memory to 1 MB. Other stored procedures, functions, and objects are not affected by these settings.

In this example, the AUTHID DEFINER clause temporarily grants privileges to a role that may not be allowed to execute the statements within the stored procedure. To instruct the server to use the privileges associated with the role that calls the stored procedure, replace the AUTHID DEFINER clause with the AUTHID CURRENT_USER clause.

## 12.25 CREATE QUEUE

Creates a queue.

### Syntax

You can use the following syntax to define a new queue:

```
CREATE QUEUE name QUEUE TABLE queue_table_name [ ( { option_name option_value}
 [, ... ] ) ]
```

The following section describes valid values of the `option_name` and `option_value` parameters.

```
TYPE [normal_queue | exception_queue]
RETRIES [INTEGER]
RETRYDELAY [DOUBLE PRECISION]
RETENTION [DOUBLE PRECISION]
```

### Description

You can use the CREATE QUEUE command to create a queue in the current database if you are a superuser or a user who has the aq_administrator_role privilege.

If a queue name is schema-qualified, the queue is created in the specified schema. Otherwise, the queue is created in the current schema. A queue can only be created in the schema in which the queue table resides. A queue name must be unique in the schema.

**Note:**

- PolarDB databases compatible with Oracle provides additional syntax of the CREATE QUEUE SQL command. You can use this additional syntax with the DBMS_AQADM package.
- You can use the DROP QUEUE command to delete a queue.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the sequence to be created. The name can be schema-qualified. |
| queue_table_name | The name of the queue table that is associated with the queue. |
| option_name<br>option_value | The names and values of options that are associated with the new queue. If the call to the CREATE QUEUE TABLE includes duplicate option names, the server returns an error. The following section describes valid values of these two parameters:<br>• TYPE: specifies whether a queue is a normal queue or exception queue. Valid values: normal_queue and exception_queue. The exception queue only supports dequeuing operations.<br>• RETRIES: specifies the maximum number of attempts to delete a message from the queue. Data type: INTEGER.<br>• RETRYDELAY: specifies the number of seconds after a rollback that the server waits before retrying the message. Data type: DOUBLE PRECISION.<br>• RETENTION: specifies the number of seconds for which a message is stored in the queue table after dequeuing. Data type: DOUBLE PRECISION. |

### Example

Run the following command to create a queue named work_order that is associated with the queue table named work_order_table:

```
CREATE QUEUE work_order QUEUE TABLE work_order_table (RETRIES 5, RETRYDELAY 2);
```

The server allows five attempts to delete messages from the queue and requires an interval of 2 seconds between two retry attempts.

# 12.26 CREATE QUEUE TABLE

Creates a queue table.

### Syntax

```
CREATE QUEUE TABLE name OF type_name [ ( { option_name option_value } [, ... ] ) ]
```

The following table lists valid options of the option_name and option_value parameters.

| option_name | option_value |
|---|---|
| SORT_LIST | priority and enq_time |
| MULTIPLE_CONSUMERS | FALSE and TRUE |
| MESSAGE_GROUPING | NONE and TRANSACTIONAL |
| STORAGE_CLAUSE | TABLESPACE tablespace_name, PCTFREE integer, PCTUSED integer, INITRANS integer, MAXTRANS integer, and STORAGE storage_option<br>storage_option can be one or more of the following clauses:<br>MINEXTENTS integer, MAXEXTENTS integer, PCTINCREASE integer, INITIAL size_clause, NEXT, FREELISTS integer, OPTIMAL size_clause, and BUFFER_POOL {KEEP\|RECYCLE\|DEFAULT}.<br>**Note:**<br>Only the TABLESPACE clause is enforced. You can skip all other options, which are supported for compatibility. You can use the TABLESPACE clause to specify the name of the tablespace in which the table will be created. |

### Description

You can use the CREATE QUEUE TABLE command to create a queue table if you are a superuser or a user who has the aq_administrator_role privilege.

If the call to CREATE QUEUE TABLE includes a schema name, the queue table is created in the specified schema. If you do not specify a schema, the queue table is created in the current schema.

The name of a queue table must be unique in a schema.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the queue table to be created. The name can be schema-qualified. |
| type_name | The name of the current type that describes payloads of each entry in the queue table. For more information about how to define a type, see the CREATE TYPE topic. |
| option_name<br>option_value | The names and values of options that are associated with the new queue table. If the call to the CREATE QUEUE TABLE includes duplicate option names, the server returns an error. The following table lists valid options of these two parameters. |

**Table 12-4: Table of option names and values**

| Parameter | Description |
|---|---|
| SORT_LIST | Specifies the dequeuing order and the names of one or more columns that are used to sort the queue in ascending order.<br>Valid values include the following combinations of enq_time and priority:<br>• enq_time. priority<br>• priority. enq_time<br>• priority<br>• enq_time<br>• If you specify any other value, ERROR is returned. |
| MULTIPLE_CONSUMERS | Specifies whether a message can be consumed by multiple users or only one user. Data type: BOOLEAN. Valid values: TRUE and FALSE. |
| MESSAGE_GROUPING | Specifies the method in which a message is dequeued. none: indicates that each message is dequeued separately. transactional: indicates that multiple messages are added to the queue in a single transaction and dequeued as a group. |

| Parameter | Description |
|---|---|
| STORAGE_CLAUSE | Specifies the parameters of a table. Valid values: TABLESPACE tablespace_name, PCTFREE integer, PCTUSED integer, INITRANS integer, MAXTRANS integer, and STORAGE storage_option.<br>Storage_option can be one or more of the following values: MINEXTENTS integer, MAXEXTENTS integer, PCTINCREASE integer, INITIAL size_clause, NEXT, FREELISTS integer, OPTIMAL size_clause, and BUFFER_POOL {KEEP\|RECYCLE\|DEFAULT}.<br>**Note:**<br>Only the TABLESPACE clause is executed. You can skip all other options, which are supported for compatibility. You can use the TABLESPACE clause to specify the name of the tablespace in which the table will be created. |

### Example

Before creating a queue table, you must create a custom type. This type describes the columns and data types in the table. You can run the following command to create a type named work_order:

```
CREATE TYPE work_order AS (name VARCHAR2, project TEXT, completed BOOLEAN);
```

You can run the following command to use the work_order type to create a queue table named work_order_table:

```
CREATE QUEUE TABLE work_order_table OF work_order (sort_list (enq_time, priority));
```

## 12.27 CREATE ROLE

Defines a new database role.

### Syntax

```
CREATE ROLE name [IDENTIFIED BY password [REPLACE old_password]]
```

### Description

You can use the CREATE ROLE command to create a role for a PolarDB database cluster. A role is an entity that owns database objects and is authorized to manage the database . A role can be considered a user, group, or combination of both based on usage. A new role does not have the LOGIN privilege and cannot be used to start a session. You can use the ALTER ROLE command to grant the LOGIN privilege to the role. To use the CREATE ROLE command, you must be a database superuser or have the CREATEROLE privilege.

If you specify the IDENTIFIED BY clause when using the CREATE ROLE command, a schema that is owned by and has the same name as the new role is created.

**Note:**

Roles are defined at the database cluster level and are valid in all databases in a cluster.

**Parameters**

| Parameter | Description |
|---|---|
| name | The name of the new role. |
| IDENTIFIED BY password | Specifies the password of the role. A password is only used for roles who have the LOGIN privilege. However, you can also define a password for roles who do not have this privilege. If you do not want to use password verification, you can leave this parameter empty. |

**Notes**

You can use the ALTER ROLE command to modify the parameters of a role, and the DROP ROLE command to delete a role. You can use the ALTER ROLE command to modify the parameters that are specified by the CREATE ROLE command.

You can use the GRANT and REVOKE command to add and remove role members when roles are used as groups.

A role name or password can be up to 63 characters in length.

### Examples

Create a role named admins and a schema, and specify a password:

```
CREATE ROLE admins IDENTIFIED BY Rt498zb;
```

## 12.28 CREATE SCHEMA

Defines a new schema.

### Syntax

```
CREATE SCHEMA AUTHORIZATION username schema_element [ ... ]
```

### Description

You can use the variant of the CREATE SCHEMA command to create a schema that has one or more objects. The username parameter specifies the owner of the schema. A schema and objects are created in a single transaction. Therefore, all the created objects include the schema. Otherwise, none of the created objects include the schema. Note: If you are using an Oracle database, no new schema (username) is created. Therefore, the schema must already exist.

A schema is a namespace that contains named objects such as tables and views. Different schemas may have the same named objects. You can access named objects by using either of the following methods: 1. Qualify the name of an object by using the schema name as the prefix. 2. Specify a search path that includes the required schema. Unqualified objects are created in the current schema (the schema before the search path, which can be determined by the CURRENT_SCHEMA function). The search paths and CURRENT_SCHEMA function are incompatible with Oracle databases.

The CREATE SCHEMA command includes subcommands to create objects within the schema. Subcommands are processed in the same methods as separate commands that are issued after the schema is created. All the created objects are owned by the specified user.

**Note:**

To create a schema, you must have the CREATE privilege on the current database.

### Parameters

| Parameter | Description |
|---|---|
| username | The name of the user who owns the new schema. The schema name is the same as the username. Only superusers can create schemas that are owned by other users. Note: In PolarDB databases compatible with Oracle, the role and username must already exist, and the schema must not exist. In Oracle, a user that is equivalent to a schema must already exist. |
| schema_element | An SQL statement that defines the objects to be created in the schema. You can use the CREATE TABLE, CREATE VIEW, and GRANT clauses within the CREATE SCHEMA statement. After creating a schema, you can create other object types by using separate commands. |

### Example

```
CREATE SCHEMA AUTHORIZATION enterprisedb
    CREATE TABLE empjobs (ename VARCHAR2(10), job VARCHAR2(9))
    CREATE VIEW managers AS SELECT ename FROM empjobs WHERE job = 'MANAGER'
    GRANT SELECT ON managers TO PUBLIC;
```

## 12.29 CREATE SEQUENCE

Defines a new sequence generator.

### Syntax

```
CREATE SEQUENCE name [ INCREMENT BY increment ]
  [ { NOMINVALUE | MINVALUE minvalue } ]
  [ { NOMAXVALUE | MAXVALUE maxvalue } ]
  [ START WITH start ] [ CACHE cache | NOCACHE ] [ CYCLE ]
```

### Description

You can use the CREATE SEQUENCE command to create a sequence generator. A single-row table named name is generated and initialized. The generator is owned by the user who runs the command.

If you specify a schema, a sequence is created in the specified schema. Otherwise, a sequence is created in the current schema. The sequence name must be different from the name of any other sequence, table, index, or view in the same schema.

After a sequence is created, you can use the NEXTVAL and CURRVAL functions to manage the sequence.

**Parameters**

| Parameter | Description |
|---|---|
| name | The name of the sequence to be created. The name can be schema-qualified. |
| increment | The INCREMENT BY increment clause is optional and specifies the value to be added to the current sequence value. A positive value indicates an ascending sequence, and a negative value indicates a descending sequence. Default value: 1. |
| NOMINVALUE \| MINVALUE minvalue | The MINVALUE minvalue clause is optional and specifies the minimum value that a sequence can generate. If you do not specify this clause, the default value is used. Default value for ascending sequences: 1. Default value for descending sequences: -263 - 1. Note that you can use the NOMINVALUE keyword to specify the default value. |
| NOMAXVALUE \| MAXVALUE maxvalue | The MAXVALUE maxvalue clause is optional and specifies the maximum value that a sequence can generate. If you do not specify this clause, the default value is used. Default value for ascending sequences: 263 - 1. Default value for descending sequences: -1. Note that you can use the NOMAXVALUE keyword to specify the default value. |
| start | The START WITH start clause is optional and specifies the number from which a sequence starts. By default, ascending sequences start from the value that is specified for the minvalue parameter, and descending sequences start from the value that is specified for the maxvalue parameter. |
| cache | The CACHE cache clause is optional and specifies the number of sequence numbers to be allocated and stored in memory for fast access. The minimum value is 1, indicating that only one value can be generated at a time, such as NOCACHE. Default value: 1. |
| CYCLE | Allows a sequence to wrap around when the ascending sequence reaches the maximum value or descending sequence reaches the minimum value. If the limit is reached, the next number generated is the value that is specified by the minvalue or maxvalue parameter.<br>This parameter is not specified by default. If you do not specify this parameter, any call to the NEXTVAL function after the sequence has reached the maximum value returns an error. Note: You can use the NO CYCLE keyword to specify the default value. This keyword is not compatible with Oracle databases. |

### Notes

Sequences are based on big integer arithmetic. The sequence range cannot exceed the range of an eight-byte integer. Valid values: -9223372036854775808 to +9223372036854775807. On early platforms, compilers may not support eight-byte integers. In this case, sequences use regular integer arithmetic that ranges from -2147483648 to +2147483647.

If multiple sessions concurrently use a sequence object whose cache parameter is set to a value greater than 1, unexpected results may be retrieved. Each session allocates and caches consecutive sequence values during each access to the sequence object, and increases the final value of the sequence object. Then, the next cache-1 uses of the NEXTVAL function within the session return the preallocated values without touching the sequence object. Therefore, when the session ends, all values that have been allocated but not used within the session are lost and several gaps are generated in the sequence.

Although different sequence values can be assigned to multiple sessions, these values are generated out of order when all sessions are considered. For example, if the cache parameter is set to 10, Session A may retain values from 1 to 10 and return NEXTVAL=1. Then, Session B may retain values from 11 to 20 and return NEXTVAL=11 before Session A generates NEXTVAL=2. Therefore, if the cache parameter is set to 1, NEXTVAL values are generated sequentially. If the cache parameter is set to a value greater than 1, NEXTVAL values are different and may not be generated sequentially. The last value reflects the latest value retained by any session no matter whether the value has been returned by NEXTVAL.

### Examples

Create an ascending sequence named serial, that starts from 101:

```
CREATE SEQUENCE serial START WITH 101;
```

Select the next number from this sequence:

```
SELECT serial.NEXTVAL FROM DUAL;

 nextval
---------
     101
(1 row)
```

Create a sequence named supplier_seq and specify the NOCACHE option.

```
CREATE SEQUENCE supplier_seq
    MINVALUE 1
    START WITH 1
    INCREMENT BY 1
    NOCACHE;
```

Select the next number from this sequence:

```
SELECT supplier_seq.NEXTVAL FROM DUAL;

 nextval
---------
       1
(1 row)
```

## 12.30 CREATE SYNONYM

Creates a synonym.

### Syntax

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema.]syn_name
       FOR object_schema.object_name[@dblink_name];
```

### Description

The CREATE SYNONYM command creates a synonym for specific types of database objects . PolarDB databases compatible with Oracle support synonyms for the following types of database objects:

- Tables
- Views
- Materialized views
- Sequences
- Stored procedures
- Stored functions
- Types
- Objects that are accessible through a database link
- Other synonyms

### Parameters

| Parameter | Description |
|---|---|
| syn_name | The name of the synonym. A synonym name must be unique within a schema. |
| schema | The name of the schema where the synonym resides. If you do not specify a schema name, the synonym is created in the first existing schema in your search path. |

| Parameter | Description |
|---|---|
| object_name | The name of the object. |
| object_schema | The name of the schema where the referenced object resides. |
| dblink_name | The name of the database link through which an object is accessed. |

You can use the REPLACE clause to replace an existing synonym definition with a new synonym definition.

You can use the PUBLIC clause to create the synonym in the public schema. The PUBLIC SYNONYM command is compatible with Oracle databases. You can use this command to create a synonym that resides in the public schema.

```
CREATE [OR REPLACE] PUBLIC SYNONYM syn_name FOR object_schema.object_name;
```

The following statement is a short form:

```
CREATE [OR REPLACE] SYNONYM public.syn_name FOR object_schema.object_name;
```

### Notes

Access to the object referenced by the synonym is determined by the permissions of the current user. The synonym user must have proper permissions on the underlying database object.

### Examples

Create a synonym for the emp table in a schema named enterprisedb:

```
CREATE SYNONYM personnel FOR enterprisedb.emp;
```

## 12.31 CREATE TABLE

Creates a table.

### Syntax

```
CREATE [ GLOBAL TEMPORARY ] TABLE table_name (
  { column_name data_type [ DEFAULT default_expr ]
  [ column_constraint [ ... ] ] | table_constraint } [, ...]
  )
  [ ON COMMIT { PRESERVE ROWS | DELETE ROWS } ]
[ TABLESPACE tablespace ]
```

where column_constraint is:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
 NULL |
 UNIQUE [ USING INDEX TABLESPACE tablespace ] |
 PRIMARY KEY [ USING INDEX TABLESPACE tablespace ] |
 CHECK (expression) |
 REFERENCES reftable [ ( refcolumn ) ]
  [ ON DELETE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED |
 INITIALLY IMMEDIATE ]
```

table_constraint is:

```
[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ...] )
  [ USING INDEX TABLESPACE tablespace ] |
 PRIMARY KEY ( column_name [, ...] )
  [ USING INDEX TABLESPACE tablespace ] |
 CHECK ( expression ) |
 FOREIGN KEY ( column_name [, ...] )
   REFERENCES reftable [ ( refcolumn [, ...] ) ]
  [ ON DELETE action ] }
[ DEFERRABLE | NOT DEFERRABLE ]
[ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

**Description**

The CREATE TABLE command creates an empty table in the current database. The table is owned by the user who runs the command.

If you specify a schema name (for example, you specify CREATE TABLE myschema.mytable), the table is created in the specified schema. Otherwise, the table is created in the current schema. Temporary tables exist in a special schema. Therefore, you do not need to specify the schema name when creating a temporary table. The table name must be different from all other tables, sequences, indexes, or views in the same schema.

The CREATE TABLE command automatically creates a composite data type that corresponds to a row in the table. Therefore, a table cannot have the same name as an existing data type in the same schema.

A table can have up to 1,600 columns. In practice, the effective limit is lower because of tuple-length constraints

The optional constraint clauses specify constraints or tests that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table.

Constraints are classified into table constraints and column constraints. A column constraint is defined as part of a column definition. The table constraint definition does not depend on specific columns and can contain multiple columns. Each column constraint can also be written as a table constraint. If a constraint affects only one column, the constraint is a column constraint. This means that a column constraint is only a notational convenience.

**Parameters**

| Parameter | Description |
|---|---|
| GLOBAL TEMPORARY | If this parameter is specified, the table is created as a temporary table. Temporary tables are automatically deleted at the end of a session or at the end of the current transaction. For more information, see the ON COMMIT parameter. If a temporary table exists, existing permanent tables with the same names are invisible to the current session, unless the temporary table is referenced by schema-qualified names. A temporary table is invisible outside the session in which it was created. This aspect of global temporary tables is incompatible with Oracle databases. All indexes created on a temporary table are automatically temporary. |
| table_name | The name of the table to be created. The name can be schema-qualified. |
| column_name | The name of a column to be created in the new table. |
| data_type | The data type of the column. Array specifiers can be included. |
| DEFAULT default_expr | The DEFAULT clause assigns a default data value for the column. The value is a variable-free expression. Subqueries or cross-references to other columns in the current table are not allowed. The data type of the default expression must be the same as that of the column.<br>**Note:**<br>The default expression is used in an insert operation that does not specify a value for the column. If no default value is specified for the column, the default value is null. |
| CONSTRAINT constraint_name | An optional name for a column or table constraint. If this parameter is not specified, the system generates a name. |
| NOT NULL | The column cannot contain null values. |

| Parameter | Description |
|---|---|
| NULL | The column can contain null values. This is the default value.<br>This clause is available only for compatibility with non-standard SQL databases. We recommend that you do not use this clause in new applications. |
| UNIQUE: column constraint<br>UNIQUE ( column_name [, ...] ): table constraint | The UNIQUE constraint specifies that a group of one or more distinct columns of a table can contain only unique values. The behavior of a unique table constraint is the same as that of a column constraint except the additional capability to span multiple columns.<br>When a unique constraint is evaluated, null values are not considered to be equal.<br>Each unique table constraint must name a set of columns that is different from the set of columns named by other unique or primary key constraints defined for the table. Otherwise, the same constraint is listed twice. |
| PRIMARY KEY: column constraint<br>PRIMARY KEY ( column_name [, ...] ): table constraint | The primary key constraint specifies that one or more columns of a table can contain only unique, non-duplicate, and non-null values. PRIMARY KEY is a combination of UNIQUE and NOT NULL. PRIMARY KEY identifies a set of columns as the primary key and provides metadata about the design of the schema. A primary key implies that other tables can rely on this set of columns as a unique identifier for rows.<br>Only one primary key can be specified for a table, whether as a column constraint or a table constraint.<br>The primary key constraint must name a set of columns that is different from other sets of columns named by a unique constraint defined for the same table. |

| Parameter | Description |
|---|---|
| CHECK (expression) | The CHECK clause specifies an expression that produces a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. If an expression is evaluated as TRUE or unknown, the INSERT or UPDATE operation is successful. If a row of an insert or update operation produces a FALSE result, an error occurs and the insert or update does not alter the database. A check constraint specified as a column constraint must reference only the value of the column. An expression in a table constraint can reference multiple columns.<br>CHECK expressions cannot contain subqueries or reference variables other than columns of the current row. |
| REFERENCES reftable [ ( refcolumn ) ] [ ON DELETE action ]: column constraint<br>FOREIGN KEY ( column [, ...] ) REFERENCES reftable [ ( refcolumn [, ...] ) ] [ ON DELETE action ]: table constraint | These clauses specify a foreign key constraint. A group of one or more columns in the new table must contain only values that match the values in the referenced columns of a row in the referenced table. If refcolumn is omitted, the primary key of the reftable is used. The referenced columns must be the columns of a unique or primary key constraint in the referenced table.<br>In addition, when data in the referenced columns is changed, actions are performed on the data in the columns of this table. The ON DELETE clause specifies the action to perform when a referenced row in the referenced table is being deleted. Referential actions cannot be deferred even if the constraint is deferrable. Possible actions for each clause are described as follows:<br>• CASCADE: deletes all rows that reference the deleted row, or updates the value of the referencing column to the new value of the referenced column.<br>• SET NULL: sets the referencing columns to NULL.<br>If the referenced column changes frequently, you can add an index to the foreign key column to facilitate reference actions associated with the foreign key column. |

| Parameter | Description |
|---|---|
| DEFERRABLE<br>NOT DEFERRABLE | This parameter controls whether the constraint can be deferred. A constraint that is not deferrable is checked immediately after each command. Checking of deferrable constraints can be postponed until the end of the transaction by using the SET CONSTRAINTS command. NOT DEFERRABLE is the default value. Only foreign key constraints accept this clause. All other constraint types are not deferrable. |
| INITIALLY IMMEDIATE<br>INITIALLY DEFERRED | If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is INITIALLY IMMEDIATE, it is checked after each statement. This is the default value. If the constraint is INITIALLY DEFERRED, it is checked only at the end of the transaction. You can use the SET CONSTRAINTS command to change the constraint check time. |
| ON COMMIT | You can use the ON COMMIT clause to control the behavior of temporary tables at the end of a transaction block. The following options are available:<br>• PRESERVE ROWS: No special action is performed at the end of each transaction. This is the default behavior. Note that this aspect is incompatible with Oracle databases. The default value for Oracle databases is DELETE ROWS.<br>• DELETE ROWS: All rows in the temporary table are deleted at the end of each transaction block. An automatic TRUNCATE command is executed at each commit operation. |
| TABLESPACE tablespace | The tablespace is the name of the tablespace in which the new table is to be created. If you do not specify the tablespace, default_tablespace is used. If default_tablespace is an empty string, the default tablespace of the database is used. |
| USING INDEX TABLESPACE tablespace | This clause allows you to select the tablespace in which the index associated with a UNIQUE or PRIMARY KEY constraint is created. If you do not specify the tablespace, default_tablespace is used. If default_tablespace is an empty string, the default tablespace of the database is used. |

**Note:**

The PolarDB database compatible with Oracle automatically creates an index for each unique constraint and primary key constraint to enforce the uniqueness. You do not need to create an explicit index for primary key columns. For more information, see the CREATE INDEX command.

### Examples

Create the dept table and the emp table:

```
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14),
    loc             VARCHAR2(13)
);
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
```

Define a unique table constraint for the dept table. Unique table constraints can be defined on one or more columns of the table.

```
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc             VARCHAR2(13)
);
```

Define a check column constraint:

```
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
```

Define a check table constraint:

```
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno),
    CONSTRAINT new_emp_ck CHECK (ename IS NOT NULL AND empno > 7000)
);
```

Define a primary key table constraint for the jobhist table. Primary key table constraints can be defined on one or more columns of the table.

```
CREATE TABLE jobhist (
    empno           NUMBER(4) NOT NULL,
    startdate       DATE NOT NULL,
    enddate         DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2),
    chgdesc         VARCHAR2(80),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate)
);
```

Assign a literal constant default value for the column job and set the default value of hiredate to the date at which the row is inserted.

```
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9) DEFAULT 'SALESMAN',
    mgr             NUMBER(4),
    hiredate        DATE DEFAULT SYSDATE,
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
```

Create the dept table in the diskvol1 tablespace:

```
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14),
    loc             VARCHAR2(13)
) TABLESPACE diskvol1;
```

## 12.32 CREATE TABLE AS

Creates a table based on the results of a query.

### Syntax

```
CREATE [ GLOBAL TEMPORARY ] TABLE table_name
  [ (column_name [, ...] ) ]
  [ ON COMMIT { PRESERVE ROWS | DELETE ROWS } ]
  [ TABLESPACE tablespace ]
  AS query
```

### Description

The CREATE TABLE AS command creates a table and fills it with data computed by a SELECT command. The table columns have the same names and data types as the output columns of the SELECT command. However, you can override the column names by specifying an explicit list of new column names.

The CREATE TABLE AS command is similar to creating a view. However, unlike creating a view, the CREATE TABLE AS command creates a new table and evaluates the query only once to fill the new table. The new table does not track subsequent changes to the source tables of the query. In contrast, a view evaluates its defining SELECT statement whenever it is queried.

### Parameters

| Parameter | Description |
|---|---|
| GLOBAL TEMPORARY | If this parameter is specified, the table is created as a temporary table. For more information, see the CREATE TABLE command. |
| table_name | The name of the table to be created. The name can be schema-qualified. |
| column_name | The name of a column to be created in the new table. If no column names are specified, the names of columns in the query result are used. |
| query | A query statement. It is also a SELECT command. For more information about the supported syntax, see the SELECT command. |

## 12.33 CREATE TRIGGER

Creates a trigger.

### Syntax

```
CREATE [ OR REPLACE ] TRIGGER name
  { BEFORE | AFTER | INSTEAD OF }
  { INSERT | UPDATE | DELETE }
     [ OR { INSERT | UPDATE | DELETE } ] [, ...]
    ON table
  [ REFERENCING { OLD AS old | NEW AS new } ...]
  [ FOR EACH ROW ]
  [ WHEN condition ]
  [ DECLARE
     [ PRAGMA AUTONOMOUS_TRANSACTION; ]
     declaration; [, ...] ]
    BEGIN
     statement; [, ...]
```

```
[ EXCEPTION
  { WHEN exception [ OR exception ] [...] THEN
      statement; [, ...] } [, ...]
]
  END
```

**Description**

CREATE TRIGGER creates a trigger. CREATE OR REPLACE TRIGGER either creates a new trigger or replaces an existing definition.

If you use the CREATE TRIGGER keywords to create a new trigger, the name of the new trigger must be different from an existing trigger that is defined on the same table. New triggers are created in the same schema as the table on which the triggering event is defined.

To update the definition of an existing trigger, you can use the CREATE OR REPLACE TRIGGER keywords.

If you use syntax that is compatible with Oracle to create a trigger, the trigger runs as a SECURITY DEFINER function.

**Parameters**

| Parameter | Description |
|---|---|
| name | The name of the trigger to be created. |
| BEFORE \| AFTER | Specifies whether the trigger is executed before or after the triggering event. |
| INSERT \| UPDATE \| DELETE | The triggering event. |
| table | The name of the table or view on which the triggering event occurs . |
| condition | condition is a Boolean expression that determines whether the trigger is executed. If condition evaluates to TRUE, the trigger is executed.<br>• If the trigger definition includes the FOR EACH ROW keywords, the WHEN clause can reference the columns of the old or new row values by writing OLD.column_name or NEW.column_name respectively. INSERT triggers cannot reference OLD. DELETE triggers cannot reference NEW.<br>• If the trigger includes the INSTEAD OF keywords, it may not include a WHEN clause. A WHEN clause cannot contain subqueries. |

| Parameter | Description |
|---|---|
| REFERENCING { OLD AS old \| NEW AS new } ... | The REFERENCING clause that is used to reference old rows and new rows. The old value can be replaced only by an identifier named old or an equivalent that is saved in lowercase. For example, the statement can be REFERENCING OLD AS old, REFERENCING OLD AS OLD, or REFERENCING OLD AS "old". In addition, the new value can be replaced only by an identifier named new or an equivalent that is saved in lowercase. For example, the statement can be REFERENCING NEW AS new, REFERENCING NEW AS NEW, or REFERENCING NEW AS "new".<br>You can specify one or both of the following phrases in the REFERENCING clause: OLD AS old and NEW AS new. For example, you can specify REFERENCING NEW AS New OLD AS Old.<br>**Note:**<br>This clause is incompatible with Oracle databases because you cannot use identifiers other than old and new. |
| FOR EACH ROW | Specifies whether the trigger is executed for each row that is affected by the triggering event or only once by each SQL statement. If specified, a row-level trigger is executed for each affected row. Otherwise, a statement-level trigger is executed. |
| PRAGMA AUTONOMOUS _TRANSACTION | PRAGMA AUTONOMOUS_TRANSACTION is the directive that sets the trigger as an autonomous transaction. |
| declaration | A variable, type, REF CURSOR, or subprogram declaration. If subprogram declarations are included, they must be declared after all other variable, type, and REF CURSOR declarations. |
| statement | An SPL program statement. Note that a DECLARE - BEGIN - END block is considered an SPL statement. Therefore, the trigger body can contain nested blocks. |
| exception | The name of an exception condition, such as NO_DATA_FOUND and OTHERS. |

### Examples

The following statement-level trigger is executed after the trigger statement (INSERT, UPDATE, or DELETE on table emp) is executed.

```
CREATE OR REPLACE TRIGGER user_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_action        VARCHAR2(24);
```

```
BEGIN
    IF INSERTING THEN
        v_action := ' added employee(s) on ';
    ELSIF UPDATING THEN
        v_action := ' updated employee(s) on ';
    ELSIF DELETING THEN
        v_action := ' deleted employee(s) on ';
    END IF;
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
        TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
```

The following row-level trigger is executed before each row is inserted, updated, or deleted in the emp table.

```
CREATE OR REPLACE TRIGGER emp_sal_trig
    BEFORE DELETE OR INSERT OR UPDATE ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    IF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE('Inserting employee ' || :NEW.empno);
        DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
    END IF;
    IF UPDATING THEN
        sal_diff := :NEW.sal - :OLD.sal;
        DBMS_OUTPUT.PUT_LINE('Updating employee ' || :OLD.empno);
        DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
        DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
        DBMS_OUTPUT.PUT_LINE('..Raise     : ' || sal_diff);
    END IF;
    IF DELETING THEN
        DBMS_OUTPUT.PUT_LINE('Deleting employee ' || :OLD.empno);
        DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
    END IF;
END;
```

## 12.34 CREATE TYPE

Creates a user-defined type, which can be an object type, a collection type (a nested table type or a varray type), or a composite type.

### Syntax

Object type

```
CREATE [ OR REPLACE ] TYPE name
  [ AUTHID { DEFINER | CURRENT_USER } ]
  { IS | AS } OBJECT
( { attribute { datatype | objtype | collecttype } }
    [, ...]
  [ method_spec ] [, ...]
) [ [ NOT ] { FINAL | INSTANTIABLE } ] ...
```

where method_spec is:

```
[ [ NOT ] { FINAL | INSTANTIABLE } ] ...
[ OVERRIDING ]
  subprogram_spec
```

subprogram_spec is:

```
{ MEMBER | STATIC }
{ PROCEDURE proc_name
    [ ( [ SELF [ IN | IN OUT ] name ]
        [, argname [ IN | IN OUT | OUT ] argtype
                [ DEFAULT value ]
        ] ...)
    ]
|
  FUNCTION func_name
    [ ( [ SELF [ IN | IN OUT ] name ]
        [, argname [ IN | IN OUT | OUT ] argtype
                [ DEFAULT value ]
        ] ...)
    ]
  RETURN rettype
}
```

Nested table type

```
CREATE [ OR REPLACE ] TYPE name { IS | AS } TABLE OF
  { datatype | objtype | collecttype }
```

Varray type

```
CREATE [ OR REPLACE ] TYPE name { IS | AS }
  { VARRAY | VARYING ARRAY } (maxsize) OF { datatype | objtype }
```

Composite type

```
CREATE [ OR REPLACE ] TYPE name { IS | AS }
( [ attribute datatype ][, ...]
)
```

## Description

The CREATE TYPE command creates a user-defined data type. The types that can be created include object type, nested table type, varray type, and composite type. The nested table type and varray type belong to the collection type.

Composite types are incompatible with Oracle databases. However, composite types can be accessed through SPL programs, which is the same as other types described in this topic .

**Note:**

For packages only, a composite type can be included in a user-defined record type declared using the TYPE IS RECORD statement within the package specification or package body. Such nested structure is not allowed in other SPL programs such as functions, stored procedures, and triggers.

If you specify a schema name in the CREATE TYPE command, the type is created in the specified schema. Otherwise, the type is created in the current schema. The name of a new type must be different from an existing type in the same schema. If you want to update the definition of an existing type, you can use the CREATE OR REPLACE TYPE command.

**Note:**

- The OR REPLACE option cannot be used to add, delete, or modify the attributes of an existing object type. However, you can use the DROP TYPE command to delete the existing object type. The OR REPLACE option can be used to add, delete, or modify the methods in an existing object type.
- The PostgreSQL form of the ALTER TYPE ALTER ATTRIBUTE command can be used to change the data type of an attribute in an existing object type. However, the ALTER TYPE command cannot add or delete attributes in the object type.

The user that creates the type is the owner of the type.

**Parameters**

| Parameter | Description |
|---|---|
| name | The name of the type to be created. The name can be schema-qualified. |
| DEFINER \| CURRENT_USER | Specifies the privileges that are used to determine whether access is allowed to database objects referenced in the object type. DEFINER indicates the privileges of the object type owner . CURRENT_USER indicates the privileges of the current user executing a method in the object type. The default value is DEFINER . |
| attribute | The name of an attribute in the object type or composite type. |
| datatype | The data type that defines an attribute of the object type or composite type, or the elements of the collection type that is being created. |

| Parameter | Description |
|---|---|
| objtype | The name of an object type that defines an attribute of the object type or the elements of the collection type that is being created. |
| collecttype | The name of a collection type that defines an attribute of the object type or the elements of the collection type that is being created. |
| FINAL \| NOT FINAL | - For an object type, this parameter specifies whether a subtype can be derived from the object type. The default value is FINAL, indicating that no subtype can be derived from the object type.<br>- For method_spec, this parameter specifies whether to override the method in a subtype. The default value is NOT FINAL, indicating that the method can be overridden in a subtype. |
| INSTANTIABLE \| NOT INSTANTIABLE | - For an object type, this parameter specifies whether an instance of this object type can be created. The default value is INSTANTIABLE, indicating that an instance of this object type can be created. If you specify NOT INSTANTIABLE, you must also specify NOT FINAL. If method_spec for a method in the object type contains the NOT INSTANTIABLE qualifier, the object type must be defined with NOT INSTANTIABLE and NOT FINAL following the closing parenthesis of the object type specification .<br>- For method_spec, this parameter specifies whether the object type definition provides an implementation for the method. The default value is INSTANTIABLE, indicating that the CREATE TYPE BODY command for the object type provides the implementation of the method. If you specify NOT INSTANTIABLE, the CREATE TYPE BODY command for the object type cannot contain the implementation of the method. |
| OVERRIDING | If you specify OVERRIDING, method_spec overrides an identically named method with the same number of identically named method arguments. The arguments have the same data types, the same order, and the same return type (if the method is a function) as defined in a supertype. |
| MEMBER \| STATIC | If the subprogram runs on an object instance, specify MEMBER. If the subprogram runs independently of a particular object instance , specify STATIC. |
| proc_name | The name of the stored procedure to be created. |

| Parameter | Description |
|---|---|
| SELF [ IN \| IN OUT ] name | For a member method, an implicit built-in parameter named SELF is available. The data type of this parameter is the data type of the object type being created. SELF references the object instance that is calling the method. SELF can be explicitly declared as an IN or IN OUT parameter in the parameter list. If explicitly declared, SELF must be the first parameter in the parameter list. If SELF is not explicitly declared, its parameter mode defaults to IN OUT for member stored procedures and IN for member functions. |
| argname | The name of an argument. The argument is referenced by this name in the method body. |
| argtype | The data types of the method arguments. The argument types can be a base data type or a user-defined type such as a nested table type or an object type. You cannot specify the length of a base data type. For example, you can specify VARCHAR2 rather than VARCHAR2(10). |
| DEFAULT value | If no default value is specified in the method call, this parameter specifies a default value for an input argument. DEFAULT may not be specified for arguments with the IN OUT or OUT mode. |
| func_name | The name of the function to be created. |
| rettype | The return data type, which can be one of the types listed for the argtype parameter. For argtype, you cannot specify a length for rettype. |
| maxsize | The maximum number of elements in the varray. |

**Examples**

- Create an object type

  Create an object type named addr_obj_typ.

  ```
  CREATE OR REPLACE TYPE addr_obj_typ AS OBJECT (
      street          VARCHAR2(30),
      city            VARCHAR2(20),
      state           CHAR(2),
      zip             NUMBER(5)
  );
  ```

  Create an object type named emp_obj_typ that contains a member method display_emp.

  ```
  CREATE OR REPLACE TYPE emp_obj_typ AS OBJECT (
      empno           NUMBER(4),
      ename           VARCHAR2(20),
      addr            ADDR_OBJ_TYP,
  ```

```
    MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
);
```

Create an object type named dept_obj_typ that contains a static method get_dname.

```
CREATE OR REPLACE TYPE dept_obj_typ AS OBJECT (
    deptno          NUMBER(2),
    STATIC FUNCTION get_dname (p_deptno IN NUMBER) RETURN VARCHAR2,
    MEMBER PROCEDURE display_dept
);
```

- Creating a collection type

  Create a nested table type named budget_tbl_typ of data type NUMBER(8,2).

```
CREATE OR REPLACE TYPE budget_tbl_typ IS TABLE OF NUMBER(8,2);
```

- Create and use a composite type

  The following example shows how to access a composite type from an anonymous block
  .

  The composite type is created as follows:

```
CREATE OR REPLACE TYPE emphist_typ AS (
    empno           NUMBER(4),
    ename           VARCHAR2(10),
    hiredate        DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2)
);
```

The following example shows the anonymous block that accesses the composite type:

```
DECLARE
    v_emphist       EMPHIST_TYP;
BEGIN
    v_emphist.empno    := 9001;
    v_emphist.ename    := 'SMITH';
    v_emphist.hiredate := '01-AUG-17';
    v_emphist.job      := 'SALESMAN';
    v_emphist.sal      := 8000.00;
    DBMS_OUTPUT.PUT_LINE('   EMPNO: ' || v_emphist.empno);
    DBMS_OUTPUT.PUT_LINE('   ENAME: ' || v_emphist.ename);
    DBMS_OUTPUT.PUT_LINE('HIREDATE: ' || v_emphist.hiredate);
    DBMS_OUTPUT.PUT_LINE('     JOB: ' || v_emphist.job);
    DBMS_OUTPUT.PUT_LINE('     SAL: ' || v_emphist.sal);
END;

   EMPNO: 9001
   ENAME: SMITH
HIREDATE: 01-AUG-17 00:00:00
     JOB: SALESMAN
```

```
  SAL: 8000.00
```

The following example shows how to access a composite type from a user-defined record type that is declared in a package body.

The composite type is created as follows:

```
CREATE OR REPLACE TYPE salhist_typ AS (
    startdate       DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2)
);
```

The package specification is defined as follows:

```
CREATE OR REPLACE PACKAGE emp_salhist
IS
    PROCEDURE fetch_emp (
        p_empno     IN NUMBER
    );
END;
```

The package body is defined as follows:

```
CREATE OR REPLACE PACKAGE BODY emp_salhist
IS
    TYPE emprec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR(10),
        salhist     SALHIST_TYP
    );
    TYPE emp_arr_typ IS TABLE OF emprec_typ INDEX BY BINARY_INTEGER;
    emp_arr         emp_arr_typ;

    PROCEDURE fetch_emp (
        p_empno     IN NUMBER
    )
    IS
        CURSOR emp_cur IS SELECT e.empno, e.ename, h.startdate, h.job, h.sal
            FROM emp e, jobhist h
            WHERE e.empno = p_empno
              AND e.empno = h.empno;

        i           INTEGER := 0;
    BEGIN
        DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    STARTDATE  JOB         ' ||
        'SAL        ');
        DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '---------');

        FOR r_emp IN emp_cur LOOP
            i := i + 1;
            emp_arr(i) := (r_emp.empno, r_emp.ename,
                (r_emp.startdate, r_emp.job, r_emp.sal));
        END LOOP;

        FOR i IN 1 .. emp_arr.COUNT LOOP
            DBMS_OUTPUT.PUT_LINE(emp_arr(i).empno || '   ' ||
                RPAD(emp_arr(i).ename,8) || ' ' ||
                TO_CHAR(emp_arr(i).salhist.startdate,'DD-MON-YY') || '  ' ||
            RPAD(emp_arr(i).salhist.job,10) || ' ' ||
            TO_CHAR(emp_arr(i).salhist.sal,'99,999.99'));
        END LOOP;
    END;
END;
```

Note that in the declaration of the TYPE emprec_typ IS RECORD data structure in the package body, the salhist field is of the SALHIST_TYP composite type that is created by the CREATE TYPE salhist_typ statement.

The associative array definition TYPE emp_arr_typ IS TABLE OF emprec_typ references the record type data structure emprec_typ. The data structure includes the salhist field of the SALHIST_TYP composite type.

The following example shows how to call the package stored procedure that loads the array from a join of the emp and jobhist tables and displays the array content.

```
EXEC emp_salhist.fetch_emp(7788);

EMPNO  ENAME    STARTDATE  JOB        SAL
-----  -------  ---------  ---------  ---------
7788   SCOTT    19-APR-87  CLERK       1,000.00
7788   SCOTT    13-APR-88  CLERK       1,040.00
7788   SCOTT    05-MAY-90  ANALYST     3,000.00

EDB-SPL Procedure successfully completed
```

## 12.35 CREATE TYPE BODY

Defines a new object type body.

**Syntax**

```
CREATE [ OR REPLACE ] TYPE BODY name
  { IS | AS }
 method_spec [...]
END
```

Where method_spec is:

```
subprogram_spec
```

and subprogram_spec is:

```
  { MEMBER | STATIC }
  { PROCEDURE proc_name
     [ ( [ SELF [ IN | IN OUT ] name ]
         [, argname [ IN | IN OUT | OUT ] argtype
                 [ DEFAULT value ]
         ] ...)
     ]
  { IS | AS }
 program_body
```

```
    END;
  |
    FUNCTION func_name
      [ ( [ SELF [ IN | IN OUT ] name ]
          [, argname [ IN | IN OUT | OUT ] argtype
                 [ DEFAULT value ]
          ] ...)
      ]
    RETURN rettype
  { IS |AS }
 program_body
    END;
  }
```

### Description

Use CREATE TYPE BODY to define a new object type body. Use CREATE OR REPLACE TYPE BODY to either create a new object type body, or replace an existing body.

If a schema name is included, the object type body is created in the specified schema. Otherwise, the object type body is created in the current schema. The name of the new object type body must match an existing object type specification in the same schema. The new object type body name must not match any existing object type body in the same schema unless you want to update the definition of an existing object type body. In which case, you can use CREATE OR REPLACE TYPE BODY.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the object type for which a body is to be created. The name may be optional and schema-qualified. |
| MEMBER \| STATIC | Specify MEMBER if the subprogram runs on an object instance . Specify STATIC if the subprogram runs independently of any particular object instance. |
| proc_name | The name of the procedure to create. |
| SELF [ IN \| IN OUT ] name | For a member method, there is an implicit and built-in parameter named SELF. The data type of this parameter is the data type of the object type that is defined. SELF refers to the object instance that is invoking the method. SELF can be explicitly declared as an IN or IN OUT parameter in the parameter list. If explicitly declared, the SELF parameter must be the first in the parameter list. If the SELF parameter is not explicitly declared, the default parameter mode is IN OUT for member procedures and IN for member functions. |
| argname | The name of an argument. The argument is referenced by this name within the method body. |

| Parameter | Description |
|---|---|
| argtype | The data type(s) of the arguments. The argument types may be a base data type or a user-defined type such as a nested table or an object type. The basic data type cannot be specified a length. For example, you must specify VARCHAR2 instead of VARCHAR2(10) as the data type. |
| DEFAULT value | This parameter provides a default value for an input argument if no default value is provided in the method call. DEFAULT may not be specified for arguments with the IN OUT or OUT modes. |
| program_body | The pragma, declarations, and SPL statements that comprise the body of the function or procedure. The pragma can be PRAGMA AUTONOMOUS_TRANSACTION to set the function or procedure as an autonomous transaction. |
| func_name | The name of the function to create. |
| rettype | The data type returned. It can be any of the types listed for argtype . For argtype, a length must not be specified for rettype. |

**Example**

The following example shows how to create the object type body for the emp_obj_typ object type that is created by the CREATE TYPE command.

```
CREATE OR REPLACE TYPE BODY emp_obj_typ AS
    MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Employee No   : ' || empno);
        DBMS_OUTPUT.PUT_LINE('Name          : ' || ename);
        DBMS_OUTPUT.PUT_LINE('Street        : ' || addr.street);
        DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || addr.city || ', ' ||
            addr.state || ' ' || LPAD(addr.zip,5,'0'));
    END;
END;
```

The following example shows how to create the object type body for the dept_obj_typ object type that is created by the CREATE TYPE command.

```
CREATE OR REPLACE TYPE BODY dept_obj_typ AS
    STATIC FUNCTION get_dname (p_deptno IN NUMBER) RETURN VARCHAR2
    IS
        v_dname     VARCHAR2(14);
    BEGIN
        CASE p_deptno
            WHEN 10 THEN v_dname := 'ACCOUNING';
            WHEN 20 THEN v_dname := 'RESEARCH';
            WHEN 30 THEN v_dname := 'SALES';
            WHEN 40 THEN v_dname := 'OPERATIONS';
            ELSE v_dname := 'UNKNOWN';
        END CASE;
        RETURN v_dname;
    END;
    MEMBER PROCEDURE display_dept
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Dept No    : ' || SELF.deptno);
        DBMS_OUTPUT.PUT_LINE('Dept Name  : ' ||
            dept_obj_typ.get_dname(SELF.deptno));
    END;
END;
```

## 12.36 CREATE VIEW

Creates a view.

### Syntax

```
CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ]
  AS query
```

### Description

You can use the CREATE VIEW command to define a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

CREATE OR REPLACE VIEW is similar, but if a view of the same name already exists, the name is replaced.

If a schema name is specified (for example, CREATE VIEW myschema.myview...), the view is created in the specified schema. Otherwise, it is created in the current schema. The view name must be different from the name of any other view, table, sequence, or index in the same schema.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of a view to be created. The name can be schema-qualified. |
| column_name | An optional list of columns names in the view. If not specified, the column names are deduced from the query. |
| query | A query (a SELECT statement), which provides the columns and rows of the view. |

**Note:**

For more information about valid queries, see the SELECT topic.

### Notes

Views are read-only. The system does not allow the insert, update, or delete operations on views. You can obtain the effect of an updatable view by creating rules that convert the insert operations on the view into appropriate operations on other tables.

Access to tables referenced in the view is determined by permissions of the view owner. However, the functions that are called in the view are treated the same as those called from the query by using the view. Therefore, the user of a view must have permissions to call all functions that are used by the view.

### Examples

Create a view that consists of all employees in department 30:

```
CREATE VIEW dept_30 AS SELECT * FROM emp WHERE deptno = 30;
```

## 12.37 DELETE

Deletes rows of a table.

### Syntax

```
DELETE [ optimizer_hint ] FROM table[@dblink ]
  [ WHERE condition ]
  [ RETURNING return_expression [, ...]
      { INTO { record | variable [, ...] }
      | BULK COLLECT INTO collection [, ...] } ]
```

### Description

You can use the DELETE command to delete rows that satisfy the WHERE clause from the specified table. If you do not specify the WHERE clause, all rows in the table are deleted. The result is valid, and the table becomes empty.

**Note:**

The TRUNCATE command provides a faster mechanism to delete all rows from a table.

If you use the DELETE command within an SPL program, you can specify the RETURNING INTO { record | variable [, ...] } clause. In addition, the result set of the DELETE command must not include multiple rows. Otherwise, an exception occurs. If the result set is empty, the content of the target record or variables is set to null.

If you use the DELETE command within an SPL program, you can specify the RETURNING BULK COLLECT INTO collection [, ...] clause. If you specify multiple collection as the target of the BULK COLLECT INTO clause, each collection must consist of a single scalar field. collection must not be a record. The result set of the DELETE command may contain zero, one, or more rows. return_expression evaluated for each row of the result set becomes an element in collection, starting from the first element. Existing rows in collection are deleted. If the result set is empty, collection is empty.

You must have the DELETE privilege on the table to delete rows from it, and the SELECT privilege on tables whose data is read in the condition.

**Parameters**

| Parameter | Description |
|---|---|
| optimizer_hint | Comment-embedded hints to the optimizer, which is used to select execution plan. |
| table | The name of an existing table. The name can be schema-qualified. |
| dblink | The database link name, which identifies a remote database. For more information about database links, see the CREATE DATABASE LINK command. |
| condition | A value expression that returns a value of the BOOLEAN type. The value expression determines the rows to be deleted. |
| return_expression | An expression that can include one or more columns in table. If a column name in table is specified in return_expression, the value substituted for the column when return_expression is evaluated is the value from the deleted row. |
| record | A record to whose field you want to assign the evaluation result of return_expression. For example, the first return_expression is assigned to the first field in record, and the second return_expression is assigned to the second field in record. The number of fields in record must match the number of expressions, and the fields must be type-compatible with the corresponding expressions. |
| variable | A variable to which you want to assign the evaluation result of return_expression. If you specify multiple return_expression and variable, the first return_expression is assigned to the first variable, the second return_expression is assigned to the second variable. The number of the specified variables that follow the INTO keyword must match the number of expressions that follow the RETURNING keyword, and the variables must be type-compatible with the corresponding expressions. |

| Parameter | Description |
|---|---|
| collection | A collection in which an element is created from the evaluated return_expression. You can specify a collection of a single field or a collection of a record type. You can also specify multiple collections where each collection consists of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each return_expression must be type-compatible with the corresponding collection field. |

### Examples

Delete all rows for employee 7900 from the jobhist table:

```
DELETE FROM jobhist WHERE empno = 7900;
```

Clear the jobhist table:

```
DELETE FROM jobhist;
```

## 12.38 DROP DATABASE LINK

Deletes a database link.

### Syntax

```
DROP [ PUBLIC ] DATABASE LINK name
```

### Description

You can use the DROP DATABASE LINK command to drop existing database links. To run this command on a database link, you must be the owner of the database link.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of a database link to be deleted. |
| PUBLIC | Specifies that name is a public database link. |

### Examples

Delete public database link whose name is `oralink`:

```
DROP PUBLIC DATABASE LINK oralink;
```

Delete the public database link whose name is `edblink`:

```
DROP DATABASE LINK edblink;
```

## 12.39 DROP FUNCTION

Remove a function.

### Syntax

```
DROP FUNCTION [ IF EXISTS ] name
  [ ([ [ argmode ] [ argname ] argtype ] [, ...]) ]
  [ CASCADE | RESTRICT ]
```

### Description

You can run the `DROP FUNCTION` command to remove an existing function. To run this command, you must be a superuser or the owner of the function. All data types of the input argument in the mode of `IN` or `IN OUT` to the function must be specified if this is an overloaded function. This requirement is not compatible with Oracle databases. In Oracle, only the function name is specified. PolarDB database compatible with Oracle allows overloading of function names, so the function signature provided by the input argument data types is required in the DROP FUNCTION command of an overloaded function.

The usage of `IF EXISTS`, `CASCADE`, or `RESTRICT` is not compatible with Oracle databases and is used only by PolarDB database compatible with Oracle.

### Parameters

| Parameter | Description |
|---|---|
| IF EXISTS | Dose not throw an error if the function does not exist. In this case, a notification is issued. |
| name | The name of an existing function, which may be optional and schema-qualified. |

| Parameter | Description |
|---|---|
| argmode | The mode of an argument. Valid values: IN, IN OUT or OUT. If this parameter is omitted, the default is IN. The DROP FUNCTION is not actually affected by the OUT arguments, since only the input arguments are required to determine the identity of the function. So it is sufficient to list only the IN and IN OUT arguments. The specification of argmode is not compatible with Oracle databases and applies only to PolarDB database compatible with Oracle. |
| argname | The name of an argument. The DROP FUNCTION is not actually affected by argument names, since only the argument data types are required to determine the identity of the function. The specification of argname is not compatible with Oracle databases and applies only to PolarDB database compatible with Oracle. |
| argtype | The data type of an argument of the function. The specification of argtype is not compatible with Oracle databases and applies only to PolarDB database compatible with Oracle. |
| CASCADE | Automatically drop objects that depend on the function (such as operators or triggers), and in turn all objects that depend on those objects. |
| RESTRICT | Refuses to drop the function if any objects depend on it. This is the default value. |

**Example**

The following command removes the emp_comp function.

```
DROP FUNCTION emp_comp(NUMBER, NUMBER);
```

## 12.40 DROP INDEX

Deletes an index.

**Syntax**

```
DROP INDEX name
```

**Description**

You can use the DROP INDEX command to drop an existing index from the database system. To run this command on an index, you must be a superuser or the owner of the index. If objects depend on the index, an error occurs, but the index is not dropped.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of an index to be deleted. The name can be schema-qualified. |

### Examples

Delete the name_idx index:

```
DROP INDEX name_idx;
```

## 12.41 DROP PACKAGE

Deletes a package.

### Syntax

```
DROP PACKAGE [ BODY ] name
```

### Description

You can use the DROP PACKAGE command to drop an existing package. To run this command on a package, you must be a superuser or the owner of the package. If you specify BODY, only the package body is deleted, and the package specification is not dropped. If you omit BODY, both the package specification and body are deleted.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of a package to be deleted. The name can be schema-qualified. |

### Examples

Delete the emp_admin package:

```
DROP PACKAGE emp_admin;
```

## 12.42 DROP PROCEDURE

Deletes a stored procedure.

### Syntax

```
DROP PROCEDURE [ IF EXISTS ] name
```

```
[ ([ [ argmode ] [ argname ] argtype ] [, ...]) ]
[ CASCADE | RESTRICT ]
```

### Description

You can use the DROP PROCEDURE command to delete an existing stored procedure. To run this command on a stored procedure, you must be a superuser or the owner of the procedure. If the stored procedure is overloaded, you must specify all input (IN and IN OUT ) argument data types to the procedure. This requirement is not compatible with Oracle databases. In Oracle, you can only specify procedure names. PolarDB-O allow overloading of stored procedure names, so the procedure signature that is given by the input argument data types is required in the DROP PROCEDURE command that is run on an overloaded stored procedure in PolarDB-O.

Usage of IFEXISTS, CASCADE, or RESTRICT is not compatible with Oracle databases, and can be used only by PolarDB-O.

### Parameters

| Parameter | Description |
|---|---|
| IF EXISTS | Specifies that the system does not report an error if the stored procedure does not exist. The server issues a notice in this case. |
| name | The name of an existing stored procedure. The name can be schema-qualified. |
| argmode | The modes of an argument. The argument modes include IN , IN OUT, and OUT. The default mode is IN. Note that DROP PROCEDURE is irrelevant to OUT argument, because only the input arguments are required to determine the identity of the stored procedure. Therefore, only the IN and INOUT arguments are listed. Specification of argmode is not compatible with Oracle databases and applies only to PolarDB-O. |
| argname | The name of an argument. Note that DROP PROCEDURE is irrelevant to argument names, because only the argument data types are required to determine the identity of the stored procedure. Specification of argname is not compatible with Oracle databases and applies only to PolarDB-O. |
| argtype | The data type of an argument of the stored procedure. Specificat ion of argtype is not compatible with Oracle databases and applies only to PolarDB-O. |
| CASCADE | Specifies that all objects that depend on the stored procedure and objects that depend on those objects are automatically dropped. |

| Parameter | Description |
|---|---|
| RESTRICT | Specifies that the stored procedure is not dropped if objects depend on it. This is the default behavior. |

### Examples

Delete the select_emp procedure:

```
DROP PROCEDURE select_emp;
```

## 12.43 DROP PROFILE

Drops a user-defined profile.

### Syntax

```
DROP PROFILE [IF EXISTS] profile_name [CASCADE | RESTRICT];
```

### Description

The IF EXISTS clause instructs the server not to report an error even if the specified profile does not exist. If the specified profile does not exist, the server issues a notice.

The optional CASCADE clause reassigns users that are associated with the profile to the default profile, and then drops the profile. The optional RESTRICT clause instructs the server not to drop the profile that is associated with a role. This is the default behavior.

### Parameters

| Parameter | Description |
|---|---|
| profile_name | The name of the profile to be dropped. |

### Examples

Drop a profile whose name is acctg_profile:

```
DROP PROFILE acctg_profile CASCADE;
```

In the following example, the roles were associated with the acctg_profile profile. The command re-associates the roles with the default profile and then drops the acctg_profile profile.

Drop a profile whose name is acctg_profile:

```
DROP PROFILE acctg_profile RESTRICT;
```

The RESTRICT clause in the command instructs the server not to drop acctg_profile if the profile is associated with certain roles.

## 12.44 DROP QUEUE

Drops an existing queue.

### Syntax

```
DROP QUEUE [IF EXISTS] name
```

### Description

You can use the DROP QUEUE command to drop an existing queue. To run this command, you must be a user that has the aq_administrator_role privilege.

**Note:**

PolarDB databases compatible with Oracle provide the syntax of the DROP QUEUE SQL command that is not provided by Oracle. You can use this syntax together with DBMS_AQADM.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the queue. The name can be schema-qualified. |
| IF EXISTS | The IF EXISTS clause instructs the server not to return an error even if the specified queue does not exist. If the specified queue does not exist, the server issues a notice. |

### Examples

Drop a queue whose name is work_order:

```
DROP QUEUE work_order;
```

## 12.45 DROP QUEUE TABLE

Drops a queue table.

### Syntax

```
DROP QUEUE TABLE [ IF EXISTS ] name [, ...] [CASCADE | RESTRICT]
```

### Description

You can use the DROP QUEUE TABLE command to drop a queue table. Only a user with the aq_administrator_role privilege can run this command.

**Note:**

PolarDB databases compatible with Oracle include extra syntax for the DROP QUEUE TABLE SQL command. The extra syntax is not offered by Oracle. You can use the syntax in association with DBMS_AQADM.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the queue table to be deleted. The name can be schema-qualified. |
| IFEXISTS | You can include the IF EXISTS clause to instruct the server not to return an error if the queue table does not exist. Instead, the server issues a notice. |
| CASCADE | You can include the CASCADE keyword to automatically delete the objects that depend on the queue table. |
| RESTRICT | You can include the RESTRICT keyword to instruct the server not to delete the queue table if other objects depend on it. This is the default behavior. |

### Examples

The following example deletes a queue table whose name is work_order_table and the objects that depend on the queue table:

```
DROP QUEUE TABLE work_order_table CASCADE;
```

## 12.46 DROP SYNONYM

Deletes a synonym.

### Syntax

```
DROP [PUBLIC] SYNONYM [schema.]syn_name
```

### Description

You can use the DROP SYNONYM command to delete existing synonyms. To run this command on a synonym, you must be the owner of the synonym and have the USAGE privileges on the schema in which the synonym resides.

### Parameters

| Parameter | Description |
|---|---|
| syn_name | syn_name is the name of the synonym. A synonym name must be unique within a schema. |
| schema | schema specifies the name of the schema where the synonym resides. |

Similar to other objects that can be schema-qualified, you may have two synonyms with the same name in your search path. To disambiguate the name of the synonym to be dropped, include a schema name. Unless a synonym is schema-qualified in the DROP SYNONYM command, PolarDB databases compatible with Oracle deletes the first instance of the synonym that is found in your search path.

You can optionally add the PUBLIC clause to drop a synonym that resides in the public schema. The DROP PUBLIC SYNONYM command is compatible with Oracle databases and drops a synonym that resides in the public schema:

```
DROP PUBLIC SYNONYM syn_name;
```

The following example drops the personnel synonym:

```
DROP SYNONYM personnel;
```

## 12.47 DROP SEQUENCE

Deletes a sequence.

### Syntax

```
DROP SEQUENCE name [, ...]
```

### Description

You can use the DROP SEQUENCE command to delete sequence number generators. To run this command on a sequence, you must be a superuser or the owner of the sequence.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of a sequence. The name can be schema-qualified. |

### Examples

Delete the serial sequence:

```
DROP SEQUENCE serial;
```

## 12.48 DROP TABLE

Deletes a table.

### Syntax

```
DROP TABLE name [CASCADE | RESTRICT | CASCADE CONSTRAINTS]
```

### Description

You can use the DROP TABLE command to delete tables from the database. Only the owner of a table can delete a table. To clear a table of rows without deleting the table, you can use the DELETE command. DROP TABLE always deletes indexes, rules, triggers, and constraints that exist for the target table.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of a package to be deleted. The name can be schema-qualified. |

You can include the RESTRICT keyword to specify that the server does not drop the table if other objects depend on it. If objects depend on the table, the DROP TABLE command reports an error. This is the default behavior.

You can include the CASCADE clause to drop the objects that depend on the table.

You can include the CASCADE CONSTRAINTS clause to specify that the PolarDB database compatible with Oracle drops the dependent constraints (excluding other object types) on the specified table.

### Examples

Drop a table named emp that has no dependencies:

```
DROP TABLE emp;
```

The results of a DROP TABLE command varies depending on whether the table has dependencies. Therefore, you can control the result by specifying a drop behavior. For example, you create two tables named orders and items, and the items table is dependent on the orders table:

```
CREATE TABLE orders
  (order_id int PRIMARY KEY, order_date date, ...) ;
CREATE TABLE items
  (order_id REFERENCES orders, quantity int, ...) ;
```

Depending on the drop behavior that you specify, the PolarDB database compatible with Oracle drops the orders table as follows:

- If you specify DROP TABLE orders RESTRICT, the PolarDB database compatible with Oracle reports an error.
- If you specify DROPTABLE orders CASCADE, the PolarDB database compatible with Oracle drops the orders table and the items table.

- If you specify DROPTABLE orders CASCADE CONSTRAINTS, the PolarDB database compatible with Oracle drops the orders table and deletes the foreign key specification from the items table, but does not drop the items table.

## 12.49 DROP TABLESPACE

Deletes a tablespace.

### Syntax

```
DROP TABLESPACE tablespacename
```

### Description

You can use the DROP TABLESPACE command to delete a tablespace from the system.

Only the owner of a table can drop a table. Before dropping a tablespace, you must empty all database objects in the tablespace. Objects in other databases may still reside in the tablespace even if no objects in the current database are using the tablespace.

### Parameters

| Parameter | Syntax |
|---|---|
| tablespacename | The name of a tablespace. |

### Examples

Delete the employee_space tablespace from the system:

```
DROP TABLESPACE employee_space;
```

## 12.50 DROP TRIGGER

Deletes a trigger.

### Syntax

```
DROP TRIGGER name
```

### Description

You can use the DROP TRIGGER command to delete a trigger from its associated table. Only a superuser or the owner of the table on which the trigger is defined can run this command.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of a trigger to be deleted. |

### Examples

Delete the `emp_sal_trig` trigger:

```
DROP TRIGGER emp_sal_trig;
```

## 12.51 DROP TYPE

Deletes a type definition.

### Syntax

```
DROP TYPE [ BODY ] name
```

### Description

You can use the `DROP TYPE` command to delete the type definition. To run this command on a type, you must be a superuser or the owner of the type.

The optional `BODY` qualifier applies only to object type definitions, not to collection types or composite types. If you specify `BODY`, only the object type body is deleted and the object type specification is not deleted. If you do not specify `BODY`, both the object type specification and body are deleted.

If other database objects are dependent on the specified type, the type is not deleted.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of a type definition to be deleted. |

### Examples

Drop the addr_obj_typ object type:

```
DROP TYPE addr_obj_typ;
```

Drop the nested table type named budget_tbl_typ:

```
DROP TYPE budget_tbl_typ;
```

## 12.52 DROP USER

Deletes a database user account.

### Syntax

```
DROP USER name [ CASCADE ]
```

### Description

You can use the DROP USER command to drop the specified user. To drop a superuser, you must be a superuser or have the CREATEROLE privilege.

You cannot delete the user that is still referenced in a database of the cluster. Otherwise, an error occurs. Before dropping a user, you must drop all the objects that belong to the user or reassign their ownership, and revoke the privileges granted by the user.

However, you do not need to delete role memberships involving the user. DROP USER automatically revokes the memberships of the target user in other roles and those of other roles in the target user. Other roles are not dropped or affected.

In addition, if all objects owned by the user belong to a schema that is owned by the user and has the same name as the user, you can specify a CASCADE option. In this case, only the superuser and the name user can issue the DROP USER name CASCADE command, and the schema and all objects in the schema are deleted.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the user to be deleted. |
| CASCADE | Specifies that the schema that is owned by the user and has the same name as the user is dropped when no dependencies on the user or the schema exist. All objects owned by the user in the schema are also dropped. |

### Examples

Drop a user that does not own objects and is not granted privileges on other objects:

```
DROP USER john;
```

Drop the john user that is not granted privileges on the objects, and do not own objects outside of the john schema:

```
DROP USER john CASCADE;
```

## 12.53 DROP VIEW

Deletes a view.

### Syntax

```
DROP VIEW name
```

### Description

You can use the DROP VIEW command to drop an existing view. To run this command on a view, you must be a superuser or the owner of the view. If the specified view has dependent objects, such as a view of the view, the specified view is not deleted.

The form of the DROP VIEW command that is compatible with Oracle does not support the CASCADE clause. To drop a view and its dependencies, use the PostgreSQL-compatible form of the DROP VIEW command. For more information, visit the PostgreSQL documentation at https://www.postgresql.org/docs/11/static/sql-dropview.html.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the view to be deleted. The name can be schema-qualified. |

### Examples

Delete the dept_30 view:

```
DROP VIEW dept_30;
```

## 12.54 EXEC

### Syntax

```
EXEC function_name ['('[argument_list]')']
```

### Description

EXECUTE.

### Parameters

| Parameter | Description |
|---|---|
| procedure_name | procedure_name is the function name. The name can be schema-qualified. |
| argument_list | argument_list specifies a comma-separated list of arguments that are required by the function. Note that each member of argument_list corresponds to a formal argument that is expected by the function. Each formal argument can be an IN parameter, an OUT parameter, or an INOUT parameter. |

### Examples

The EXEC statement has multiple forms. You can use a form depending on the arguments that are required by the following functions:

```
EXEC update_balance;
EXEC update_balance();
EXEC update_balance(1,2,3);
```

## 12.55 GRANT

Defines access privileges.

### Syntax

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES }
  [,...] | ALL [ PRIVILEGES ] }
  ON tablename
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]
```

```
GRANT { { INSERT | UPDATE | REFERENCES } (column [, ...]) }
  [, ...]
  ON tablename
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT { SELECT | ALL [ PRIVILEGES ] }
  ON sequencename
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON FUNCTION progname
    ( [ [ argmode ] [ argname ] argtype ] [, ...] )
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON PROCEDURE progname
    [ ( [ [ argmode ] [ argname ] argtype ] [, ...] ) ]
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON PACKAGE packagename
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT role [, ...]
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH ADMIN OPTION ]

GRANT { CONNECT | RESOURCE | DBA } [, ...]
  TO { username | groupname } [, ...]
  [ WITH ADMIN OPTION ]

GRANT CREATE [ PUBLIC ] DATABASE LINK
  TO { username | groupname }

GRANT DROP PUBLIC DATABASE LINK
  TO { username | groupname }

GRANT EXEMPT ACCESS POLICY
  TO { username | groupname }
```

**Description**

The GRANT command has three basic variants: the one that grants privileges on a database object (table, view, sequence, or program), the one that grants membership in a role, and the one that grants system privileges. These variants are similar in many ways, but they are different. For information about each variant, see the specific topic.

In PolarDB databases compatible with Oracle, the concept of users and groups is unified into a single type of entity that is called a role. A user is a role that has the LOGIN attribute. You can use the role to create a session and connect to an application. A group is a role that does not have the LOGIN attribute. You cannot use the role to create a session or connect to an application.

A role can be a member of one or more other roles. Therefore, the traditional concept of user membership in groups is still valid. However, users can belong to users and groups due to the generalization of users and groups. This forms a general multi-level hierarchy of roles. Whether a grantee is a user or a group is not distinguished in the GRANT command, because usernames and group names share the same namespace.

## 12.56 GRANT on database objects

This variant of the GRANT command gives specific privileges on a database object to a role. These privileges are added to the privileges that are already granted to the role.

The PUBLIC keyword indicates that the privileges are granted to all roles, including those that you create later. PUBLIC is an implicitly defined group that includes all roles. A role has the privileges that are granted directly to the roles, the privileges that are granted to another role of which the role is a member, and the privileges that are granted to PUBLIC.

If you specify WITHGRANT OPTION, the recipient of the privileges can grant it to other roles. If you do not specify these keywords, the recipient cannot grant privileges. Grant options cannot be granted to PUBLIC.

You do not need to grant privileges to the owner of an object (usually the user who created the object), because the owner has all privileges by default. The owners can choose to revoke some of their own privileges for safety. Grantable privileges do not include the privileges to drop an object or alter its definition. The privileges that cannot be granted are inherent in the owner and cannot be granted or revoked. In addition, the owner implicitly has all grant options for the object.

Depending on the type of object, certain privileges can be granted to PUBLIC. The default privileges are non-public access for tables, and EXECUTE privileges for functions, procedures, and packages. The object owner can revoke these privileges. For maximum security, you can issue the REVOKE command in the same transaction that creates the object. This way, other users cannot use the object in any window.

The following table describes the possible privileges.

| Privilege | Description |
|---|---|
| SELECT | Allows to SELECT from columns of the specified table, view, or sequence. For sequences, this privilege also allows you to use the currval function. |
| INSERT | Allows to INSERT a new row into the specified table. |
| UPDATE | Allows to UPDATE a column of the specified table. SELECT ... FOR UPDATE also requires this privilege in addition to the SELECT privilege. |
| DELETE | Allows to DELETE a row from the specified table. |
| REFERENCES | Allows to create foreign key constraints. If you want to create foreign key constraints, you must have this privilege on both the referencing and referenced tables. |
| EXECUTE | Allows to use the specified package, stored procedure, or function. This privilege on a package allows you to use all public stored procedures, public functions, public variables, records, cursors, and other public objects and object types in the package. This is the only type of privilege that is applicable to functions, stored procedures, and packages.<br>The syntax for granting the EXECUTE privilege in PolarDB databases compatible with Oracle is not fully compatible with Oracle databases. PolarDB databases compatible with Oracle requires qualification of the program name by one of the following keywords: FUNCTION, PROCEDURE, and PACKAGE. However, in Oracle databases, these keywords must be omitted. For functions, PolarDB databases compatible with Oracle require all input (IN and IN OUT) argument data types after the function name. If no function arguments exist, the function name must be followed by an empty pair of parenthesis. For stored procedures, if a procedure has one or more input arguments, you must specify all input argument data types. In Oracle, function and stored procedure signatures must be omitted. This is because all programs share the same namespace in Oracle. However, the functions, stored procedures, and packages have their own individual namespaces in PolarDB databases compatible with Oracle. This allows program name overloading to a certain extent. |
| ALL PRIVILEGES | Grants all available privileges at once. |

For more information about the privileges that are required by other commands, see the topic of the corresponding command.

## 12.57 INSERT

Creates rows in a table.

### Syntax

```
INSERT INTO table[@dblink ] [ ( column [, ...] ) ]
  { VALUES ( { expression | DEFAULT } [, ...] )
    [ RETURNING return_expression [, ...]
        { INTO { record | variable [, ...] }
        | BULK COLLECT INTO collection [, ...] } ]
  | query }
```

### Description

You can run the INSERT command to insert new rows into a table. You can insert one or multiple rows as a result of a query.

You can list the columns in the order that you desire. Each column that is not in the target list will be inserted with a default value, either its declared default value or null.

If the expression for a column does not use the correct data type, automatic type conversion is attempted.

If the INSERT command is used within an SPL program and the VALUES clause is specified, you can specify RETURNINGINTO { record | variable [, ...] } clause.

If using INSERT command within an SPL program, you can specify the RETURNING BULK COLLECT INTO collection [, ...] clause. If you specify multiple collection as the target of the BULK COLLECT INTO clause, each collection must consist of a single scalar field. collection cannot be a record. For each inserted row, the evaluated value return_expression is an element in collection that starts from the first element. Existing rows in collection are deleted. If the result set is empty, collection is also empty.

You must have the INSERT privilege on a table so that you can insert into it. If you use the query clause to insert rows from a query, you must also have the SELECT privilege on the table that is used in the query.

### Parameters

| Parameter | Description |
|---|---|
| table | The name of an existing table. The name can be schema-qualified. |

| Parameter | Description |
|---|---|
| dblink | The name of the database link that is used to identify a remote database. For more information about database links, see the CREATE DATABASE LINK command. |
| column | The name of a column in table. |
| expression | An expression or value to assign to column. |
| DEFAULT | The default value of the column. |
| query | A query (the SELECT statement) that provides the rows to be inserted. For more information, see the SELECT command. |
| return_expression | An expression that can include one or more columns from table . If a column name from table is specified in return_expression, the value substituted for the column when return_expression is evaluated is determined as follows:<br>• If you assign a value in the INSERT command to the specified column in return_expression, the assigned value is used to evaluate return_expression.<br>• If you do not assign a value in the INSERT command to the specified column in return_expression and no default value is provided for the column definition, null is used to evaluate return_expression.<br>• If you do not assign a value in the INSERT command to the specified column in return_expression and a default value is provided for the column definition, the default value is used to evaluate return_expression. |
| record | A record to whose field you want to assign the evaluation result of return_expression. For example, the first return_expression is assigned to the first field in record, and the second return_expression is assigned to the second field in record. The number of fields in record must match the number of expressions, and the fields must be type-compatible with corresponding expressions. |
| variable | A variable to which you want to assign the evaluation result of return_expression. If you specify multiple return_expression and variable, the first return_expression is assigned to the first variable, and the second return_expression is assigned to the second variable. The number of the specified variables that follow the INTO keyword must match the number of expressions that follow the RETURNING keyword, and the variables must be type-compatible with corresponding expressions. |

| Parameter | Description |
|---|---|
| collection | A collection in which an element is created from the evaluated return_expression. You can specify a collection of a single field or a collection of a record type. You can also specify multiple collections where each collection consists of a single field. The number of return expressions must match in number and order of fields in all specified collections. Each return_expression must be type-compatible with the corresponding collection field. |

**Examples**

Insert a single row into the emp table:

```
INSERT INTO emp VALUES (8021,'JOHN','SALESMAN',7698,'22-FEB-07',1250,500,30);
```

In this second example, the column named comm is omitted. Therefore, it has the default value of null:

```
INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, deptno)
    VALUES (8022,'PETERS','CLERK',7698,'03-DEC-06',950,30);
```

The third example uses the DEFAULT clause for the hiredate and comm columns rather than specifying a value:

```
INSERT INTO emp VALUES (8023,'FORD','ANALYST',7566,NULL,3000,NULL,20);
```

This example creates a table for the department names, and then inserts into the table. The department names are obtained from the dname column of the dept table:

```
CREATE TABLE deptnames (
    deptname        VARCHAR2(14)
);
```

```
INSERT INTO deptnames SELECT dname FROM dept;
```

## 12.58 LOCK

Locks a table.

### Syntax

```
LOCK TABLE name [, ...] IN lockmode MODE [ NOWAIT ]
```

Where lockmode is one of the following items:

```
ROW SHARE | ROW EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE
```

### Description

You can use the LOCK TABLE command to acquire a table-level lock. If conflicting locks exist, the command waits until all of the conflicting locks are released and locks the table by default. If you specify NOWAIT, the LOCK TABLE command does not wait to acquire the desired lock. If the lock cannot be immediately acquired, the command ends and an error occurs. After you obtain a lock, the lock is held until the current transaction ends. The UNLOCK TABLE command is unavailable. Tables remain lock until transactions come to an end.

When acquiring automatic locks for the commands that reference tables, PolarDB databases compatible with Oracle use the least restrictive lock mode possible. LOCK TABLE is provided for cases when you need more restrictive locking. For example, an application runs a transaction at the isolation level of read committed and the stability of data in a table needs to be ensured during the transaction. To achieve this, you can lock the table in the SHARE mode before querying. This prevents concurrent data changes and ensures a stable view of committed data for subsequent table reads because the SHARE lock mode conflicts with the ROW EXCLUSIVE lock acquired by writers. Your LOCK TABLE name IN SHARE MODE statement waits until concurrent holders of ROW EXCLUSIVE locks commit or roll back. Therefore, after you obtain the lock, no uncommitted writes exist. In addition, none can perform operations on the table until you release the lock.

To achieve a similar effect when running a transaction at the serializable isolation level, you must run the LOCK TABLE statement before running data modification statement. A serializable transaction view of data is frozen after its first data modification statement begins. A later LOCK TABLE will still prevent concurrent writes, but the values that the transaction reads may differ from the latest committed values.

If a serializable transaction is going to change data in the table, it needs to lock the table in the SHARE ROW EXCLUSIVE mode instead of SHARE mode.

This ensures that only one transaction of this type runs at a time. Otherwise, a deadlock may occur. Two transactions may lock the table in the SHARE mode at the same time, and then neither of them can acquire the lock in the ROWEXCLUSIVE mode to perform updates. Note that locks never conflict within a transaction, so a transaction can lock a table in the ROW EXCLUSIVE mode when it holds the SHARE mode. However, a transaction cannot acquire the ROW EXCLUSIVE lock if another transaction holds the SHARE lock. To avoid deadlocks, make sure that all transactions acquire locks on the same objects in the same order. If a single object allows multiple lock modes, transactions must acquire the most restrictive mode first.

**Parameters**

| Parameter | Parameter |
|---|---|
| name | The name of the table to be locked. The name can be schema-qualified.<br>The LOCKTABLE a, b command is equivalent to LOCK TABLE a; LOCK TABLE b. The tables are locked one by one in the order specified in the LOCK TABLE command. |
| lockmode | The lock mode that specifies the locks with which this lock conflicts.<br>If no lock mode is specified, the server uses the most restrictive mode, ACCESS EXCLUSIVE. ACCESS EXCLUSIVE is not compatible with Oracle databases. In PolarDB databases compatible with Oracle, this mode ensures that no other transaction can access the locked table in any manner. |
| NOWAIT | Specifies that the LOCKTABLE command does not wait for conflicting locks to be released. If you cannot immediately acquire the specified lock, the transaction ends. |

### Notes

All forms of `LOCK` require `UPDATE` and/or `DELETE` privileges.

`LOCK TABLE` is useful only inside a transaction block because the lock is dropped when the transaction ends. A `LOCK TABLE` command that is used outside a transaction block forms a self-contained transaction, so the lock will be dropped when you obtain it.

`LOCK TABLE` only deals with table-level locks, so the mode names containing `ROW` are all misnomers. These mode names are read as indicating that the user intend to acquire row-level locks within the locked table. In addition, a `ROW EXCLUSIVE` lock is a sharable table lock. All the lock modes have identical semantics when `LOCK TABLE` is concerned, and are different only in the rules for checking conflicts.

## 12.59 REVOKE

Revokes access privileges.

### Syntax

```
REVOKE { { SELECT | INSERT | UPDATE | DELETE | REFERENCES }
  [,...] | ALL [ PRIVILEGES ] }
  ON tablename
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { SELECT | ALL [ PRIVILEGES ] }
  ON sequencename
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] }
  ON FUNCTION progname
    ( [ [ argmode ] [ argname ] argtype ] [, ...] )
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] }
  ON PROCEDURE progname
    [ ( [ [ argmode ] [ argname ] argtype ] [, ...] ) ]
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] }
  ON PACKAGE packagename
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE role [, ...] FROM { username | groupname | PUBLIC }
  [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { CONNECT | RESOURCE | DBA } [, ...]
  FROM { username | groupname } [, ...]
```

```
REVOKE CREATE [ PUBLIC ] DATABASE LINK
  FROM { username | groupname }

REVOKE DROP PUBLIC DATABASE LINK
  FROM { username | groupname }

REVOKE EXEMPT ACCESS POLICY
  FROM { username | groupname }
```

**Description**

You can use the REVOKE command to revoke privileges that have been granted to one or more roles. The PUBLIC keyword refers to the implicitly defined group of all roles.

For more information about the types of privileges, see the description of GRANT command.

Note that a role has the privileges that are granted directly to the role, the privileges that are granted to another role of which the role is a member, and the privileges that are granted to PUBLIC. For example, if you revoke the SELECT privilege from PUBLIC, it does not indicate that all roles have lost the SELECT privilege on the object. The roles that have the SELECT privilege granted directly and their member roles still have the SELECT privilege.

If the privilege is granted with the grant option, both privilege and the grant option for the privilege are revoked.

If a user has a privilege with the grant option and grants the privilege to other users, the privilege held by other users is called dependent privileges. If you want to revoke the privilege or grant option from the first user and dependent privileges exist, the dependent privileges are also revoked when CASCADE is specified. Otherwise, the revoke action failed. This recursive revocation only affects privileges that are granted by a chain of users that starts from the user who runs this REVOKE command. The affected users may keep the privilege if it is also granted by other users.

**Notice:**

The CASCADE option is not compatible with Oracle databases. By default, Oracle cascades dependent privileges. However, PolarDB databases compatible with Oracle requires the explicit CASCADE keyword. Otherwise, the REVOKE command will fail.

When revoking membership in a role, use GRANT OPTION instead of ADMIN OPTION, but the behavior is similar.

### Notes

A user can only revoke privileges that are granted by the user. For example, if User A grants a privilege with the grant option to User B and User B grants the privilege to User C, User A cannot revoke the privilege directly from User C. Instead, User A can revoke the grant option from User B and use the CASCADE option so that the privilege is revoked from User C. For another example, if both User A and User B grant the same privilege to User C, User A can revoke the privilege granted by User A but not by User B. Therefore, after User A revokes the privilege, User C still has the privilege that is granted by User B.

If a user has no privileges on an object that belongs to another user and the non-owner user attempts to revoke privileges on the object by running the REVOKE command, the command fails. If a privilege can be revoked, the command proceeds but revokes only the privileges for which the user has grant options. If no grant options are held, the REVOKE ALL PRIVILEGES forms issue a warning message. Other forms also issue a warning message if the grant option for a privilege specified in the command is not held. This mechanism applies to the object owner. However, no warning messages are issued for the object owner because the owner holds all grant options.

In addition to the object owner, REVOKE can also be done by a member of the role that owns the object or a member of a role that holds the WITH GRANT OPTION privilege on the object. In this case, the command result is same as the result of the command that is issued by the containing role that owns the object or holds the WITH GRANT OPTION privilege. For example, if the t1 table is owned by the g1 role of which the u1 role is a member, u1 can revoke privileges on t1 that are granted by g1. Both the grants made by the u1 role and other members of the g1 role are revoked.

If the role that runs the REVOKE command holds privileges that are granted through multiple role chains, you cannot specify the role chain from which the privilege is revoked. In such cases, use SET ROLE to assume the role as which you want to run the REVOKE command. Otherwise, the privileges that are revoked are not the ones you intended, or are not revoked at all.

**Note:**

The ALTER ROLE command of PolarDB databases compatible with Oracle also supports syntax that revokes the system privileges required to create a public or private database link, or the exemptions from fine-grained access control policies (DBMS_RLS). The ALTER

ROLE command is functionally equivalent to the respective REVOKE command, and is compatible with Oracle databases.

### Examples

Revoke the INSERT privilege on the emp table from the PUBLIC group:

```
REVOKE INSERT ON emp FROM PUBLIC;
```

Revoke all privileges on the salesemp view from the user named mary:

```
REVOKE ALL PRIVILEGES ON salesemp FROM mary;
```

Note that all privileges granted by the user that runs the command are revoked.

Revoke membership in the admins role from the user named joe:

```
REVOKE admins FROM joe;
```

Revoke the CONNECT privilege from the user named joe:

```
REVOKE CONNECT FROM joe;
```

Revoke the CREATE DATABASE LINK privilege from the user named joe:

```
REVOKE CREATE DATABASE LINK FROM joe;
```

Revoke the EXEMPT ACCESS POLICY privilege from the user named joe:

```
REVOKE EXEMPT ACCESS POLICY FROM joe;
```

## 12.60 ROLLBACK

Rolls back the current transaction.

### Syntax

```
ROLLBACK [ WORK ]
```

### Description

You can use the ROLLBACK command to roll back the current transaction and discard all the updates made by the transaction.

### Parameters

| Parameter | Description |
|---|---|
| WORK | An optional keyword, which has no effect. |

### Notes

You can use the COMMIT command to end a transaction.

If you run ROLLBACK at a time point which is not in a transaction, no changes are made.

**Notice:**

If an Oracle-style SPL procedure exists on the runtime stack, an error occurs when you run a ROLLBACK command in a PL/pgSQL procedure.

### Examples

Roll back all changes:

```
ROLLBACK;
```

## 12.61 ROLLBACK TO SAVEPOINT

Rolls back to a savepoint.

### Syntax

```
ROLLBACK [ WORK ] TO [ SAVEPOINT ] savepoint_name
```

### Description

You can use the ROLLBACK TO SAVEPOINT command to roll back all commands that are run after the specified savepoint is created. The savepoint remains valid and can be rolled back to again later if needed.

ROLLBACK TO SAVEPOINT implicitly deletes all savepoints that are created after the specified savepoint.

### Parameters

| Parameter | Description |
|---|---|
| savepoint_name | The savepoint to which to roll back. |

### Notes

An error occurs if you specify a savepoint name that does not exist.

SPL programs do not support ROLLBACK TO SAVEPOINT.

### Examples

Undo the effects of commands that are run after the depts savepoint:

```
\set AUTOCOMMIT off
INSERT INTO dept VALUES (50, 'HR', 'NEW YORK');
SAVEPOINT depts;
INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);
INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);
ROLLBACK TO SAVEPOINT depts;
```

## 12.62 SAVEPOINT

Defines a new savepoint in the current transaction.

### Syntax

```
SAVEPOINT savepoint_name
```

### Description

The SAVEPOINT command creates a new savepoint in the current transaction.

A savepoint is a special mark in a transaction. It allows all commands that are executed after it is created to be rolled back. If the commands are rolled back, the transaction state is restored to what it was at the time of the savepoint.

### Parameters

| Parameter | Description |
|---|---|
| savepoint_name | The name that you want to specify for the savepoint. |

### Description

You can run the ROLLBACK TO SAVEPOINT command to roll back to a savepoint.

Savepoints can be created only in a transaction block. You can define multiple savepoints in a transaction.

If another savepoint with the same name as a previous savepoint is created, the previous savepoint is retained. However, only the more recent savepoint is used during a rollback.

The SAVEPOINT command is not supported within SPL programs.

### Examples

The following example shows how to create a savepoint and then undo all commands that are executed after the savepoint is created.

```
\set AUTOCOMMIT off
INSERT INTO dept VALUES (50, 'HR', 'NEW YORK');
SAVEPOINT depts;
INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);
INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);
SAVEPOINT emps;
INSERT INTO jobhist VALUES (9001,'17-SEP-07',NULL,'CLERK',800,NULL,50,'New Hire');
INSERT INTO jobhist VALUES (9002,'20-SEP-07',NULL,'CLERK',700,NULL,50,'New Hire');
ROLLBACK TO depts;
COMMIT;
```

The preceding transaction submits a row to the dept table, but the contents inserted into the emp and joblist tables are rolled back.

# 12.63 SELECT

## 12.63.1 SELECT

Retrieves rows from a table or view.

### Syntax

```
SELECT [ optimizer_hint ] [ ALL | DISTINCT ]
  * | expression [ AS output_name ] [, ...]
  FROM from_item [, ...]
  [ WHERE condition ]
  [ [ START WITH start_expression ]
      CONNECT BY { PRIOR parent_expr = child_expr |
 child_expr = PRIOR parent_expr }
    [ ORDER SIBLINGS BY expression [ ASC | DESC ] [, ...] ] ]
  [ GROUP BY { expression | ROLLUP ( expr_list ) |
      CUBE ( expr_list ) | GROUPING SETS ( expr_list ) } [, ...]
      [ LEVEL ] ]
  [ HAVING condition [, ...] ]
  [ { UNION [ ALL ] | INTERSECT | MINUS } select ]
  [ ORDER BY expression [ ASC | DESC ] [, ...] ]
  [ FOR UPDATE [WAIT n|NOWAIT|SKIP LOCKED]]
```

The following options for from_item are available:

```
 table_name[@dblink ] [ alias ]
  ( select ) alias
 from_item [ NATURAL ] join_type from_item
```

```
  [ ON join_condition | USING ( join_column [, ...] ) ]
```

### Description

You can use the SELECT statement to retrieve rows from one or more tables. The general processing of SELECT is described as follows:

- All elements in the FROM list are computed. Each element in the FROM list is a real or virtual table. If you specify more than one element in the FROM list, the specified elements are cross-joined. For more information, see the FROM clause topic.
- If you specify the WHERE clause, all rows that do not satisfy the condition are eliminated from the output. For more information, see the WHERE clause topic.
- If you specify the GROUP BY clause, the output is divided into groups of rows that match on one or more values. If you specify the HAVING clause, groups that do not satisfy the specified condition are eliminated from the output. For more information, see the GROUP BY clause and HAVING clause topics.
- You can use the UNION, INTERSECT, and MINUS operators to combine the output of more than one SELECT statement to form a single result set. The UNION operator returns all rows that are in one or both of the result sets. The INTERSECT operator returns all rows that are in both of the result sets. The MINUS operator returns the rows that are in the first result set but not in the second result set. In all the preceding three cases, duplicate rows are eliminated. If you specify ALL in the UNION operator, duplicate rows are not eliminated. For more information, see the UNION clause, INTERSECT clause, and MINUS clause topics.
- The actual output rows are computed using the SELECT output expressions for each selected row. For more information, see the SELECT list topic.
- The CONNECT BY clause is used to select data that has a hierarchical relationship. This type of data has a parent-child relationship between rows. For more information, see the CONNECT BY clause topic.
- If you specify the ORDER BY clause, the returned rows are sorted in the specified order. If you do not specify the ORDER BY clause, the rows are returned in whatever order the system finds fastest to produce. For more information, see the ORDER BY clause topic.
- DISTINCT eliminates duplicate rows from the result. ALL returns all candidate rows, including duplicate rows. The default value is ALL. For more information, see the DISTINCT clause topic.
- The FOR UPDATE clause causes the SELECT statement to lock the selected rows against concurrent updates. For more information, see the FOR UPDATE clause topic.

You must have the SELECT privilege on a table to read its values. To use the FOR UPDATE statement, you must have the UPDATE privilege.

### Parameters

| Parameter | Description |
|---|---|
| optimizer_hint | Comment-embedded hints to the optimizer . This parameter is used to select an execution plan. |

## 12.63.2 FROM clause

The FROM clause specifies one or more source tables for a SELECT statement.

### Syntax

```
FROM source [, ...]
```

The following table describes the available parameters for source.

| Parameter | Description |
|---|---|
| table_name[@dblink ] | The name of an existing table or view. The name can be schema-qualified. dblink is the name of a database link that identifies a remote database. For more information about database links, see the CREATE DATABASE LINK command topic. |
| alias | A substitute name for the FROM item that contains the alias. An alias is used for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). If you specify an alias for a table or function, the alias hides the actual name of the table or function. For example, if FROM foo AS f is specified, the remainder of the SELECT statement must refer to the FROM item as f rather than foo. |
| select | You can nest a SELECT statement in the FROM clause. This creates a derived table for the duration of the SELECT statement. You must enclose the nested SELECT statement in parentheses and specify an alias for it. |

| Parameter | Description |
|---|---|
| join_type | The following join types are available:<br>- [ INNNER ] JOIN<br>- LEFT [ OUTER ] JOIN<br>- RIGHT [ OUTER ] JOIN<br>- FULL [ OUTER ] JOIN<br>- CROSS JOIN<br>For the INNER and OUTER join types, a join condition must be specified. In other words, the join condition is one of NATURAL, ON join_condition, or USING (join_column [, ...] ). The following paragraphs describe the join types. For CROSS JOIN, none of these clauses appear.<br>A JOIN clause combines two FROM items. You can use parentheses to determine the order of nesting. In the absence of parentheses, JOIN clauses nest from left to right. The JOIN clause binds tighter than the commas separating FROM items.<br>CROSS JOIN and INNER JOIN produce a simple Cartesian product. The result is the same as that of listing the two tables at the top level of FROM, but is restricted by the join condition. CROSS JOIN is equivalent to INNER JOIN ON (TRUE). No rows are removed by qualification. The listed join types are for notational convenience. You can use the FROM and WHERE clauses to perform all operations that you can perform by using join types.<br>LEFT OUTER JOIN returns all rows in the qualified Cartesian product. The qualified Cartesian product contain all combined rows that pass the join condition. LEFT OUTER JOIN also returns the left-side rows that do not have a matching right-side row. Each left-side row that does not have a matching right-side row is extended to the full width of the joined table by inserting null values for the right-side columns. Note that only the condition of the JOIN clause is considered when whether rows have matches is decided. Then, outer conditions are applied.<br>RIGHT OUTER JOIN returns all the matching rows and the right-side rows that do not have a matching left-side row. Each right-side row is extended with null values on the left. This is a notational convenience. You can convert it to a LEFT OUTER JOIN by switching the left and right inputs. |

| Parameter | Description |
|---|---|
| ON join_condition | join_condition is an expression resulting in a value of the BOOLEAN type (similar to a WHERE clause) that specifies which rows in a join are considered to match. |
| USING (join_column [, ...] ) | A clause of the USING (a, b, ... ) form is short for ON left_table.a = right_table.a AND left_table.b = right_table.b ... In addition, USING indicates that only one of each pair of equivalent columns is included in the join output. |
| NATURAL | NATURAL is short for a USING list that includes all columns in the two tables that have the same names. |

If multiple sources are specified, the result is the Cartesian product (cross join) of all the sources. In most cases, qualification conditions are added to restrict the returned rows to a small subset of the Cartesian product.

### Examples

The following example selects all of the entries from the dept table:

```
SELECT * FROM dept;
deptno | dname      | loc
-------+------------+-----------
    10 | ACCOUNTING | NEW YORK
    20 | RESEARCH   | DALLAS
    30 | SALES      | CHICAGO
    40 | OPERATIONS | BOSTON
 (4 rows)
```

## 12.63.3 WHERE clause

### Syntax

The syntax of the optional WHERE clause is as follows:

```
WHERE condition
```

condition is an expression whose result is of the BOOLEAN type. Rows that do not satisfy this condition are eliminated from the output. A row satisfies the condition if it returns TRUE when the actual row values are substituted for variable references.

### Examples

The following example joins the contents of the emp and dept tables. In the WHERE clause, the value of the deptno column in the emp table is equal to the value of the deptno column in the deptno table.

```
SELECT d.deptno, d.dname, e.empno, e.ename, e.mgr, e.hiredate
    FROM emp e, dept d
    WHERE d.deptno = e.deptno;

 deptno |   dname    | empno | ename  | mgr  |      hiredate
--------+------------+-------+--------+------+--------------------
     10 | ACCOUNTING |  7934 | MILLER | 7782 | 23-JAN-82 00:00:00
     10 | ACCOUNTING |  7782 | CLARK  | 7839 | 09-JUN-81 00:00:00
     10 | ACCOUNTING |  7839 | KING   |      | 17-NOV-81 00:00:00
     20 | RESEARCH   |  7788 | SCOTT  | 7566 | 19-APR-87 00:00:00
     20 | RESEARCH   |  7566 | JONES  | 7839 | 02-APR-81 00:00:00
     20 | RESEARCH   |  7369 | SMITH  | 7902 | 17-DEC-80 00:00:00
     20 | RESEARCH   |  7876 | ADAMS  | 7788 | 23-MAY-87 00:00:00
     20 | RESEARCH   |  7902 | FORD   | 7566 | 03-DEC-81 00:00:00
     30 | SALES      |  7521 | WARD   | 7698 | 22-FEB-81 00:00:00
     30 | SALES      |  7844 | TURNER | 7698 | 08-SEP-81 00:00:00
     30 | SALES      |  7499 | ALLEN  | 7698 | 20-FEB-81 00:00:00
     30 | SALES      |  7698 | BLAKE  | 7839 | 01-MAY-81 00:00:00
     30 | SALES      |  7654 | MARTIN | 7698 | 28-SEP-81 00:00:00
     30 | SALES      |  7900 | JAMES  | 7698 | 03-DEC-81 00:00:00
(14 rows)
```

## 12.63.4 GROUP BY clause

### Syntax

The syntax of the optional `GROUP BY` clause is as follows:

```
GROUP BY { expression | ROLLUP ( expr_list ) |
  CUBE ( expr_list ) | GROUPING SETS ( expr_list ) } [, ...]
```

### Description

The `GROUP BY` clause condenses all selected rows that share the same values for the grouped expressions into a single row. `expression` can be an input column name, or the name or ordinal number of an output column that is specified in the `SELECT` list. It can also be an expression formed from the values of input columns. In case of ambiguity, a `GROUP BY` name is interpreted as the name of an input column rather than an output column.

`ROLLUP`, `CUBE`, and `GROUPING SETS` are extensions to the `GROUP BY` clause. These extensions are used to support multidimensional analysis.

If aggregate functions are used, the aggregate functions are computed across all rows in each group. This produces a separate value for each group. If no `GROUP BY` clause is specified, an aggregate function produces a single value computed across all the selected rows. If the GROUP BY clause is used, the SELECT list expressions cannot refer to ungrouped columns except within aggregate functions. This is because more than one value may be returned for an ungrouped column.

**Examples**

The following example computes the sum of the sal column in the emp table and groups the results by department number.

```
SELECT deptno, SUM(sal) AS total
    FROM emp
    GROUP BY deptno;

 deptno |  total
--------+----------
     10 |  8750.00
     20 | 10875.00
     30 |  9400.00
(3 rows)
```

## 12.63.5 HAVING clause

**Syntax**

The syntax of the optional HAVING clause is as follows:

```
HAVING condition
```

condition is the same as that specified for the WHERE clause.

**Description**

The HAVING clause eliminates group rows that do not satisfy the specified condition. The HAVING clause is different from the WHERE clause. The WHERE clause filters individual rows before the application of GROUP BY. The HAVING clause filters group rows created by GROUP BY. Each column referenced in a condition must explicitly reference a grouping column unless the column is referenced in an aggregate function.

**Examples**

To sum up the sal column for all employees, group the results by department number and show group totals that are less than 10,000.

```
SELECT deptno, SUM(sal) AS total
    FROM emp
    GROUP BY deptno
    HAVING SUM(sal) < 10000;

 deptno |  total
--------+---------
     10 | 8750.00
     30 | 9400.00
(2 rows)
```

## 12.63.6 SELECT list

The `SELECT` list between the `SELECT` and `FROM` keywords specifies expressions that form the output rows of the `SELECT` statement. The expressions can refer to columns computed in the `FROM` clause. You can specify another name for an output column by using the `AS output_name` clause. This name is used to label the column to be displayed. It can also be used to refer to the column value in the `ORDER BY` and `GROUP BY` clauses instead of the `WHERE` or `HAVING` clause. In this case, you must write out the expression.

You can enter an asterisk (*) instead of an expression in the output list to indicate all columns of the selected rows.

### Examples

The `SELECT` list in the following example specifies that the result set includes the empno column, the ename column, the mgr column, and the hiredate column.

```
SELECT empno, ename, mgr, hiredate FROM emp;

 empno | ename  | mgr  |      hiredate
-------+--------+------+--------------------
  7934 | MILLER | 7782 | 23-JAN-82 00:00:00
  7782 | CLARK  | 7839 | 09-JUN-81 00:00:00
  7839 | KING   |      | 17-NOV-81 00:00:00
  7788 | SCOTT  | 7566 | 19-APR-87 00:00:00
  7566 | JONES  | 7839 | 02-APR-81 00:00:00
  7369 | SMITH  | 7902 | 17-DEC-80 00:00:00
  7876 | ADAMS  | 7788 | 23-MAY-87 00:00:00
  7902 | FORD   | 7566 | 03-DEC-81 00:00:00
  7521 | WARD   | 7698 | 22-FEB-81 00:00:00
  7844 | TURNER | 7698 | 08-SEP-81 00:00:00
  7499 | ALLEN  | 7698 | 20-FEB-81 00:00:00
  7698 | BLAKE  | 7839 | 01-MAY-81 00:00:00
  7654 | MARTIN | 7698 | 28-SEP-81 00:00:00
  7900 | JAMES  | 7698 | 03-DEC-81 00:00:00
```

```
(14 rows)
```

## 12.63.7 UNION clause

### Syntax

The syntax of the UNION clause is as follows:

```
select_statement UNION [ ALL ] select_statement
```

### Description

select_statement is a SELECT statement that does not contain an ORDER BY or FOR UPDATE clause. You can enclose the ORDER BY clause in parentheses to attach it to a sub-expression. Without parentheses, these clauses are applied to the result of the UNION clause, not to the expression on the right side.

The UNION operator computes the set union of the rows returned by the involved SELECT statements. If a row is included in at least one of two result sets, the row is in the set union of the two result sets. The two SELECT statements that represent the direct operands of the UNION clause must produce the same number of columns. The corresponding columns must be of compatible data types.

The result of the UNION clause contains duplicate rows only if the ALL option is specified. The ALL option prevents elimination of duplicate rows.

Unless otherwise specified in parentheses, multiple UNION operators in the same SELECT statement are evaluated from left to right.

The FOR UPDATE clause may not be specified either for a UNION result or for an input of a UNION clause.

## 12.63.8 INTERSECT clause

### Syntax

The syntax of the INTERSECT clause is as follows:

```
select_statement INTERSECT select_statement
```

### Description

select_statement is a SELECT statement that does not contain an ORDER BY or FOR UPDATE clause.

The INTERSECT operator computes the set intersection of the rows returned by the involved SELECT statements. If a row is included in two result sets, the row is in the intersection of the two result sets.

The result of the INTERSECT clause does not contain duplicate rows.

Unless otherwise specified in parentheses, multiple INTERSECT operators in the same SELECT statement are evaluated from left to right. The INTERSECT clause binds tighter than the UNION clause. A UNION B INTERSECT C is read as A UNION (B INTERSECT C).

## 12.63.9 MINUS clause

The syntax of the MINUS clause is as follows:

```
select_statement MINUS select_statement
```

select_statement is a SELECT statement that does not contain an ORDER BY or FOR UPDATE clause.

The MINUS operator computes the set of rows that are in the result of the left SELECT statement but not in the result of the right one.

The result of the MINUS clause does not contain duplicate rows.

Unless otherwise specified in parentheses, multiple MINUS operators in the same SELECT statement are evaluated from left to right. The MINUS clause binds at the same level as the UNION clause.

## 12.63.10 CONNECT BY clause

The CONNECT BY clause determines the parent-child relationship of rows when performing a hierarchical query. The syntax of the CONNECT BY clause is as follows:

```
CONNECT BY { PRIOR parent_expr = child_expr |
 child_expr = PRIOR parent_expr }
```

parent_expr is evaluated on a candidate parent row. If parent_expr = child_expr results in TRUE for a row returned by the FROM clause, this row is considered a child of the parent.

The following optional clauses can be specified in conjunction with the CONNECT BY clause:

```
START WITH start_expression
```

The rows returned by the FROM clause on which start_expression evaluates to TRUE become the root nodes of the hierarchy.

```
ORDER SIBLINGS BY expression [ ASC | DESC ] [, ...]
```

Sibling rows of the hierarchy are ordered by expression in the result set.

**Note:**

PolarDB database compatible with Oracle does not support the use of AND or other operators in the CONNECT BY clause.

## 12.63.11 ORDER BY clause

The syntax of the optional ORDER BY clause is as follows:

```
ORDER BY expression [ ASC | DESC ] [, ...]
```

expression can be the name or ordinal number of an output column in the SELECT list. It can also be an arbitrary expression formed from input-column values.

The ORDER BY clause causes the result rows to be sorted according to the specified expressions. If two rows are equal according to the leftmost expression, they are compared according to the next expression. If they are equal according to all specified expressions, they are returned in an implementation-dependent order.

The ordinal number refers to the ordinal (left-to-right) position of the result column. This feature enables sorting based on a column that does not have a unique name. This is not necessary because you can use the AS clause to assign a name to a result column.

You can also use arbitrary expressions in the ORDER BY clause, including columns that do not appear in the SELECT output list. Therefore, the following statement is valid:

```
SELECT ename FROM emp ORDER BY empno;
```

An ORDER BY clause applying to the result of a UNION, INTERSECT, or MINUS clause can specify only an output column name or number rather than an expression.

If an ORDER BY expression is a simple name that matches both an output column name and an input column name, ORDER BY interprets it as the output column name. This is the opposite of the choice made by the GROUP BY clause in the same situation. This inconsistency is made to be compatible with the SQL standard.

You can add the ASC (ascending) or DESC (descending) keyword after any expression in the ORDER BY clause. If you specify neither ASC nor DESC, ASC is used.

The null value is sorted in a higher order than other values. In other words, null values are at the end of an ascending order and are at the beginning of a descending order.

String data is sorted based on the sorting rule set for specific regions created when the database cluster is initialized.

**Examples**

The following two examples show how to sort the results based on the content of the second column (dname):

```
SELECT * FROM dept ORDER BY dname;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     40 | OPERATIONS | BOSTON
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
(4 rows)

SELECT * FROM dept ORDER BY 2;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     40 | OPERATIONS | BOSTON
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
(4 rows)
```

## 12.63.12 DISTINCT clause

If you specify DISTINCT in a SELECT statement, all duplicate rows are removed from the result set. One row is retained from each group of duplicates. If you specify the ALL keyword instead, all rows are retained. This is the default value.

# 12.63.13 FOR UPDATE clause

### Syntax

```
FOR UPDATE [WAIT n|NOWAIT|SKIP LOCKED]
```

### Description

The FOR UPDATE clause causes the rows retrieved by the SELECT statement to be locked. This prevents a row from being modified or deleted by other transactions until the current transaction ends. All transactions that attempt to run the UPDATE, DELETE, or SELECT FOR UPDATE command on a selected row are blocked until the current transaction ends. If an UPDATE, DELETE, or SELECT FOR UPDATE command from another transaction has already locked a selected row or rows, SELECT FOR UPDATE waits for the previous transaction to complete. Then, SELECT FOR UPDATE locks and returns the updated rows. If the rows were deleted, SELECT FOR UPDATE locks and returns no rows.

FOR UPDATE cannot be used in contexts where returned rows cannot be clearly identified with individual table rows.

You can use FOR UPDATE options to specify locking preferences.

- Include the WAIT n keywords to specify the number of seconds or fractional seconds that the SELECT statement will wait for a row locked by another session. Use a decimal form to specify fractional seconds. For example, WAIT 1.5 instructs the server to wait one and a half seconds. You can specify a maximum of four digits to the right of the decimal point.
- Include the NOWAIT keyword to immediately report an error if a row cannot be locked by the current session.
- Include SKIP LOCKED to instruct the server to lock rows if possible, and skip rows that are already locked by another session.

# 12.64 SET CONSTRAINTS

Sets the constraint checking modes for the current transaction.

### Syntax

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

### Description

The SET CONSTRAINTS command sets the constraint check behavior in the current transaction. IMMEDIATE constraints are checked at the end of each statement. DEFERRED constraints are checked only after the transaction is committed. Each constraint has its own IMMEDIATE or DEFERRED mode.

When a constraint is created, one of the following three characteristics is assigned to the constraint: DEFERRABLE INITIALLY DEFERRED, DEFERRABLE INITIALLY IMMEDIATE, or NOT DEFERRABLE. The third class is always IMMEDIATE and is not affected by the SET CONSTRAINTS command. The first two classes start each transaction in the specified mode. You can use the SET CONSTRAINTS command to change the behavior of the first two classes in a transaction.

If you specify a list of constraint names, the SET CONSTRAINTS command changes the modes of the specified constraints. The specified constraints must be deferrable. If multiple constraints match a specified name, the modes of all the matching constraints are changed. The SET CONSTRAINTS ALL command changes the modes of all deferrable constraints.

If the SET CONSTRAINTS command changes the mode of a constraint from DEFERRED to IMMEDIATE, the new mode has a retroactive effect. During the execution of the SET CONSTRAINTS command, all unfinished data changes are checked. These data changes are no longer checked at the end of the transaction. If a constraint is violated, the SET CONSTRAINTS command fails and does not change the constraint mode. Therefore, the SET CONSTRAINTS command can be used to force constraints to be checked at a specific point in a transaction.

The setting of constraint checking modes affects only foreign key constraints. Check and UNIQUE constraints are not deferrable.

**Note:**

> This command changes the behavior of constraints only within the current transaction. If you run this command outside of a transaction block, the command has no effects.

## 12.65 SET ROLE

Sets the user identifier of the current session.

### Syntax

```
SET ROLE { rolename | NONE }
```

### Description

This command sets the user identifier of the current SQL session context to rolename. After you run the SET ROLE command, privileges that the specified role have on SQL commands are checked.

The specified rolename must be a role of the current session user.

### Notes

You can use this command to add or restrict the privileges of a user. If the session user role has the INHERITS attribute, it is automatically assigned the privileges to run the SET ROLE command on all roles. In this case, the SET ROLE command deletes all the privileges assigned to the session user and to the other roles of the user. Only the privileges available to the specified role are retained. If the session user role has the NOINHERITS attribute, the SET ROLE deletes the privileges assigned to the session user and retains the privileges available to the specified role. If a superuser runs the SET ROLE command to set the user role to a non-superuser role, the superuser no longer has superuser privileges.

### Examples

Run the following command to set the role of user mary to admins:

```
SET ROLE admins;
```

Run the following command to set the role of the user back to mary:

```
SET ROLE NONE;
```

## 12.66 SET TRANSACTION

Sets the characteristics of the current transaction.

### Syntax

```
SET TRANSACTION transaction_mode
```

Transaction_mode can be one of the following options:

```
  ISOLATION LEVEL { SERIALIZABLE | READ COMMITTED }
  READ WRITE | READ ONLY
```

### Description

The SET TRANSACTION command sets the characteristics of the current transaction. This command has no effect on subsequent transactions. The available transaction characteristics are the transaction isolation level and the transaction access mode (read/write or read-only). The isolation level of a transaction determines what data the transaction can read when other transactions are running concurrently.

- READ COMMITTED

  A statement can read only rows that are committed before the statement starts. This is the default value.

- SERIALIZABLE

  All statements of the current transaction can read only rows that are committed before the first query or before data modification statement is executed in this transaction.

After the first query or data modification statement (SELECT, INSERT, DELETE, UPDATE, or FETCH) is executed, the transaction isolation level cannot be changed. The transaction access mode determines whether the transaction is read/write or read-only. The default value is read/write.

A read-only transaction does not support the following SQL commands: CREATE, ALTER , DROP, COMMENT, GRANT, REVOKE, and TRUNCATE. The read-only transaction does not support the INSERT, UPDATE, and DELETE commands if the table to which these commands write is not a temporary table. The read-only transaction does not support the EXECUTE command if one of the listed commands is executed within the transaction. This is an advanced read-only mode that does not block all write operations on a disk.

## 12.67 TRUNCATE

Clears a table.

### Syntax

```
TRUNCATE TABLE name [DROP STORAGE]
```

### Description

The TRUNCATE command removes all rows from a table. This command has the same effect as an unqualified DELETE command. However, the TRUNCATE command is faster because it does not scan the table. This is most useful for large tables.

The DROP STORAGE clause is used for compatibility, but is ignored.

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the table to be truncated. The name can be schema-qualified. |

### Notes

If other tables have foreign-key references to the table to be truncated, you cannot use the TRUNCATE command. This is because table scans are required for validity check.

The TRUNCATE command does not run user-defined ON DELETE triggers for the table even if you have configured such triggers.

### Examples

Run the following command to truncate the bigtable table:

```
TRUNCATE TABLE bigtable;
```

## 12.68 UPDATE

Updates rows of a table.

### Syntax

```
UPDATE [ optimizer_hint ] table[@dblink ]
   SET column = { expression | DEFAULT } [, ...]
  [ WHERE condition ]
  [ RETURNING return_expression [, ...]
      { INTO { record | variable [, ...] }
      | BULK COLLECT INTO collection [, ...] } ]
```

### Description

The UPDATE command changes the values of the specified columns in all rows that satisfy the condition. You only need to specify the columns to be modified in the SET clause. Columns that are not specified retain their previous values.

You can specify the RETURNING INTO { record | variable [, ...] } clause only within an SPL program. In addition, the result set of the UPDATE command cannot return multiple rows. Otherwise, an exception occurs. If the result set is empty, the content of the target record or variables is set to null.

You can specify the RETURNING BULK COLLECT INTO collection [, ...] clause only if the UPDATE command is used within an SPL program. If more than one collection is specified as the target of the BULK COLLECT INTO clause, each collection must contain a scalar field. In other words, collection cannot be a record. The result set of the UPDATE command can contain none, one, or more rows. return_expression that is evaluated for each row of the result set is an element in collection starting from the first element. All existing rows in collection are deleted. If the result set is empty, collection is also empty.

To update a table, you must have the UPDATE privilege for the table and the SELECT privilege for all tables whose values are read in expression or condition.

**Parameters**

| Parameter | Description |
|---|---|
| optimizer_hint | Comment-embedded hints to the optimizer. This parameter is used to select an execution plan. |
| table | The name of the table to be updated. The name can be schema-qualified. |
| dblink | The name of the database link. This parameter is used to identify a remote database. For more information about database links, see the CREATE DATABASE LINK command. |
| column | The name of a column in the table. |
| expression | An expression to assign to the column. The expression can use the old values of this column and other columns in the table. |
| DEFAULT | The default expression of the column. If no specific default expression is assigned, the default value is null. |
| condition | An expression that returns a value of the BOOLEAN type. Only rows for which this expression returns true are updated. |
| return_expression | An expression that includes one or more columns from the table. If you specify a column name from the table in return_expression , the value substituted for the column when return_expression is evaluated is determined as follows:<br>• If the column specified in return_expression is assigned a value in the UPDATE command, the assigned value is used in the evaluation of return_expression.<br>• If the column specified in return_expression is not assigned a value in the UPDATE command, the current value of the column in the affected row is used in the evaluation of return_expression. |
| record | A record that contains fields to which the evaluated return_expression is assigned. The first return_expression is assigned to the first field in record. The second return_expression is assigned to the second field in record. The number of fields in record must be the same as the number of expressions. The fields must be type-compatible with their assigned expressions. |

| Parameter | Description |
|---|---|
| variable | A variable to which the evaluated return_expression is assigned. If more than one return_expression and variable are specified, the first return_expression is assigned to the first variable and the second return_expression is assigned to the second variable . The number of variables specified following the INTO keyword must be the same as the number of expressions following the RETURNING keyword. The variables must be type-compatible with their assigned expressions. |
| collection | A collection in which an element is created from the evaluated return_expression. One or more collections can exist. A single collection can be a collection of a single field or a collection of a record type. If multiple collections exist, each collection must consist of a single field. The number and sequence of returned expressions must be the same as the number and sequence of fields in all specified collections. Each corresponding return_expression and collection fields must be type-compatible. |

**Examples**

Run the following command to change the location to AUSTIN for department 20 in the dept table:

```
UPDATE dept SET loc = 'AUSTIN' WHERE deptno = 20;
```

For all employees with job = SALESMAN in the emp table, run the following command to update the salary by 10% and increase the commission by 500:

```
UPDATE emp SET sal = sal * 1.1, comm = comm + 500 WHERE job = 'SALESMAN';
```

# 13 Built-in functions

## 13.1 Logical operators

The usual logical operators are AND, OR, and NOT.

SQL uses a three-valued Boolean logic where the null value represents "unknown". For more information, see the following truth tables.

**Table 13-1: AND/OR truth table**

| a | b | a AND b | a OR b |
|---|---|---|---|
| True | True | True | True |
| True | False | False | True |
| True | Null | Null | True |
| False | False | False | False |
| False | Null | False | Null |
| Null | Null | Null | Null |

**Table 13-2: NOT truth table**

| a | NOT a |
|---|---|
| True | False |
| False | True |
| Null | Null |

The operators AND and OR are commutative. You can switch the left and right operand without affecting the result.

## 13.2 Comparison operators

The following table lists the frequently used comparison operators.

**Table 13-3: Comparison operators**

| Operator | Description |
|---|---|
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| = | Equal |
| <> | Not equal |
| ! = | Not equal |

You can use comparison operators for all valid data types. All comparison operators are binary operators that return boolean values. Expressions like 1 < 2 < 3 are invalid (because no < operator is available to compare a Boolean value with 3).

In addition to the comparison operators, you can also use the BETWEEN construct.

-
  ```
  a BETWEEN x AND y
  ```

  is equivalent to

  ```
  a >= x AND a <= y
  ```

-
  ```
  a NOT BETWEEN x AND y
  ```

  is equivalent to

  ```
  a < x OR a > y
  ```

No difference exists between the two expression forms except that the CPU cycles require that you internally rewrite the first one into the second one.

To check whether a value is null, you can use the following constructs:

```
expression IS NULL
expression IS NOT NULL
```

Do not use expression = NULL because NULL is not equal to the null value. (The null value represents an unknown value, and it cannot be determined whether two unknown values are equal). This behavior complies with the SQL standard.

If expression evaluates to the null value, some applications may expect that expression = NULL returns true. We recommend that you modify these applications to comply with the SQL standard.

## 13.3 Mathematical functions and operators

Mathematical operators are provided to manipulate values of data types supported by POLARDB compatible with Oracle. For types without common mathematical conventions for all possible permutations (for example, date/time types), the actual behavior is described in subsequent sections.

The following table shows the allowed mathematical operators.

**Table 13-4: Mathematical operators**

| Operator | Description | Example | Result |
|---|---|---|---|
| + | Addition | 2 + 3 | 5 |
| - | Subtraction | 2 - 3 | -1 |
| * | Multiplication | 2 * 3 | 6 |
| / | Division (integer division truncates results) | 4 / 2 | 2 |
| ** | Exponentiation operator | 2 ** 3 | 8 |

The following table shows the available mathematical functions. Many of these functions are provided in multiple forms with different argument types. Note that any form of function returns the same data type as its argument. The functions that involve DOUBLE PRECISION data are mostly implemented on top of the C library of the host system. The accuracy and behavior in boundary cases may vary depending on the host system.

**Table 13-5: Mathematical functions**

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| ABS(x) | Same as x | The absolute value. | ABS(-17.4) | 17.4 |
| CEIL(DOUBLE PRECISION or NUMBER) | Same as input | The smallest integer not less than argument. | CEIL(-42.8) | -42 |

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| EXP(DOUBLE PRECISION or NUMBER) | Same as input | Exponential | EXP(1.0) | 2.7182818284 5904 52 |
| FLOOR(DOUBLE PRECISION or NUMBER) | Same as input | The largest integer not greater than argument. | FLOOR(-42.8) | 43 |
| LN(DOUBLE PRECISION or NUMBER) | Same as input | Natural logarithm | LN(2.0) | 0.6931471805 5994 53 |
| LOG(b NUMBER, X NUMBER) | NUMBER | The logarithm to base b. | LOG(2.0, 64.0) | 6.0000000000 0000 00 |
| MOD(y, X) | Same as argument types | The remainder of y/x. | MOD(9, 4) | 1 |
| NVL(x, y) | Same as argument types ; where both arguments are of the same data type. | If X is null, NVL returns y. | NVL(9, 0) | 9 |
| POWER(a DOUBLE PRECISION, b DOUBLE PRECISION) | DOUBLE PRECISION | a raised to the power of b | POWER(9.0, 3.0) | 729.0000000000 00 0000 |
| POWER(a NUMBER, b NUMBER) | NUMBER | a raised to the power of b | POWER(9.0, 3.0) | 729.0000000000 00 0000 |
| ROUND(DOUBLE PRECISION or NUMBER) | Same as input | Rounds to the nearest integer. | ROUND(42.4) | 42 |
| ROUND(v NUMBER, s INTEGER) | NUMBER | Rounds to s decimal places. | ROUND(42.4382 , 2) | 42.44 |
| SIGN(DOUBLE PRECISION or NUMBER) | Same as input | Sign of the argument (-1, 0 , +1) | SIGN(-8.4) | -1 |

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| SQRT(DOUBLE PRECISION or NUMBER) | Same as input | Square root | SQRT(2.0) | 1.4142135623 7309 5 |
| TRUNC(DOUBLE PRECISION or NUMBER) | Same as input | Truncates toward zero. | TRUNC(42.8) | 42 |
| TRUNC(v NUMBER, s INTEGER) | NUMBER | Truncates to s decimal places. | TRUNC(42.4382, 2) | 42.43 |
| WIDTH BUCKET ( op NUMBER, b1 NUMBER, b2 NUMBER, count INTEGER) | INTEGER | Returns the bucket to which op will be assigned in an equidepth histogram with count buckets, in the range b1 to b2. | WIDTH BUCKET(5 .35, 0.024, 10.06 , 5) | 3 |

The following table shows the available trigonometric functions. The arguments and return values of all trigonometric functions are of type DOUBLE PRECISION.

**Table 13-6: Trigonometric functions**

| Function | Description |
|---|---|
| ACOS(x) | Inverse cosine |
| ASIN(x) | Inverse sine |
| ATAN(x) | Inverse tangent |
| ATAN2 (x, y) | Inverse tangent of x/y |
| COS(x) | Cosine |
| SIN(x) | Sine |
| TAN(x) | Tangent |

## 13.4 String functions and operators

This topic describes functions and operators that are used to identify and manipulate string values. Strings include values of the CHAR, VARCHAR2, and CLOB types. Note that the functions listed below can work on all these types of values, but be aware of the potential effects of automatic padding when using the CHAR type. In most cases, the functions described here can also work on data of non-string types by first converting the data into values of string types.

**Table 13-7: SQL string functions and operators**

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| string ‖ string | CLOB | String concatenation | 'Enterprise' ‖ 'DB' | EnterpriseDB |
| CONCAT(string, string) | CLOB | String concatenation | CONCAT('a' ‖ 'b') | ab |
| HEXTORAW ( varchar2) | RAW | Converts a VARCHAR2 value to a RAW value. | HEXTORAW(' 303132') | '012' |
| RAWTOHEX(raw) | VARCHAR2 | Converts a RAW value to a HEXADECIMAL value. | RAWTOHEX ('012 ') | '303132' |
| INSTR(string , set, [ start [, occurrence ] ]) | INTEGER | Finds the location of a set of characters in a string, starting at position start in the string , string, and looking for the first, second, third and so on occurrences of the set. Returns 0 if the set is not found. | INSTR('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PI',1,3) | 30 |
| INSTRB(string, set) | INTEGER | Returns the position of the set within the string. Returns 0 if set is not found. | INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS', 'PICK') | 13 |

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| INSTRB(string, set, start) | INTEGER | Returns the position of the set within the string, beginning at start. Returns 0 if set is not found. | INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 14) | 30 |
| INSTRB(string, set, start, occurrence) | INTEGER | Returns the position of the specified occurrence of set within the string, beginning at start. Returns 0 if set is not found. | INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 1, 2) | 30 |
| LOWER(string) | CLOB | Converts string to lowercase. | LOWER('TOM') | tom |
| SUBSTR(string, start [, count ]) | CLOB | Extracts substring starting from start and going for count characters. If count is not specified, the string is clipped from the start till the end. | SUBSTR('This is a test',6,2) | is |
| SUBSTRB(string, start [, count ]) | CLOB | Same as SUBSTR except start and count are in number of bytes. | SUBSTRB('abc',3) (assuming a double-byte character set) | c |
| SUBSTR2(string, start[, count ]) | CLOB | Alias for SUBSTR. | SUBSTR2('This is atest',6,2) | is |
| SUBSTR2(string, start [, count ]) | CLOB | Alias for SUBSTRB. | SUBSTR2('abc',3) (assuming a double-byte character set) | c |

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| SUBSTR4(string, start [, count ]) | CLOB | Alias for SUBSTR . | SUBSTR4('This is a test',6,2) | is |
| SUBSTR4 (string , start [, count]) | CLOB | Alias for SUBSTRB. | SUBSTR4('abc ',3) (assuming a double-byte character set) | c |
| SUBSTRC(string, start [, count ]) | CLOB | Alias for SUBSTR . | SUBSTRC('This is a test',6,2) | is |
| SUBSTRC(string, start [, count ]) | CLOB | Alias for SUBSTRB. | SUBSTRC('abc ',3) (assuming a double-byte character set) | c |
| TRIM([ LEADING \| TRAILING \| BOTH ] [ characters ] FROM string) | CLOB | Removes the longest string containing only the characters (a space by default) from the start/end/both ends of the string. | TRIM(BOTH 'x' FROM 'xTomxx') | Tom |
| LTRIM(string [, set]) | CLOB | Removes all the characters specified in set from the left of a given string. If set is not specified, a blank space is used as default. | LTRIM(' abcdefghi', 'abc ') | defghi |
| RTRIM(string [, set]) | CLOB | Removes all the characters specified in set from the right of a given string. If set is not specified, a blank space is used as default. | RTRIM(' abcdefghi', 'ghi') | abcdef |

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| UPPER(string) | CLOB | Converts string to uppercase | UPPER('tom') | TOM |

The following table lists other available string manipulation functions. Some of the functions are used internally to implement the SQL-standard string functions listed in Table 13-3: Comparison operators.

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| ASCII(string) | INTEGER | ASCII code of the first byte of the argument | ASCII('x') | 120 |
| CHR(INTEGER) | CLOB | Character with the given ASCII code | CHR(65) | A |
| DECODE(expr, exprla, exprlb [, expr2a, expr2b ]... [, default ]) | Same as argument types of expr1b, expr2b,..., default | Finds the first match of expr with expr1a , expr2a, etc. When the match is found, returns corresponding parameter pair, expr1b, expr2b, etc. If no match is found, returns default. If no match is found and default is not specified, returns null. | DECODE(3, 1,' One', 2,'Two', 3,'Three', 'Not found') | Three |

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| INITCAP(string) | CLOB | Converts the first letter of each word to uppercase and the rest to lowercase. Words are sequences of alphanumeric characters separated by non-alphanumeric characters. | INITCAP('hi THOMAS') | Hi Thomas |
| LENGTH | INTEGER | Returns the number of characters in a string value. | LENGTH('Coted'' Azur') | 11 |
| LENGTHC | INTEGER | This function is identical in functionality to LENGTH. The function name is supported for compatibility. | LENGTHC ('Cote d''Azur') | 11 |
| LENGTH2 | INTEGER | This function is identical in functionality to LENGTH. The function name is supported for compatibility. | LENGTH2 ('Cote d''Azur') | 11 |
| LENGTH4 | INTEGER | This function is identical in functionality to LENGTH. The function name is supported for compatibility. | LENGTH4 ('Cote d''Azur') | 11 |

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| LENGTHB | INTEGER | Returns the number of bytes required to hold the given value. | LENGTHB ('Cote d''Azur') | 12 |
| LPAD(string, length INTEGER [, fill ]) | CLOB | Fills up string to size, length by prepending the characters , fill (a space by default). If string is longer than length, it is truncated (on the right). | LPAD('hi', 5, 'xy') | xyxhi |
| REPLACE(string, search string [, replace string ] | CLOB | Replaces one value in a string with another . If you do not specify a value for replace string, the search_string value when found, is removed. | REPLACE( ' GEORGE', 'GE', ' EG') | EGOREG |
| RPAD(string, length INTEGER [, fill ]) | CLOB | Fills up string to size, length by appending the characters , fill (a space by default). If string is already longer than length, it is truncated. | RPAD('hi', 5, 'xy') | hixyx |

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| TRANSLATE( string, from, to) | CLOB | Any character in string that matches a character in the from set is replaced by the corresponding character in the to set. | TRANSLATE(' 12345', '14', 'ax') | a23x5 |

## 13.5 Pattern matching string functions

### 13.5.1 Overview

POLARDB compatible with Oracle supports the functions: REGEXP_COUNT, REGEXP_INSTR, and REGEXP_SUBSTR. These functions are used to perform a search on a string for a specific pattern that is specified by a regular expression. This will then return specific information about occurrences of the pattern within the string. The pattern must be a POSIX-style regular expression. For more information about POSIX-style regular expressions, see the core documentation available at: http://www.enterprisedb.com/docs/en/9.3/pg/functions-matching.html

### 13.5.2 REGEXP_COUNT

The REGEXP_COUNT function searches a string for a regular expression and returns the number of times that the regular expression occurs.

**Syntax**

```
INTEGER REGEXP_COUNT
(
  srcstr    TEXT,
  pattern   TEXT,
  position  DEFAULT 1
  modifier  DEFAULT NULL
)
```

**Parameters**

| Parameter | Description |
|---|---|
| srcstr | The string to search. |

| Parameter | Description |
|---|---|
| pattern | The regular expression for which REGEXP_COUNT will search. |
| position | An integer value that indicates the position in the source string at which REGEXP_COUNT will start searching. The default value is 1. |
| modifier | The values that control the pattern matching behavior. The default value is NULL. |

**Note:**

For a complete list of the modifiers supported by POLARDB compatible with Oracle, see the PostgreSQL core documentation available at: http://www.enterprisedb.com/docs/en/9.3/pg/functions-matching.html

**Examples**

In the following example, REGEXP_COUNT returns the number of times the letter i is used in the string 'reinitializing':

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 1) FROM DUAL;
 regexp_count
--------------
            5
(1 row)
```

In the first example, the command instructs REGEXP_COUNT to start counting in the first position. If you want to start counting in the sixth position, use the following command:

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 6) FROM DUAL;
 regexp_count
--------------
            3
(1 row)
```

Then REGEXP_COUNT function returns 3, and the count does not include occurrences of the letter i that occur before the sixth position.

## 13.5.3 REGEXP_INSTR

The REGEXP_INSTR function searches a string for a POSIX-style regular expression. This function returns the position within the string where the match is located.

### Syntax

```
INTEGER REGEXP_INSTR
(
  srcstr       TEXT,
  pattern      TEXT,
  position     INT  DEFAULT 1,
  occurrence   INT  DEFAULT 1,
  returnparam  INT  DEFAULT 0,
  modifier     TEXT DEFAULT NULL,
  subexpression INT  DEFAULT 0,
)
```

### Parameters

| Parameter | Description |
|---|---|
| srcstr | The string to search. |
| pattern | The regular expression for which REGEXP_INSTR will search. |
| position | An integer value that indicates the start position in a source string. The default value is 1. |
| occurrence | Specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is 1. |
| returnparam | An integer value that specifies the location within the string that REGEXP_INSTR returns as expected. The default value is 0. Specify:<br>• 0 to return the location within the string of the first character that matches the pattern.<br>• A value greater than 0 to return the location of the first character following the end of the pattern. |

| Parameter | Description |
|---|---|
| modifier | The values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by POLARDB compatible with Oracle, see the PostgreSQL core documentation available at: http://www.enterprisedb.com/docs/en/9.3/pg/functions-matching.html |
| subexpression | An integer value that identifies the portion of the pattern that will be returned by REGEXP_INSTR. The default value of subexpression is 0.<br>If you specify a value for subexpression , you must include one (or multiple) set of parentheses in the pattern to isolate a portion of the value being searched . The value specified by subexpression indicates which set of parentheses will be returned. For example, if the value of subexpression is 2, REGEXP_INSTR returns the value contained within the second set of parentheses. |

**Examples**

In the following example, REGEXP_INSTR searches a string that contains a phone number for the first occurrence of a pattern that contains three consecutive digits:

```
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM DUAL;
 regexp_instr
--------------
            1
(1 row)
```

The command instructs REGEXP_INSTR to return the position of the first occurrence. If you want to return the start of the second occurrence of three consecutive digits, use the following command:

```
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;
 regexp_instr
--------------
            5
```

```
(1 row)
```

## 13.5.4 REGEXP_SUBSTR

The REGEXP_SUBSTR function searches a string for a pattern specified by a POSIX compliant regular expression. This function returns the string that matches the pattern specified in the call to the function.

### Syntax

```
TEXT REGEXP_SUBSTR
(
 srcstr       TEXT,
 pattern      TEXT,
 position     INT  DEFAULT 1,
 occurrence   INT  DEFAULT 1,
 modifier     TEXT DEFAULT NULL,
 subexpression INT  DEFAULT 0
)
```

### Parameters

| Parameter | Description |
|---|---|
| srcstr | The string to search. |
| pattern | The regular expression for which REGEXP_SUBSTR will search. |
| position | An integer value that indicates the start position in a source string. The default value is 1. |
| occurrence | Specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is 1. |
| modifier | The values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by POLARDB compatible with Oracle, see the PostgreSQL core documentation available at: http://www.enterprisedb.com/docs/en/9.3/pg/functions-matching.html |

| Parameter | Description |
|---|---|
| subexpression | An integer value that identifies the portion of the pattern that will be returned by REGEXP_SUBSTR. The default value of subexpression is 0.<br>If you specify a value for subexpression , you must include one (or multiple) set of parentheses in the pattern to isolate a portion of the value being searched . The value specified by subexpression indicates which set of parentheses will be returned. For example, if the value of subexpression is 2, REGEXP_SUBSTR returns the value contained within the second set of parentheses. |

**Examples**

In the following example, the REGEXP_SUBSTR searches a string that contains a phone number for the first set of three consecutive digits:

```
edb=# SELECT REGEXP_SUBSTR('800-555-****', '[0-9][0-9][0-9]', 1, 1) FROM DUAL;
 regexp_substr
---------------
 800
(1 row)
```

The function locates the first occurrence of three digits and returns the string (8 0 0). If you want to search for the second occurrence of three consecutive digits, use the following command:

```
edb=# SELECT REGEXP_SUBSTR('800-555-****', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;
 regexp_substr
---------------
 555
(1 row)
```

REGEXP_SUBSTR returns 555, which is the content of the second substring.

## 13.6 Use the LIKE operator for pattern matching

POLARDB compatible with Oracle provides pattern matching by using the traditional SQL LIKE operator. The syntax of the LIKE operator is as follows.

```
string LIKE pattern [ ESCAPE escape-character ]
string NOT LIKE pattern [ ESCAPE escape-character ]
```

Each pattern parameter defines a set of strings. The LIKE expression returns true if the set of strings represented by pattern contains the value specified by the string parameter. As expected, a reciprocal inverse relationship exists and the NOT LIKE expression returns FALSE if LIKE returns TRUE. An equivalent expression of NOT LIKE is NOT (string LIKE pattern).

If the pattern does not contain percent signs (%) or underscores (_), the pattern only represents the string itself. In this case, the LIKE operator acts like the equals operator. An underscore (_) in pattern matches any single character. A percent sign (%) matches any string of zero or more characters.

**Examples:**

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

LIKE pattern matches cover the entire string. If you want to start matching a pattern from any position in the string, the pattern must start and end with a percent sign.

If you want to match a literal underscore or percent sign without matching other characters , you must precede the respective character in the pattern by an escape character. The default escape character is a backslash (\). However, you can also use the ESCAPE clause to specify a different escape character. To match the escape character itself, write two escape characters.

Note that the backslash already has a specific meaning in string literals. To write a pattern that contains a backslash, you must write two backslashes in an SQL statement. Therefore , writing a pattern that matches a literal backslash means writing four backslashes in the statement. You can avoid this by using the ESCAPE clause to specify a different escape character. Then, a backslash does not provide a special meaning to LIKE anymore. ( However, the backslash still has a special meaning for the string literal parser, and two backslashes are still required.)

You can also select no escape character by writing ESCAPE ''. This effectively disables the escape mechanism, which makes it impossible to disable the special meaning of underscores and percent signs in the pattern.

## 13.7 Functions for formatting data types

Formatting functions of POLARDB compatible with Oracle provide a powerful set of tools for converting various data types (date/time, integer, floating point, and numeric) into formatted strings. These functions can also convert formatted strings into specific data types. The following table describes these formatting functions. These functions follow a common calling convention. The first argument is the value to be formatted and the second argument is a string template that defines the output or input format.

**Table 13-8: Formatting functions**

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| TO CHAR(DATE [, format ]) | VARCHAR2 | Converts a date /time to a string in the format specified by the format argument. If you omit the format argument, the function returns a string in the default format ( DD-MON- YY). | TO CHAR( SYSDATE, 'MM/ DD/YYYY HH12: MI:SS AM') | 07/25/2007 09: 43:02 AM |
| TO CHAR( INTEGER [, format ]) | VARCHAR2 | Converts an integer to a string in the format specified by the format argument. | TO CHAR(2412, ' 999,999S') | 2,412+ |
| TO CHAR( NUMBER [, format ]) | VARCHAR2 | Converts a decimal number to a string in the format specified by the format argument. | TO CHAR(10125. 35, '999,999.99') | 10,125.35 |

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| TO CHAR (DOUBLE PRECISION, format) | VARCHAR2 | Converts a floating-point number to a string in the format specified by the format argument. | TO CHAR(CAST (123.5282 AS REAL), '999.99') | 123.53 |
| TO DATE(string [, format ]) | DATE | Converts a date formatted string to a DATE data type. | TO DATE('2007-07-04 13:39:10 ', 'YYYY-MM-DD HH24:MI:SS') | 04-JUL-07 13:39 :10 |
| TO NUMBER( string [, format ]) | NUMBER | Converts a number formatted string to a NUMBER data type. | TO NUMBER('2, 412-', '999,999S ') | -2412 |
| TO TIMESTAMP( string, format) | TIMESTAMP | Converts a timestamp formatted string to a TIMESTAMP data type. | TO TIMESTAMP(' 05 Dec 2000 08: 30:25 pm', 'DD Mon YYYY hh12: mi:ss pm') | 05-DEC-00 20:30 :25 |

In an output template string for the TO_CHAR function, some specific patterns are recognized and replaced with appropriately-formatted data from the value to be formatted . Any text that is not a template pattern is an exact copy. Similarly, in an input template string (for any function but TO_CHAR), template patterns identify the parts of the input data string to be looked at and the values to be found there.

The following table lists the available template patterns for formatting date values by using the TO_CHAR and TO_DATE functions.

**Table 13-9: Template date/time format patterns**

| Pattern | Description |
|---|---|
| HH | Hour of day (01-12) |
| HH12 | Hour of day (01-12) |
| HH24 | Hour of day (00-23) |
| MI | Minute (00-59) |

| Pattern | Description |
|---|---|
| SS | Second (00-59) |
| SSSSS | Seconds past midnight (0-86399) |
| AM or A.M. or PM or P.M. | Meridian indicator (uppercase) |
| am or a.m. or pm or p.m. | Meridian indicator (lowercase) |
| Y,YYY | Year (4 and more digits) with comma |
| YEAR | Year (spelled out) |
| SYEAR | Year (spelled out) (BC dates prefixed by a minus sign) |
| YYYY | Year (4 and more digits) |
| SYYYY | Year (4 and more digits) (BC dates prefixed by a minus sign) |
| YYY | Last 3 digits of year |
| YY | Last 2 digits of year |
| Y | Last digit of year |
| IYYY | ISO year (4 and more digits) |
| IYY | Last 3 digits of ISO year |
| IY | Last 2 digits of ISO year |
| I | Last 1 digit of ISO year |
| BC or B.C. or AD or A.D. | Era indicator (uppercase) |
| bc or b.c. or ad or a.d. | Era indicator (lowercase) |
| MONTH | Full uppercase month name |
| Month | Full mixed-case month name |
| month | Full lowercase month name |
| MON | Abbreviated uppercase month name (3 characters in English, localized lengths vary) |
| Mon | Abbreviated mixed-case month name (3 characters in English, localized lengths vary) |
| mon | Abbreviated lowercase month name (3 characters in English, localized lengths vary) |
| MM | Month number (01-12) |
| DAY | Full uppercase day name |

| Pattern | Description |
|---|---|
| Day | Full mixed-case day name |
| day | Full lowercase day name |
| DY | Abbreviated uppercase day name (3 characters in English, localized lengths vary) |
| Dy | Abbreviated mixed-case day name (3 characters in English, localized lengths vary) |
| dy | Abbreviated lowercase day name (3 characters in English, localized lengths vary) |
| DDD | Day of year (001-366) |
| DD | Day of month (01-31) |
| D | Day of week (1-7. Sunday is 1) |
| W | Week of month (1-5) (The first week starts on the first day of the month.) |
| WW | Week number of year (1-53) (The first week starts on the first day of the year.) |
| IW | ISO week number of year. The first Thursday of the new year is in week 1. |
| CC | Century (2 digits). The 21st century starts on 2001-01-01. |
| SCC | Same as CC except BC dates are prefixed by a minus sign. |
| J | Julian Day (days since January 1, 4712 BC) |
| Q | Quarter |
| RM | Month in Roman numerals (I-XII. I=January) ( uppercase) |
| rm | Month in Roman numerals (i-xii. i=January) ( lowercase) |

| Pattern | Description |
|---|---|
| RR | The first 2 digits of the year when given only the last 2 digits of the year. The result is based upon an algorithm using the current year and the given 2-digit year. The first 2 digits of the given 2-digit year will be the same as the first 2 digits of the current year with the following exceptions:<br>• If the given 2-digit year is < 50 and the last 2 digits of the current year is >= 50, then the first 2 digits for the given year is 1 greater than the first 2 digits of the current year.<br>• If the given 2-digit year is >= 50 and the last 2 digits of the current year is < 50, then the first 2 digits for the given year is 1 less than the first 2 digits of the current year. |
| RRRR | Only affects the TO_DATE function. Allows specification of 2-digit or 4-digit year. If 2-digit year given, then returns first 2 digits of year like RR format. If 4-digit year given, returns the given 4-digit year. |

Specific modifiers may be applied to any template pattern to alter its behavior. For example, FMMonth is the Month pattern with the FM modifier. The following table lists the pattern modifiers for date/time formatting.

**Table 13-10: Template pattern modifiers for date/time formatting**

| Modifier | Description | Example |
|---|---|---|
| FM prefix | Fill mode (suppress padding blanks and zeros) | FMMonth |
| TH suffix | Uppercase ordinal number suffix | DDTH |
| th suffix | Lowercase ordinal number suffix | DDth |
| FX prefix | Fixed format global option ( see note) | FX Month DD Day |
| SP suffix | Spell mode | DDSP |

**Note:**

- FM suppresses leading zeros and trailing blanks that would otherwise be added to ensure that the output conforms to a fixed width pattern.
- If the FX option is not used, TO_TIMESTAMP and TO_DATE skip multiple blank spaces in the input string. You must specify FX as the first item in the template. For example, TO_TIMESTAMP('2000 JUN', 'YYYY MON') is valid, but TO_TIMESTAMP('2000 JUN', 'FXYYYY MON') returns an error, because TO_TIMESTAMP only expects one space.
- Ordinary text is allowed in TO_CHAR templates and will be output literally.
- In conversions from string to timestamp or date, the CC field is ignored if a YYY, YYYY or Y,YYY field exists. If CC is used with the YY or Y field, the year is computed as (CC-1)*100 +YY.

The following table shows the available template patterns for formatting numeric values.

**Table 13-11: Template patterns for numeric formatting**

| Pattern | Description |
|---|---|
| 9 | Value with the specified number of digits |
| 0 | Value with leading zeroes |
| . (period) | Decimal point |
| , (comma) | Group (thousand) separator |
| $ | Dollar sign |
| PR | Negative value in angle brackets |
| S | Sign anchored to number (uses locale) |
| L | Currency symbol (uses locale) |
| D | Decimal point (uses locale) |
| G | Group separator (uses locale) |
| MI | Minus sign specified in right-most position ( if number < 0) |
| RN or rn | Roman numeral (input between 1 and 3999) |
| V | Shift specified number of digits (see note) |

**Note:**

- 9 results in a value with the same number of digits as there are 9s. If a digit is not available or specified, a space is output.
- TH does not convert values less than zero or fractional numbers.

V effectively multiplies the input values by 10n (10 to the power of n), where n indicates the number of digits following V. TO_CHAR does not support the use of V combined with a decimal point. (For example, 99.9V99 is not allowed.)

The following table shows some examples about how to use TO_CHAR and TO_DATE.

| Expression | Result |
|---|---|
| TO CHAR(CURRENT TIMESTAMP, 'Day, DD HH12:MI:SS') | 'Tuesday , 06 05:39:18' |
| TO CHAR(CURRENT TIMESTAMP, 'FMDay, FMDD HH12:MI:SS') | 'Tuesday, 6 05:39:18' |
| TO CHAR(-0.1, '99.99') | ' -.10' |
| TO CHAR(-0.1, 'FM9.99') | '-.1' |
| TO CHAR(0.1, '0.9') | ' 0.1' |
| TO CHAR(12, '9990999.9') | ' 0012.0' |
| TO CHAR(12, 'FM9990999.9') | '0012.' |
| TO CHAR(485, '999') | ' 485' |
| TO CHAR(-485, '999') | ' -485' |
| TO CHAR(1485, '9,999') | ' 1,485' |
| TO CHAR(1485, '9G999') | ' 1,485' |
| TO CHAR(148.5, '999.999') | ' 148.500' |
| TO CHAR(148.5, 'FM999.999') | '148.5' |
| TO CHAR(148.5, 'FM999.990') | '148.500' |
| TO CHAR(148.5, '999D999') | ' 148.500' |
| TO CHAR(3148.5, '9G999D999') | ' 3,148.500' |
| TO CHAR(-485, '999S') | '485- ' |
| TO CHAR(-485, '999MI') | '485- ' |
| TO CHAR(485, '999MI') | '485 ' |
| TO CHAR(4 85, 'FM999MI') | '485' |
| TO CHAR(-485, '999PR') | '<485>' |

| Expression | Result |
|---|---|
| TO CHAR(485, 'L999') | '$ 485' |
| TO CHAR(4 85, 'RN') | ' CDLXXXV' |
| TO CHAR(4 85, 'FMRN') | 'CDLXXXV' |
| TO CHAR(5.2, 'FMRN') | 'V' |
| TO CHAR(12, '99V999') | ' 12000' |
| TO CHAR(12.4, '99V999') | ' 12400' |
| TO CHAR(12.45, '99V9') | ' 125' |

# 13.8 Date/Time functions and operators

## 13.8.1 Overview

Table 13-13: Date/Time functions shows the available functions that can be used to process date/time values. For more information about these functions, see the subsequent topics. Table 13-12: Date/Time operators illustrates the behaviors of the basic arithmetic operators (+, -). For formatting functions, see the Functions for formatting data types topic. You need to be familiar with the background information on date/time data types from topic Date and time type.

**Table 13-12: Date/Time operators**

| Operator | Example | Result |
|---|---|---|
| + | DATE '2001-09-28' + 7 | 05-OCT-01 00:00:00 |
| + | TIMESTAMP '2001-09-28 13:30:00' + 3 | 01-OCT-01 13:30:00 |
| - | DATE '2001-10-01' - 7 | 24-SEP-01 00:00:00 |
| - | TIMESTAMP '2001-09-28 13:30:00' - 3 | 25-SEP-01 13:30:00 |
| - | TIMESTAMP '2001-09-29 03:00:00' - TIMESTAMP '2001-09-27 12:00:00' | @ 1 day 15 hours |

In the date/time functions listed in Table 13-13: Date/Time functions, the use of the DATE and TIMESTAMP data types are interchangeable.

**Table 13-13: Date/Time functions**

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| ADD MONTHS( DATE, NUMBER) | DATE | Adds months to a date. | ADD MONTHS(' 28-FEB-97', 3.8) | 31-MAY-97 00:00:00 |
| CURRENT DATE | DATE | Returns the current date. | CURRENT DATE | 04-JUL-07 |
| CURRENT TIMESTAMP | TIMESTAMP | Returns the current date and time. | CURRENT TIMESTAMP | 04-JUL-07 15:33:23.484 |
| EXTRACT( field FROM TIMESTAMP) | DOUBLE PRECISION | Retrieves subfields. | EXTRACT( hour FROM TIMESTAMP ' 2001-02-16 20:38:40') | 20 |
| LAST DAY(DATE) | DATE | Returns the last day of the month represented by the given date. If the given date contains a time portion, the time portion is carried forward to the result unchanged. | LAST DAY('14-APR-98') | 30-APR-98 00:00:00 |
| LOCALTIMESTAMP [ (precision) ] | TIMESTAMP | Returns the current date and time (start of current transaction). | LOCALTIMESTAMP | 04-JUL-07 15:33:23.484 |
| MONTHS BETWEEN(DATE, DATE) | NUMBER | Returns the number of months between two dates. | MONTHS BETWEEN('28-FEB- 07', '30-N0V-06') | 3 |
| NEXT DAY(DATE, dayofweek) | DATE | Returns the date that falls on dayofweek following the specified date. | NEXT DAY('16-APR- 07','FRI') | 2 0-APR-07 00:00:00 |

| Function | Return type | Description | Example | Result |
|---|---|---|---|---|
| NEW TIME(DATE , VARCHAR, VARCHAR) | DATE | Converts a date and time to an alternate time zone. | NEW TIME(T0 DATE '2005/05/29 01:45', 'AST', ' PST') | 2005/05/29 21:45:00 |
| ROUND(DATE [, format ]) | DATE | Returns the date rounded according to format. | R0UND(T0 DATE ('29-MAY- 05'),' M0N') | 01-JUN-05 00:00 :00 |
| SYS EXTRACT UTC(TIME STAMP WITH TIME ZONE ) | TIMESTAMP | TIMESTAMP | SYS EXTRACT UTC(CAST('24 - MAR-11 12:30: 00PM - 04:00' AS TIMESTAMP WITH TIME ZONE )) | 2 4-MAR-11 16:30:00 |
| SYSDATE | DATE | Returns the current date and time. | SYSDATE | 01-AUG-12 11:12:34 |
| SYSTIMESTAMP() | TIMESTAMP | Returns the current date and time. | SYSTIMESTAMP | 01-AUG-12 11:11:23.665 229 -07:00 |
| TRUNC(DATE [ format]) | DATE | Truncates according to format. | TRUNC(T0 DATE ('2 9-MAY- 05'), ' MON') | 01-MAY-05 00:00:00 |

## 13.8.2 ADD_MONTHS

The ADD_MONTHS function adds (or subtract if the second parameter is negative) the specified number of months to the given date. The resulting day of the month and the given date are the same. However, if the day of the month of the given date is the last day of the month, the resulting date always falls on the last day of the month.

**Note:**

- Any fractional part for the number of months parameter is truncated before calculation.
- If the given date contains a time portion, the time portion is carried forward without changing the result.

**Examples**

```
SELECT ADD_MONTHS('13-JUN-07',4) FROM DUAL;

    add_months
--------------------
 13-OCT-07 00:00:00
(1 row)

SELECT ADD_MONTHS('31-DEC-06',2) FROM DUAL;

    add_months
--------------------
 28-FEB-07 00:00:00
(1 row)


SELECT ADD_MONTHS('31-MAY-04',-3) FROM DUAL;

    add_months
--------------------
 29-FEB-04 00:00:00
(1 row)
```

## 13.8.3 EXTRACT

The EXTRACT function retrieves subfields such as year or hour from date/time values. This function returns a value of the data type DOUBLE PRECISION.

**YEAR**

The year field.

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
      2001
(1 row)
```

**MONTH**

The number of the month within the year (1-12).

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
         2
(1 row)
```

**DAY**

The day of the month (1-31).

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

```
 date_part
-----------
        16
(1 row)
```

**HOUR**

The hour of the day (0-23).

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
        20
(1 row)
```

**MINUTE**

The minute of the hour (0-59).

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
        38
(1 row)
```

**SECOND**

The second of the minute, including the fractional part (0-59).

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
        40
(1 row)
```

## 13.8.4 MONTHS_BETWEEN

The MONTHS_BETWEEN function returns the number of months between two dates. The result is a numeric value that is positive if the first date is later than the second date or negative if the first date is less than the second date.

The result is always a whole number of months if the day of the month for both date parameters is the same, or both date parameters fall on the last day of their respective months.

The following are some examples of the MONTHS_BETWEEN function:

```
SELECT MONTHS_BETWEEN('15-DEC-06','15-OCT-06') FROM DUAL;

 months_between
----------------
```

```
            2
(1 row)

SELECT MONTHS_BETWEEN('15-OCT-06','15-DEC-06') FROM DUAL;

 months_between
----------------
           -2
(1 row)

SELECT MONTHS_BETWEEN('31-JUL-00','01-JUL-00') FROM DUAL;

 months_between
----------------
    0.967741935
(1 row)

SELECT MONTHS_BETWEEN('01-JAN-07','01-JAN-06') FROM DUAL;

 months_between
----------------
           12
(1 row)
```

## 13.8.5 NEXT_DAY

The NEXT_DAY function returns the date of the first occurrence of the given day that is strictly later than the given date. You must specify at least the first three letters of the day , for example, SAT. If the given date contains a time portion, the time portion is carried forward without changing the result.

The following is an example of the NEXT_DAY function:

```
SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'SUNDAY') FROM DUAL;

      next_day
--------------------
 19-AUG-07 00:00:00
(1 row)

SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'MON') FROM DUAL;

      next_day
--------------------
 20-AUG-07 00:00:00
```

```
(1 row)
```

## 13.8.6 NEW_TIME

The NEW_TIME function converts a date and time from one time zone to another. This function returns a value of the data type DATE. The syntax is:

```
NEW_TIME(DATE, time_zone1, time_zone2)
```

The time_zone1 and time_zone2 parameters must be string values from the time zone column in the following table:

| Time zone | Offset from UTC | Description |
|---|---|---|
| AST | UTC+4 | Atlantic Standard Time |
| ADT | UTC+3 | Atlantic Daylight Time |
| BST | UTC+11 | Bering Standard Time |
| BDT | UTC+10 | Bering Daylight Time |
| CST | UTC+6 | Central Standard Time |
| CDT | UTC+5 | Central Daylight Time |
| EST | UTC+5 | Eastern Standard Time |
| EDT | UTC+4 | Eastern Daylight Time |
| GMT | UTC | Greenwich Mean Time |
| HST | UTC+10 | Alaska-Hawaii Standard Time |
| HDT | UTC+9 | Alaska-Hawaii Daylight Time |
| MST | UTC+7 | Mountain Standard Time |
| MDT | UTC+6 | Mountain Daylight Time |
| NST | UTC+3:30 | Newfoundland Standard Time |
| PST | UTC+8 | Pacific Standard Time |
| PDT | UTC+7 | Pacific Daylight Time |
| YST | UTC+9 | Yukon Standard Time |
| YDT | UTC+8 | Yukon Daylight Time |

The following is an example of the NEW_TIME function:

```
SELECT NEW_TIME(TO_DATE('08-13-07 10:35:15','MM-DD-YY HH24:MI:SS'),'AST', 'PST') "
Pacific Standard Time" FROM DUAL;
Pacific Standard Time
---------------------
 13-AUG-07 06:35:15
(1 row)
```

## 13.8.7 ROUND

The ROUND function returns a date rounded according to a specified template pattern. If the template pattern is omitted, the date is rounded to the nearest day. The following table shows the template patterns that can be used for the ROUND function.

**Table 13-14: Template date patterns for the ROUND function**

| Pattern | Description |
|---|---|
| CC, SCC | Returns January 1, cc01 where cc is the first 2 digits of the given year if the last 2 digits are at most 50, or 1 greater than the first 2 digits of the given year if the last 2 digits are greater than 50. |
| SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y | Returns January 1, yyyy where yyyy is rounded to the nearest year. The date rounds down on June 30, and rounds up on July 1. |
| IYYY, IYY, IY, I | Rounds the date to the beginning of the ISO year, which is determined by rounding down if the month and day is on or before June 30. The date rounds up if the month and day is July 1 or later. |
| Q | Returns the first day of the quarter, which is determined by rounding down if the month and day is on or before the 15th day of the second month of the quarter. Otherwise, the date is rounded up if the month and day is the 16th day of the second month or later of the quarter. |
| MONTH, MON, MM, RM | Returns the first day of the specified month if the day of the month is on or before the 15th day. Returns the first day of the following month if the day of the month is the 16th day or later. |

| Pattern | Description |
|---|---|
| WW | Rounds the date to the nearest date that corresponds to the same day of the week as the first day of the year. |
| IW | Rounds the date to the nearest date that corresponds to the same day of the week as the first day of the ISO year. |
| W | Rounds the date to the nearest date that corresponds to the same day of the week as the first day of the month. |
| DDD, DD, J | Rounds the date to the start of the nearest day. Rounds to the start of the same day if the specified time is 11:59:59 AM or earlier . Rounds to the start of the next day if the specified time is 12:00:00 PM or later. |
| DAY, DY, D | Rounds the date to the nearest Sunday. |
| HH, HH12, HH24 | Rounds the date to the nearest hour. |
| MI | Rounds the date to the nearest minute. |

The following section provides ROUND function examples.

The following examples round the date to the nearest century.

```
SELECT TO_CHAR(ROUND(TO_DATE('1950','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM
DUAL;

   Century
-------------
 01-JAN-1901
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM
DUAL;

   Century
-------------
 01-JAN-2001
(1 row)
```

The following examples round the date to the nearest year.

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "
Year" FROM DUAL;

    Year
-------------
 01-JAN-1999
(1 row)
```

```
SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "
Year" FROM DUAL;

    Year
-------------
 01-JAN-2000
(1 row)
```

The following examples round the date to the nearest ISO year. The first example rounds to 2004. The ISO year for 2004 begins on December 29, 2003. The second example rounds the date to 2005. The ISO year for 2005 begins on January 3 of that same year.

**Note:**

An ISO year begins on the first Monday from which a seven day span (Monday to Sunday) contains at least 4 days of the new year. Therefore, the beginning of an ISO year can start in December of the previous year.

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY')
 "ISO Year" FROM DUAL;

  ISO Year
-------------
 29-DEC-2003
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY')
 "ISO Year" FROM DUAL;

  ISO Year
-------------
 03-JAN-2005
(1 row)
```

The following examples round the date to the nearest quarter.

```
SELECT ROUND(TO_DATE('15-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

      Quarter
--------------------
 01-JAN-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

      Quarter
--------------------
 01-APR-07 00:00:00
(1 row)
```

The following examples round the date to the nearest month.

```
SELECT ROUND(TO_DATE('15-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

       Month
--------------------
```

```
 01-DEC-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

       Month
--------------------
 01-JAN-08 00:00:00
(1 row)
```

The following examples round the date to the nearest week. The first day of 2007 is a Monday. Therefore, in the first example, the Monday that is closest to January 18 is January 15. In the second example, the Monday that is closest to January 19 is January 22.

```
SELECT ROUND(TO_DATE('18-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

        Week
--------------------
 15-JAN-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

        Week
--------------------
 22-JAN-07 00:00:00
(1 row)
```

The following examples round the date to the nearest ISO week. An ISO week starts on a Monday. In the first example, the Monday that is closest to January 1, 2004 is December 29 , 2003. In the second example, the Monday that is closest to January 2, 2004 is January 5, 2004.

```
SELECT ROUND(TO_DATE('01-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

      ISO Week
--------------------
 29-DEC-03 00:00:00
(1 row)

SELECT TRUNC(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

      ISO Week
--------------------
 05-JAN-04 00:00:00
(1 row)
```

The following examples round the date to the nearest week where a week is considered to start on the same day as the first day of the month.

```
SELECT ROUND(TO_DATE('05-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

        Week
--------------------
 08-MAR-07 00:00:00
(1 row)
```

```
SELECT ROUND(TO_DATE('04-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

        Week
--------------------
 01-MAR-07 00:00:00
(1 row)
```

The following examples round the date to the nearest day.

```
SELECT ROUND(TO_DATE('04-AUG-07 11:59:59 AM','DD-MON-YY HH:MI:SS AM'),'J') "Day"
FROM DUAL;

        Day
--------------------
 04-AUG-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J') "Day"
FROM DUAL;

        Day
--------------------
 05-AUG-07 00:00:00
(1 row)
```

The following examples round the date to the nearest Sunday.

```
SELECT ROUND(TO_DATE('08-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

    Day of Week
--------------------
 05-AUG-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

    Day of Week
--------------------
 12-AUG-07 00:00:00
(1 row)
```

The following examples round the date to the nearest hour.

```
SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:29','DD-MON-YY HH:MI'),'HH'),'DD-MON
-YY HH24:MI:SS') "Hour" FROM DUAL;

        Hour
--------------------
 09-AUG-07 08:00:00
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-MON
-YY HH24:MI:SS') "Hour" FROM DUAL;

        Hour
--------------------
 09-AUG-07 09:00:00
```

```
(1 row)
```

The following examples round the date to the nearest minute.

```
SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30:29','DD-MON-YY HH:MI:SS'),'MI'),'DD
-MON-YY HH24:MI:SS') "Minute" FROM DUAL;

      Minute
--------------------
 09-AUG-07 08:30:00
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30:30','DD-MON-YY HH:MI:SS'),'MI'),'DD
-MON-YY HH24:MI:SS') "Minute" FROM DUAL;

      Minute
--------------------
 09-AUG-07 08:31:00
(1 row)
```

## 13.8.8 TRUNC

The TRUNC function returns a date that is truncated according to a specified template pattern. If the template pattern is omitted, the date is truncated to the nearest day. The following table shows the template patterns that can be used for the TRUNC function.

**Table 13-15: Template date patterns for the TRUNC function**

| Pattern | Description |
|---|---|
| CC, SCC | Returns January 1, cc01 where cc is the first 2 digits of the given year. |
| SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y | Returns January 1, yyyy where yyyy is the given year. |
| IYYY, IYY, IY, I | Returns the start date of the ISO year containing the given date. |
| Q | Returns the first day of the quarter containing the given date. |
| MONTH, MON, MM, RM | Returns the first day of the specified month. |
| WW | Returns the largest date just prior to, or the same as the given date that corresponds to the same day of the week as the first day of the year. |
| IW | Returns the start of the ISO week containing the given date. |

| Pattern | Description |
|---|---|
| W | Returns the largest date just prior to, or the same as the given date that corresponds to the same day of the week as the first day of the month. |
| DDD, DD, J | Returns the start of the day for the given date. |
| DAY, DY, D | Returns the start of the week (Sunday) containing the given date. |
| HH, HH12, HH24 | Returns the start of the hour. |
| MI | Returns the start of the minute. |

**Examples**

The following example truncates the date to the hundred years unit.

```
SELECT TO_CHAR(TRUNC(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM
DUAL;

   Century
-------------
 01-JAN-1901
(1 row)
```

The following example truncates the date to the year.

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "
Year" FROM DUAL;

    Year
-------------
 01-JAN-1999
(1 row)
```

The following example truncates the date to the beginning of the ISO year.

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "
ISO Year" FROM DUAL;

  ISO Year
-------------
 29-DEC-2003
(1 row)
```

The following example truncates the date to the start date of the quarter.

```
SELECT TRUNC(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

      Quarter
--------------------
 01-JAN-07 00:00:00
```

```
(1 row)
```

The following example truncates the date to the start date of the month.

```
SELECT TRUNC(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

       Month
--------------------
 01-DEC-07 00:00:00
(1 row)
```

The following example truncates the date to the start of the week determined by the first day of the year. For example, the first day of 2007 is a Monday, so the first Monday before January 19 is January 15.

```
SELECT TRUNC(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

        Week
--------------------
 15-JAN-07 00:00:00
(1 row)
```

The following example truncates the date to the beginning of an ISO week. An ISO week starts on a Monday. January 2, 2004 is within the ISO week that starts on Monday, December 29, 2003.

```
SELECT TRUNC(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

      ISO Week
--------------------
 29-DEC-03 00:00:00
(1 row)
```

The following example truncates the date to the start of the week where a week is considered to start on the same day as the first day of the month.

```
SELECT TRUNC(TO_DATE('21-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

        Week
--------------------
 15-MAR-07 00:00:00
(1 row)
```

The following example truncates the date to the start of the day.

```
SELECT TRUNC(TO_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J') "Day"
FROM DUAL;

        Day
--------------------
 04-AUG-07 00:00:00
```

```
(1 row)
```

The following example truncates the date to the start of the week (Sunday).

```
SELECT TRUNC(TO_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

    Day of Week
--------------------
 05-AUG-07 00:00:00
(1 row)
```

The following example truncates the date to the start of the hour.

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-MON-
YY HH24:MI:SS') "Hour" FROM DUAL;

        Hour
--------------------
 09-AUG-07 08:00:00
(1 row)
```

The following example truncates the date to the start of the minute.

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30:30','DD-MON-YY HH:MI:SS'),'MI'),'DD-
MON-YY HH24:MI:SS') "Minute" FROM DUAL;

       Minute
--------------------
 09-AUG-07 08:30:00
(1 row)
```

## 13.8.9 CURRENT DATE/TIME

POLARDB compatible with Oracle provides many functions that return values related to the current date and time. All these functions return values based on the start time of the current transaction.

- CURRENT_DATE
- CURRENT_TIMESTAMP
- LOCALTIMESTAMP
- LOCALTIMESTAMP(precision)

The CURRENT_DATE function returns the current date and time based on the start time of the current transaction. If CURRENT_DATE is called multiple times within a transaction, the value of CURRENT_DATE will not change.

```
SELECT CURRENT_DATE FROM DUAL;

   date
-----------
```

```
 06-AUG-07
```

The CURRENT_TIMESTAMP function returns the current date and time. When called from an SQL statement, this function returns the same value for each occurrence within the statement. If called from multiple statements within a transaction, this function may return different values for each occurrence. If called from a function, this function may return a different value other than the value returned by current_timestamp in the caller.

```
SELECT CURRENT_TIMESTAMP, CURRENT_TIMESTAMP FROM DUAL;

          current_timestamp         |          current_timestamp
----------------------------------+----------------------------------
 02-SEP-13 17:52:29.261473 +05:00 | 02-SEP-13 17:52:29.261474 +05:00
```

The LOCALTIMESTAMP function can optionally be assigned a precision parameter. This parameter causes the result to be rounded to that many fractional digits in the seconds field. If no precision parameter is assigned, the result is given to the full available precision.

```
SELECT LOCALTIMESTAMP FROM DUAL;

       timestamp
------------------------
 06-AUG-07 16:11:35.973
(1 row)

SELECT LOCALTIMESTAMP(2) FROM DUAL;

       timestamp
-----------------------
 06-AUG-07 16:11:44.58
(1 row)
```

The preceding functions return the start time of the current transaction. Their values do not change during the transaction. This is considered a feature: The intent is to allow a single transaction to have a consistent notion of the "current" time. Therefore, multiple modifications within the same transaction bear the same timestamp. Other database systems can more frequently use these values.

## 13.9 Sequence manipulation functions

This topic describes the functions that POLARDB compatible with Oracle provides to manage sequence objects.

A sequence object (also known as a sequence generator or sequence) is a specific single-row table created by the CREATE SEQUENCE command. A sequence object is used to generate unique identifiers for rows of a table. The following sequence functions provide simple and multiuser-safe methods for obtaining successive sequence values from sequence objects.

### Syntaxes

```
sequence.NEXTVAL
sequence.CURRVAL
```

### Parameters

- sequence: the identifier assigned to the sequence in the CREATE SEQUENCE command. The following section describes how to use the preceding functions.
- NEXTVAL: This function advances the sequence object to its next value and returns the value. This operation cannot be reversed after it is complete. If multiple sessions concurrently run the NEXTVAL function, each session will safely receive a distinct sequence value.
- CURRVAL: This function returns the value that is most recently obtained by the NEXTVAL function for the specified sequence in the current session. (If NEXTVAL has never been called for this sequence in this session, an error is reported.) Note that this function returns a session-local value. It provides a predictable answer to whether other sessions have run NEXTVAL since the current session ran NEXTVAL.

If a sequence object has been created by using default parameters, calls to the NEXTVAL function on this object will return successive values starting with 1. You can use specific parameters in the CREATE SEQUENCE command to obtain other behavior.

**Note:**

To avoid blocking concurrent transactions that obtain numbers from the same sequence, you cannot roll back a NEXTVAL operation. Once a value has been retrieved, it is considered used. The value is still considered used if the transaction that did the NEXTVAL later aborts. This means that aborted transactions may leave unused "gaps" in the sequence of assigned values.

## 13.10 Conditional expressions

The following section describes the SQL-compliant conditional expressions available in POLARDB compatible with Oracle.

### CASE

The CASE expression in SQL is a generic condition expression, similar to the if/else statements in other programming languages:

```
CASE WHEN condition THEN result
   [ WHEN ... ]
   [ ELSE result ]
END
```

CASE clauses can be used wherever an expression is valid. condition is an expression that returns a BOOLEAN result. If the result is TRUE, the value of the CASE expression is the result that follows the condition. If the result is FALSE, any subsequent WHEN clauses are searched in the same manner. If no WHEN condition is TRUE, the value of the CASE expression is the result in the ELSE clause. If the ELSE clause is omitted and no condition matches, the result is NULL.

```
SELECT * FROM test;

 a
---
 1
 2
 3
(3 rows)

SELECT a,
    CASE WHEN a=1 THEN 'one'
         WHEN a=2 THEN 'two'
         ELSE 'other'
    END
FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
(3 rows)
```

The data types of all the result expressions must support conversion into a single output type.

The following "simple" CASE expression is a specialized variant of the general form above:

```
CASE expression
    WHEN value THEN result
```

```
  [ WHEN ... ]
  [ ELSE result ]
END
```

The expression is computed and compared to all the value specifications in the WHEN clauses until a match is found. If no match is found, the result in the ELSE clause (or a null value) is returned.

The preceding example can be written using the simple CASE syntax:

```
SELECT a,
    CASE a WHEN 1 THEN 'one'
           WHEN 2 THEN 'two'
           ELSE 'other'
    END
FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
(3 rows)
```

A CASE expression does not evaluate any subexpressions that are not used to determine the result. For example, you can avoid a division-by-zero failure by using the following method:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

### COALESCE

The COALESCE function returns the first of its arguments that is not null. Null is only returned when all arguments are null.

```
COALESCE(value [, value2 ] ... )
```

This function is often used to substitute a default value for null values when data is retrieved for display or further computation. For example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

Sam as a CASE expression, COALESCE does not evaluate any arguments that are not used to determine the result. Arguments to the right of the first non-null argument are not evaluated. This SQL-standard function provides capabilities similar to NVL and IFNULL, which can be used in some other database systems.

### NULLIF

The NULLIF function returns a null value if value1 and value2 are equal. If the values are not equal, value1 is returned.

```
NULLIF(valuel, value2)
```

This function can be used to perform the inverse operation of the preceding COALESCE example:

```
SELECT NULLIF(value1, '(none)') ...
```

If value1 is (none), a null value is returned. Otherwise, value1 is returned.

### NVL

The NVL function returns the first of its arguments that is not null. This function evaluates the first expression. If that expression is evaluated to null, NVL returns the second expression.

```
NVL(exprl, expr2)
```

The return type is the same as the argument type. All arguments must be of the same data type (or must support conversion into a common data type). If all arguments are null, NVL returns null.

The following example calculates a bonus for employees who have no commission. If an employee receives commission, this expression returns the commission of the employee. If the employee does not receive commission (the commission is null), this expression returns a bonus that is equal to 10% of the employee's salary.

```
bonus = NVL(emp.commission, emp.salary * .10)
```

### NVL2

NVL2 evaluates an expression and returns the second or third expression, depending on the value of the first expression. If the first expression is not null, NVL2 returns the value in expr2. If the first expression is null, NVL2 returns the value in expr3.

```
NVL2(expr1, expr2, expr3)
```

The return type is the same as the argument type. All arguments must be of the same data type (or must support conversion into a common data type).

The following example calculates a bonus for employees who receive commission. If an employee receives commission, this expression returns an amount that is equal to 110% of the employee's commission. If the employee does not receive commission (the commission is null), this expression returns 0.

```
bonus = NVL2(emp.commission, emp.commission * 1-1, 0)
```

### GREATEST and LEAST

The GREATEST and LEAST functions select the highest or lowest value from a list of any number of expressions.

```
GREATEST(value [, value2 ] ... )
LEAST(value [, value2 ] ... )
```

The expressions must support conversion into a common data type, which will be the data type of the result. Null values in the list are ignored. The result is null only if all expressions are evaluated to null.

Note that the GREATEST and LEAST functions are not in the SQL standard, but are a common extension.

## 13.11 Aggregate functions

Aggregate functions compute a single result value from a set of input values. The following tables list the built-in aggregate functions.

**Table 13-16: General-purpose aggregate functions**

| Function | Argument type | Return type | Description |
|---|---|---|---|
| AVG(expression) | INTEGER, REAL, DOUBLE PRECISION, or NUMBER | NUMBER for any integer-type argument, DOUBLE PRECISION for a floating-point argument, otherwise the same as the argument data type | Returns the average ( arithmetic mean) of all input values. |
| COUNT(*) | | BIGINT | Returns the number of input rows. |

| Function | Argument type | Return type | Description |
|---|---|---|---|
| COUNT(expression) | Any | BIGINT | Returns the number of input rows for which the value of expression is not null . |
| MAX(expression) | Any numeric, string, or date/time type | Same as argument type | Returns the maximum value of expression across all input values. |
| MIN(expression) | Any numeric, string, or date/time type | Same as argument type | Returns the minimum value of expression across all input values. |
| SUM(expression) | INTEGER, REAL, DOUBLE PRECISION, or NUMBER | BIGINT for SMALLINT or INTEGER arguments, NUMBER for BIGINT arguments , DOUBLE PRECISION for floating-point arguments, otherwise the same as the argument data type | Returns the sum of expression across all input values. |

Note that except for the COUNT function, these functions return a null value when no rows are selected. In particular, SUM of no rows returns null, instead of returning 0 as expected. When necessary, you can use the COALESCE function to substitute zero for null.

The following table shows aggregate functions that are used in statistical analysis. ( These functions are separated out to avoid cluttering the listing of more-commonly-used aggregates.) N mentioned in any description indicates the number of input rows for which all the input expressions are not null. In all cases, null is returned if the computation is invalid (for example, when N is 0).

**Table 13-17: Aggregate functions for statistics**

| Function | Argument type | Return type | Description |
|---|---|---|---|
| CORR( Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | The correlation coefficient. |

| Function | Argument type | Return type | Description |
|---|---|---|---|
| COVAR POP( Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | The population covariance. |
| COVAR SAMP( Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | The sample covariance. |
| REGR AVGX( Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | The average of the independent variable (sum(X) / N ). |
| REGR AVGY(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | The average of the dependent variable ( sum(Y) / N). |
| REGR COUNT( Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | The number of input rows in which both expressions are not null. |
| REGR INTERCEPT(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | The y-intercept of the least-squares-fit linear equation determined by the (X , Y) pairs. |
| REGR R2( Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | The square of the correlation coefficien t. |
| REGR SLOPE( Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | The slope of the least -squares-fit linear equation determined by the (X, Y) pairs. |
| REGR SXX(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Sum (X2) - sum (X)2 / N ("sum of squares " of the independent variable) |
| REGR SXY(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Sum (X* Y) - sum ( X) * sum( Y) / N (" sum of products" of independent times dependent variable) |

| Function | Argument type | Return type | Description |
|---|---|---|---|
| REGR SYY( Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Sum (Y2) - sum (Y)2 / N ("sum of squares " of the dependent variable) |
| STDDEV(expression) | INTEGER, REAL, DOUBLE PRECISION, or NUMBER | DOUBLE PRECISION for floating-point arguments, otherwise NUMBER | The historical alias for STDDEV SAMP. |
| STDDEV POP( expression) | INTEGER, REAL, DOUBLE PRECISION, or NUMBER | DOUBLE PRECISION for floating-point arguments, otherwise NUMBER | The population standard deviation of the input values. |
| STDDEV SAMP( expression) | INTEGER, REAL, DOUBLE PRECISION, or NUMBER | DOUBLE PRECISION for floating-point arguments, otherwise NUMBER | The sample standard deviation of the input values. |
| VARIANCE(expression ) | INTEGER, REAL, DOUBLE PRECISION, or NUMBER | DOUBLE PRECISION for floating-point arguments, otherwise NUMBER | The historical alias for VAR SAMP. |
| VAR POP(expression) | INTEGER, REAL, DOUBLE PRECISION, or NUMBER | DOUBLE PRECISION for floating-point arguments, otherwise NUMBER | The population variance of the input values (square of the population standard deviation). |
| VAR SAMP( expression) | INTEGER, REAL, DOUBLE PRECISION, or NUMBER | DOUBLE PRECISION for floating-point arguments, otherwise NUMBER | The sample variance of the input values ( square of the sample standard deviation). |

## 13.12 Subquery expressions

This topic describes the SQL-compliant subquery expressions available in POLARDB compatible with Oracle. All expressions described in this topic return Boolean (true/false) results.

### EXISTS

The argument of EXISTS is an arbitrary SELECT statement or subquery. The subquery is evaluated to determine whether it returns rows. If at least one row is returned, the result of EXISTS is "true". If the subquery returns no rows, the result of EXISTS is "false".

```
EXISTS(subquery)
```

The subquery can refer to variables from the surrounding query, which will act as constants during an evaluation of the subquery.

In most cases, the time required for the subquery to run will only be enough to determine whether a minimum of one row is returned (not until completion). We do not recommend that you write a subquery that produces any potential side effects (such as calling sequence functions). It is difficult to predict when and if potential side effects may occur.

The result of EXISTS only depends on whether rows are returned, rather than on the content of the rows. Therefore, the output list of the subquery can be ignored. A common coding convention is to write all EXISTS tests in the form of EXISTS (SELECT 1 WHERE...). However, exceptions to this rule exist, such as subqueries that use INTERSECT.

This example is similar to an inner join on the deptno column. However, in this example, up to one output row is produced for each dept row and multiple matching emp rows:

```
SELECT dname FROM dept WHERE EXISTS (SELECT 1 FROM emp WHERE emp.deptno = dept
.deptno);

   dname
------------
 ACCOUNTING
 RESEARCH
 SALES
(3 rows)
```

### IN

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result.

The result of IN is "true" if one equal subquery row is found. The result is "false" if no equal row is found (including the case where the subquery returns no rows).

```
expression IN (subquery)
```

Note that the result of the IN construct will be null rather than "false" in either of the following scenarios. 1. The left-hand expression returns null. 2. No equal right-hand values are found and at least one right-hand row returns null. This is in accordance with standard SQL rules for Boolean combinations of null values.

As with EXISTS, we do not recommend you assume that a complete evaluation of the subquery will be performed.

**NOT IN**

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of NOT IN is "true" if only unequal subquery rows are found (including the case where the subquery returns no rows). The result is "false" if an equal row is found.

```
expression NOT IN (subquery)
```

Note that the result of the NOT IN construct will be null rather than "true" in either of the following scenarios. 1. The left-hand expression returns null. 2. No equal right-hand values are found and at least one right-hand row returns null. This is in accordance with standard SQL rules for Boolean combinations of null values.

As with EXISTS, we do not recommend you assume that a complete evaluation of the subquery will be performed.

**ANY/SOME**

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result by using the given operator, which must generate a BOOLEAN result. The result of ANY is "true " if a true result is returned. The result is "false" if no true result is found (including the case where the subquery returns no rows).

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

SOME is a synonym for ANY. IN is equivalent to "= ANY".

Note that if no success is achieved and at least one right-hand row returns null for the operator's result, the result of the ANY construct will be null, not "false". This is in accordance with standard SQL rules for Boolean combinations of null values.

As with EXISTS, we do not recommend you assume that a complete evaluation of the subquery will be performed.

**ALL**

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result by using the given operator, which must generate a BOOLEAN result. The result of ALL is "true" if the comparison returns true for all subquery rows (including the case where the subquery returns no rows). The result is "false" if the comparison returns false for a subquery row. The result is null if the comparison does not return false for any subquery row, and the comparison returns null for at least one row.

```
expression operator ALL (subquery)
```

NOT IN is equivalent to "<> ALL".

As with EXISTS, we do not recommend you assume that a complete evaluation of the subquery will be performed.

# 14 Oracle catalog views

## 14.1 ALL_ALL_TABLES

The ALL_ALL_TABLES view provides the information about the tables accessible by the current user.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the table belongs. |
| schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table. |
| tablespace_name | TEXT | The name of the tablespace where the table is located if this tablespace is not the default tablespace. |
| status | CHARACTER VARYING (5) | This parameter is supported for compatibility only. The value is VALID. |
| temporary | TEXT | • Y: indicates that the table is a temporary table.<br>• N: indicates that the table is a permanent table. |

## 14.2 ALL_CONS_COLUMNS

The ALL_CONS_COLUMNS view provides the information about the columns specified in constraints placed on tables accessible by the current user.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the constraint belongs. |
| schema_name | TEXT | The name of the schema to which the constraint belongs. |
| constraint_name | TEXT | The name of the constraint. |
| table_name | TEXT | The name of the table to which the constraint belongs. |

| Parameter | Type | Description |
|---|---|---|
| column_name | TEXT | The name of the column referenced in the constraint. |
| position | SMALLINT | The position of the column within the object definition. |
| constraint_def | TEXT | The definition of the constraint. |

## 14.3 ALL_CONSTRAINTS

The ALL_CONSTRAINTS view provides the information about the constraints placed on tables accessible by the current user.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the constraint belongs. |
| schema_name | TEXT | The name of the schema to which the constraint belongs. |
| constraint_name | TEXT | The name of the constraint. |
| constraint_type | TEXT | The type of the constraint. Valid values:<br>• C: the check constraint<br>• F: the foreign key constraint<br>• P: the primary key constraint<br>• U: the unique key constraint<br>• R: the referential integrity constraint<br>• V: the constraint on a view<br>• O: with a read-only property on a view |
| table_name | TEXT | The name of the table to which the constraint belongs. |
| search_condition | TEXT | The search condition that applies to a check constraint. |
| r_owner | TEXT | The owner of a table referenced by a referential constraint. |
| r_constraint_name | TEXT | The name of the constraint definition for a referenced table. |

| Parameter | Type | Description |
|---|---|---|
| delete_rule | TEXT | The delete rule for a referential constraint. Valid values:<br>• C: cascade<br>• R: restrict<br>• N: no action |
| deferrable | BOOLEAN | Indicates if the constraint is deferrable. Valid values: T and F. |
| deferred | BOOLEAN | Indicates if the constraint has been deferred . Valid values: T and F. |
| index_owner | TEXT | The username of the owner to which the index belongs. |
| index_name | TEXT | The name of the index. |
| constraint_def | TEXT | The definition of the constraint. |

## 14.4 ALL_DB_LINKS

The ALL_DB_LINKS view provides the information about the database links accessible by the current user.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the database link belongs. |
| db_link | TEXT | The name of the database link. |
| type | CHARACTER VARYING | The type of the remote server. Valid values: REDWOOD, EDB. |
| username | TEXT | The username of the user logging in. |
| host | TEXT | The name or IP address of the remote server . |

## 14.5 ALL_DIRECTORIES

The ALL_DIRECTORIES view provides the information about all directories created by the CREATE DIRECTORY command.

| Parameter | Type | Description |
|---|---|---|
| owner | CHARACTER VARYING(30) | The username of the owner to which the directory belongs. |
| Directory_name | CHARACTER VARYING(30) | The alias name assigned to the directory. |
| directory_path | CHARACTER VARYING(4000) | The path to the directory. |

## 14.6 ALL_IND_COLUMNS

The ALL_IND_COLUMNS view provides the information about columns included in indexes on the tables accessible by the current user.

| Parameter | Type | Description |
|---|---|---|
| index_owner | TEXT | The username of the owner to which the index belongs. |
| schema_name | TEXT | The name of the schema to which the index belongs. |
| index_name | TEXT | The name of the index. |
| table_owner | TEXT | The username of the owner to which the table belongs. |
| table_name | TEXT | The name of the table to which the index belongs. |
| column_name | TEXT | The name of the column. |
| column_position | SMALLINT | The position of the column within the index. |
| column_length | SMALLINT | The length of the column in bytes. |
| char_length | NUMERIC | The length of a column in characters. |
| descend | CHARACTER(1) | This parameter is supported for compatibil ity only. The value is Y in descending order. |

## 14.7 ALL_INDEXES

The ALL_INDEXES view provides the information about the indexes on tables that may be accessed by the current user.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the index belongs. |
| schema_name | TEXT | The name of the schema to which the index belongs. |
| index_name | TEXT | The name of the index. |
| index_type | TEXT | The index type is BTREE. This parameter is supported for compatibility only. |
| table_owner | TEXT | The username of the owner of the indexed table. |
| table_name | TEXT | The name of the indexed table. |
| table_type | TEXT | This parameter is supported for compatibility only. The value is TABLE. |
| uniqueness | TEXT | Indicates if the index is UNIQUE or NONUNIQUE. |
| compression | CHARACTER(1) | The value is N (not compressed). This parameter is supported for compatibility only. |
| tablespace_name | TEXT | The name of the tablespace where the table is located if this tablespace is not the default tablespace. |
| logging | TEXT | The value is LOGGING. This parameter is supported for compatibility only. |
| status | TEXT | This parameter is supported for compatibility only. The value is VALID. |
| partitioned | CHARACTER(3) | Indicates that the index is partitioned. The value is NO. |
| temporary | CHARACTER(1) | Indicates that an index is on a temporary table. This parameter is supported for compatibility only. The value is N. |
| secondary | CHARACTER(1) | This parameter is supported for compatibility only. The value is N. |

| Parameter | Type | Description |
|---|---|---|
| join_index | CHARACTER(3) | This parameter is supported for compatibility only. The value is NO. |
| dropped | CHARACTER(3) | This parameter is supported for compatibility only. The value is NO. |

## 14.8 ALL_JOBS

The ALL_JOBS view provides the information about all jobs in a database.

| Parameter | Type | Description |
|---|---|---|
| job | INTEGER | The identifier of a job (Job ID). |
| log_user | TEXT | The name of the user that submitted the job . |
| priv_user | TEXT | The same as log_user. This parameter is supported for compatibility only. |
| schema_user | TEXT | The name of the schema used to parse the job. |
| last_date | TIMESTAMP WITH TIME ZONE | The last date when this job was executed with the expected result returned. |
| last_sec | TEXT | The same as last_date. |
| this_date | TIMESTAMP WITH TIME ZONE | The date when the system starts to execute the job. |
| this_sec | TEXT | The same as this_date. |
| next_date | TIMESTAMP WITH TIME ZONE | The next date when this job will be executed . |
| next_sec | TEXT | The same as next_date. |
| total_time | INTERVAL | The period in which the job is executed. Unit : seconds. |
| broken | TEXT | • Y: indicates that no attempt will be made to run this job.<br>• N: indicates that attempts will be made to run this job. |
| interval | TEXT | The interval at which the job is repeated. |
| failures | BIGINT | The number of times that the job has failed since the last successful execution. |

| Parameter | Type | Description |
|---|---|---|
| what | TEXT | The job definition that runs when the job executes. The job definition appears as a PL /SQL code block. |
| nls_env | CHARACTER VARYING (4000) | The value is NULL. This parameter is supported for compatibility only. |
| misc_env | BYTEA | The value is NULL. This parameter is supported for compatibility only. |
| instance | NUMERIC | The value is 0. This parameter is supported for compatibility only. |

## 14.9 ALL_OBJECTS

The ALL_OBJECTS view provides the information about all objects in a database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which an object belongs. |
| schema_name | TEXT | The name of the schema to which the object belongs. |
| object_name | TEXT | The name of the object. |
| object_type | TEXT | The type of the object. Valid values: INDEX , FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW. |
| status | CHARACTER VARYING | Indicates whether the state of the object is valid. This parameter is supported for compatibility only. The value is VALID. |
| temporary | TEXT | • Y: The object is a temporary object.<br>• N: The object is a permanent object. |

## 14.10 ALL_PART_KEY_COLUMNS

The ALL_PART_KEY_COLUMNS view provides the information about the key columns of partitioned tables in a database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The owner of a partitioned table. |
| schema_name | TEXT | The name of the schema to which the table belongs. |
| name | TEXT | The name of the table to which the column belongs. |
| object_type | CHARACTER(5) | This parameter is supported for compatibility only. The value is TABLE. |
| column_name | TEXT | The name of the column on which the key is defined. |
| column_position | INTEGER | The ordinal position of this column. For example, a value of 1 indicates the first column and a value of 2 indicates the second column . All columns follow the same rule. |

## 14.11 ALL_PART_TABLES

The ALL_PART_TABLES view provides the information about all partitioned tables in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The owner of a partitioned table. |
| schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table. |
| partitioning_type | TEXT | The partition type used to define table partitions. |

| Parameter | Type | Description |
|---|---|---|
| subpartitioning_type | TEXT | The subpartition type used to define table subpartitions. |
| partition_count | BIGINT | The number of partitions in the table. |
| def_subpartition_count | INTEGER | The number of subpartitions in the table. |
| partitioning_key_count | INTEGER | The number of specified partition keys. |
| subpartitioning_key_count | INTEGER | The number of specified subpartition keys. |
| status | CHARACTER VARYING(8) | This parameter is supported for compatibility only. The value is VALID. |
| def_tablespace_name | CHARACTER VARYING(30) | This parameter is supported for compatibility only. The value is NULL. |
| def_pct_free | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_pct_used | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_ini_trans | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_max_trans | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_initial_extent | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| Def_next_extent | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| def_min_extents | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |

| Parameter | Type | Description |
|---|---|---|
| def_max_extents | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| def_pct_increase | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| def_freelists | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_freelist_groups | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_logging | CHARACTER VARYING(7) | This parameter is supported for compatibility only. The value is YES. |
| def_compression | CHARACTER VARYING(8) | This parameter is supported for compatibility only. The value is NONE. |
| def_buffer_pool | CHARACTER VARYING(7) | This parameter is supported for compatibility only. The value is DEFAULT. |
| ref_ptn_constraint_name | CHARACTER VARYING(30) | This parameter is supported for compatibility only. The value is NULL. |
| interval | CHARACTER VARYING(1000) | This parameter is supported for compatibility only. The value is NULL. |

## 14.12 ALL_QUEUES

The ALL_QUEUES view provides the information about the queues that have been defined.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which a queue belongs. |
| Parameter | TEXT | The name of the queue. |
| queue_table | TEXT | The name of the queue table to which the queue belongs. |

| Parameter | Type | Description |
|---|---|---|
| qid | OID | The object ID that the system assigns to the queue. |
| queue_type | CHARACTER VARYING | The type of the queue. Valid values : EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, and NORMAL_QUEUE. |
| max_retries | NUMERIC | The maximum number of dequeuing attempts. |
| retrydelay | NUMERIC | The maximum time allowed between retries . |
| enqueue_enabled | CHARACTER VARYING | • YES: The queue allows enqueuing.<br>• NO: The queue does not allow enqueuing. |
| dequeue_enabled | CHARACTER VARYING | • YES: The queue allows dequeuing.<br>• NO: The queue does not allow dequeuing. |
| retention | CHARACTER VARYING | The number of seconds that a processed message is retained in the queue. |
| user_comment | CHARACTER VARYING | The user-defined comment. |
| network_name | CHARACTER VARYING | The name of the network in which the queue is located. |
| sharded | CHARACTER VARYING | • YES: indicates the queue is in a sharded network.<br>• NO: indicates the queue is not in a sharded network. |

## 14.13 ALL_QUEUE_TABLES

The ALL_QUEUE_TABLES view provides the information about all queue tables in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The role name of the owner of a queue table. |
| queue_table | TEXT | The user-defined name of the queue table. |

| Parameter | Type | Description |
|---|---|---|
| type | CHARACTER VARYING | The type of data stored in the queue table. |
| object_type | TEXT | The user-defined payload type. |
| sort_order | CHARACTER VARYING | The order in which the queue table is sorted. |
| recipients | CHARACTER VARYING | The value is SINGLE. |
| message_grouping | CHARACTER VARYING | The value is NONE. |
| compatible | CHARACTER VARYING | The release number of the PolarDB database compatible with Oracle. The queue table is compatible with this release. |
| primary_instance | NUMERIC | The value is 0. |
| secondary_instance | NUMERIC | The value is 0. |
| owner_instance | NUMERIC | The instance number of the instance to which the queue table belongs. |
| user_comment | CHARACTER VARYING | The comment added when the table was created. |
| secure | CHARACTER VARYING | • YES: indicates that the queue table is secure.<br>• NO: indicates that the queue table is not secure. |

## 14.14 ALL_SEQUENCES

The ALL_SEQUENCES view provides the information about all user-defined sequences on which the user has SELECT or UPDATE permissions.

| Parameter | Type | Description |
|---|---|---|
| sequence_owner | TEXT | The username of the sequence owner. |
| schema_name | TEXT | The name of the schema to which the sequence belongs. |
| sequence_name | TEXT | The name of the sequence. |
| min_value | NUMERIC | The minimun value that the server assigns to the sequence. |

| Parameter | Type | Description |
|---|---|---|
| max_value | NUMERIC | The maximun value that the server assigns to the sequence. |
| increment_by | NUMERIC | The value added to the current sequence number to create the next sequent number. |
| cycle_flag | CHARACTER VARYING | Indicates whether the sequence wraps if it reaches min_value or max_value. |
| order_flag | CHARACTER VARYING | This parameter always returns the value of Y. |
| cache_size | NUMERIC | The number of preallocated sequence numbers stored in memory. |
| last_number | NUMERIC | The value of the last sequence number saved to the disk. |

## 14.15 ALL_SOURCE

The ALL_SOURCE view provides a source code list of the following program types: functions, procedures, triggers, package specifications, and package bodies.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the program belongs. |
| schema_name | TEXT | The name of the schema to which the program belongs. |
| name | TEXT | The name of the program. |
| type | TEXT | The type of the program. Valid values: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER. |
| line | INTEGER | The line number of the source code in a specified program. |
| text | TEXT | The line of the source code text. |

## 14.16 ALL_SUBPART_KEY_COLUMNS

The ALL_SUBPART_KEY_COLUMNS view provides the information about the key columns of those partitioned tables which are subpartitioned in the database

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The owner of the table. |
| schema_name | TEXT | The name of the schema to which the table belongs. |
| name | TEXT | The name of the table to which the column belongs. |
| object_type | CHARACTER(5) | This parameter is supported for compatibil ity only. The value is TABLE. |
| column_name | TEXT | The name of the column on which the key is defined. |
| column_position | INTEGER | The position of this column. For example, a value of 1 indicates the first column and a value of 2 indicates the second column. All columns follow the same rule. |

## 14.17 ALL_SYNONYMS

The ALL_SYNONYMS view provides the information on all synonyms that may be referenced by the current user.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the synonym belongs. |
| schema_name | TEXT | The name of the schema to which the synonym belongs. |
| synonym_name | TEXT | The name of the synonym. |
| table_owner | TEXT | The username of the owner to which the object belongs. |
| table_schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the object that the synonym refers to. |
| db_link | TEXT | The name of any associated database link. |

## 14.18 ALL_TAB_COLUMNS

The ALL_TAB_COLUMNS view provides the information on all columns in all user-defined tables and views.

| Parameter | Type | Description |
|---|---|---|
| owner | CHARACTER VARYING | The username of the owner of the table or view where the column is located. |
| schema_name | CHARACTER VARYING | The name of the schema to which the table or the view belongs. |
| table_name | CHARACTER VARYING | The name of the table or view. |
| column_name | CHARACTER VARYING | The name of the column. |
| data_type | CHARACTER VARYING | The data type of the column. |
| data_length | NUMERIC | The length of the text columns. |
| data_precision | NUMERIC | The precision of the NUMBER column. The precision is measured with the number of digits. |
| data_scale | NUMERIC | The scale of the NUMBER columns. |
| nullable | CHARACTER(1) | Whether the column can be nullable. Valid values:<br>• Y: The column can be null.<br>• N: The column cannot be null. |
| column_id | NUMERIC | The relative position of the column within the table or view. |
| data_default | CHARACTER VARYING | The default value assigned to the column. |

## 14.19 ALL_TAB_PARTITIONS

The ALL_TAB_PARTITIONS view provides the information about all of the partitions in the database.

| Parameter | Type | Description |
|---|---|---|
| table_owner | TEXT | The owner of the table to which the partition belongs. |
| schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table. |

| Parameter | Type | Description |
|---|---|---|
| composite | TEXT | • YES: The table is subpartitioned.<br>• NO: The table is not subpartitioned. |
| partition_name | TEXT | The name of the partition. |
| subpartition_count | BIGINT | The number of subpartitions in the partition. |
| high_value | TEXT | The high partitioning value specified in the CREATE TABLE statement. |
| high_value_length | INTEGER | The length of the high partitioning value. |
| partition_position | INTEGER | This parameter is supported for compatibility only. The value is NULL. |
| tablespace_name | TEXT | The name of the tablespace where the partition is located. |
| pct_free | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| pct_used | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| ini_trans | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| max_trans | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| initial_extent | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| next_extent | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| min_extent | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| max_extent | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| pct_increase | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| freelists | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |

| Parameter | Type | Description |
|---|---|---|
| freelist_groups | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| logging | CHARACTER VARYING(7) | This parameter is supported for compatibility only. The value is YES. |
| compression | CHARACTER VARYING(8) | This parameter is supported for compatibility only. The value is NONE. |
| num_rows | NUMERIC | The same as pg_class.reltuples. |
| blocks | INTEGER | The same as pg_class.relpages. |
| empty_blocks | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| avg_space | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| chain_cnt | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| avg_row_len | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| sample_size | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| last_analyzed | TIMESTAMP WITHOUT TIME ZONE | This parameter is supported for compatibility only. The value is NULL. |
| buffer_pool | CHARACTER VARYING(7) | This parameter is supported for compatibility only. The value is NULL. |
| global_stats | CHARACTER VARYING(3) | This parameter is supported for compatibility only. The value is YES. |
| user_stats | CHARACTER VARYING(3) | This parameter is supported for compatibility only. The value is NO. |
| backing_table | REGCLASS | The name of the partition backup table. |

## 14.20 ALL_TAB_SUBPARTITIONS

The ALL_TAB_SUBPARTITIONS view provides the information about all subpartitions in the database.

| Parameter | Type | Description |
|---|---|---|
| table_owner | TEXT | The owner of the table to which the subpartition belongs. |
| schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table. |
| partition_name | TEXT | The name of the partition. |
| subpartition_name | TEXT | The name of the subpartition. |
| high_value | TEXT | The high subpartitioning value specified in the CREATE TABLE statement. |
| high_value_length | INTEGER | The length of the high subpartitioning value. |
| subpartition_position | INTEGER | This parameter is supported for compatibility only. The value is NULL. |
| tablespace_name | TEXT | The name of the tablespace where the subpartition is located. |
| pct_free | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| pct_used | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| ini_trans | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| max_trans | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| initial_extent | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| next_extent | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| min_extent | NUMERIC | This parameter is supported for compatibility only. The value is 0. |

| Parameter | Type | Description |
|---|---|---|
| max_extent | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| pct_increase | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| freelists | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| freelist_groups | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| logging | CHARACTER VARYING(7) | This parameter is supported for compatibility only. The value is YES. |
| compression | CHARACTER VARYING(8) | This parameter is supported for compatibility only. The value is NONE. |
| num_rows | NUMERIC | The same as pg_class.reltuples. |
| blocks | INTEGER | The same as pg_class.relpages. |
| empty_blocks | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| avg_space | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| chain_cnt | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| avg_row_len | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| sample_size | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| last_analyzed | TIMESTAMP WITHOUT TIME ZONE | This parameter is supported for compatibility only. The value is NULL. |
| buffer_pool | CHARACTER VARYING(7) | This parameter is supported for compatibility only. The value is NULL. |
| global_stats | CHARACTER VARYING(3) | This parameter is supported for compatibility only. The value is YES. |
| user_stats | CHARACTER VARYING(3) | This parameter is supported for compatibility only. The value is NO. |
| backing_table | REGCLASS | The name of the subpartition backup table. |

## 14.21 ALL_TABLES

The ALL_TABLES view provides the information on all user-defined tables.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the table belongs. |
| schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table. |
| tablespace_name | TEXT | The name of the tablespace where the table is located if this tablespace is not the default tablespace. |
| status | CHARACTER VARYING(5) | Indicates that whether the status of the table is valid. This parameter is supported for compatibility only. The value is VALID. |
| temporary | CHARACTER(1) | • Y: indicates that the table is a temporary table.<br>• N: indicates that the table is not a temporary table. |

## 14.22 ALL_TRIGGERS

The ALL_TRIGGERS view provides the information about the triggers on tables that may be accessed by the current user.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the trigger belongs. |
| schema_name | TEXT | The name of the schema to which the trigger belongs. |
| trigger_name | TEXT | The name of the trigger. |

| Parameter | Type | Description |
|---|---|---|
| trigger_type | TEXT | The type of the trigger. Valid values:<br>• BEFORE<br>• ROW<br>• BEFORE<br>• STATEMENT<br>• AFTER<br>• ROW<br>• AFTER STATEMENT |
| triggering_event | TEXT | The event that activate the trigger. |
| table_owner | TEXT | The username of the owner that the table belongs to. The trigger is defined in this table. |
| base_object_type | TEXT | This parameter is supported for compatibility only. The value is TABLE. |
| table_name | TEXT | The name of the table on which the trigger is defined. |
| referencing_name | TEXT | This parameter is supported for compatibility only. The value is REFERENCING NEW AS NEW OLD AS OLD. |
| status | TEXT | The status of the trigger. A value of VALID indicates that the trigger is enabled, and a value of NOTVALID indicates that the trigger is disabled. |
| Description | TEXT | This parameter is supported for compatibility only. |
| trigger_body | TEXT | The body of the trigger. |
| action_statement | TEXT | The SQL statement that is executed when the trigger activates. |

## 14.23 ALL_TYPES

The ALL_TYPES view provides the information about the object types available to the current user.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The owner of an object type. |

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema in which a type is defined. |
| type_name | TEXT | The name of the type. |
| type_oid | OID | The object identifier (OID) of the type. |
| typecode | TEXT | The typecode of the type. Valid values:<br>• OBJECT<br>• COLLECTION<br>• OTHER |
| attributes | INTEGER | The number of attributes in the type. |

## 14.24 ALL_USERS

The ALL_USERS view provides the information on all usernames.

| Parameter | Type | Description |
|---|---|---|
| username | TEXT | The name of a user. |
| user_id | OID | The numeric user id assigned to the user. |
| created | TIMESTAMP WITHOUT TIME ZONE | This parameter is supported for compatibility only. The value is NULL. |

## 14.25 ALL_VIEW_COLUMNS

The ALL_VIEW_COLUMNS view provides the information on all columns in all user-defined views.

| Parameter | Type | Description |
|---|---|---|
| owner | CHARACTER VARYING | The username of the owner to which the view belongs. |
| schema_name | CHARACTER VARYING | The name of the schema to which the view belongs. |
| view_name | CHARACTER VARYING | The name of the view. |
| column_name | CHARACTER VARYING | The name of the column. |
| data_type | CHARACTER VARYING | The data type of the column. |

| Parameter | Type | Description |
|---|---|---|
| data_length | NUMERIC | The length of text columns. |
| data_precision | NUMERIC | The precision of the NUMBER column . The precision is measured in the number of digits. |
| data_scale | NUMERIC | The scale of the NUMBER columns. |
| nullable | CHARACTER(1) | Indicates whether the column can be null. Valid values:<br>• Y: indicates that the column can be null.<br>• N: indicates that the column cannot be null. |
| column_id | NUMERIC | The relative position of the column within the view. |
| data_default | CHARACTER VARYING | The default value assigned to the column. |

## 14.26 ALL_VIEWS

The ALL_VIEWS view provides the information about all user-defined views.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the view belongs. |
| schema_name | TEXT | The name of the schema to which the view belongs. |
| view_name | TEXT | The name of the view. |
| text | TEXT | The SELECT statement that defines the view. |

## 14.27 DBA_ALL_TABLES

The DBA_ALL_TABLES view provides the information about all tables in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which a table belongs. |

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table. |
| tablespace_name | TEXT | The name of the tablespace in which the table is located if this tablespace is not the default tablespace. |
| status | CHARACTER VARYING (5) | This parameter is supported for compatibility only. The value is VALID. |
| temporary | TEXT | • Y: indicates that the table is a temporary table.<br>• N: indicates that the table is a permanent table. |

## 14.28 DBA_CONS_COLUMNS

The DBA_CONS_COLUMNS view provides the information about all columns that are included in constraints. These constraints are specified on all tables in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which a constraint belongs. |
| schema_name | TEXT | The name of the schema to which the constraint belongs . |
| constraint_name | TEXT | The name of the constraint. |
| table_name | TEXT | The name of the table to which the constraint belongs . |
| column_name | TEXT | The name of the column referenced in the constraint. |
| position | SMALLINT | The position of the column within the object definition. |
| constraint_def | TEXT | The definition of the constraint. |

## 14.29 DBA_CONSTRAINTS

The DBA_CONSTRAINTS view provides the information about all constraints on tables in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which a constraint belongs. |
| schema_name | TEXT | The name of the schema to which the constraint belongs. |
| constraint_name | TEXT | The name of the constraint. |
| constraint_type | TEXT | The type of the constraint. Valid values:<br>• C: the check constraint<br>• F: the foreign key constraint<br>• P: the primary key constraint<br>• U: the unique key constraint<br>• R: the referential integrity constraint<br>• V: the constraint on a view<br>• O: with a read-only property on a view |
| table_name | TEXT | The name of the table to which the constraint belongs. |
| search_condition | TEXT | The search condition that applies to a check constraint. |
| r_owner | TEXT | The owner of a table referenced by a referential constraint. |
| r_constraint_name | TEXT | The name of the constraint definition for a referenced table. |

| Parameter | Type | Description |
|---|---|---|
| delete_rule | TEXT | The delete rule for a referential constraint. Valid values:<br>• C: cascade<br>• R: restrict<br>• N: no action |
| deferrable | BOOLEAN | Indicates whether the constraint is deferrable. Valid values: T and F. |
| deferred | BOOLEAN | Indicates whether the constraint has been deferred . Valid values: T and F. |
| index_owner | TEXT | The username of the owner to which an index belongs. |
| index_name | TEXT | The name of the index. |
| constraint_def | TEXT | The definition of the constraint. |

## 14.30 DBA_DB_LINKS

The DBA_DB_LINKS view provides the information about all database links in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which a database link belongs. |
| db_link | TEXT | The name of the database link. |
| type | CHARACTER VARYING | The type of the remote server. Valid values: REDWOOD and PolarDB. |
| username | TEXT | The username of the user logging in. |
| host | TEXT | The name or IP address of the remote server . |

## 14.31 DBA_DIRECTORIES

The DBA_DIRECTORIES view provides the information about all directories created by running the CREATE DIRECTORY command.

| Parameter | Type | Description |
|---|---|---|
| owner | CHARACTER VARYING(30) | The username of the owner to which a directory belongs. |
| directory_name | CHARACTER VARYING(30) | The alias name assigned to the directory. |
| directory_path | CHARACTER VARYING(4000) | The path to the directory. |

## 14.32 DBA_IND_COLUMNS

The DBA_IND_COLUMNS view provides the information about all columns included in indexes on all tables in the database.

| Parameter | Type | Description |
|---|---|---|
| index_owner | TEXT | The username of the owner to which an index belongs. |
| schema_name | TEXT | The name of the schema to which the index belongs. |
| index_name | TEXT | The name of the index. |
| table_owner | TEXT | The username of the owner to which the table belongs. |
| table_name | TEXT | The name of the table to which the index belongs. |
| column_name | TEXT | The name or property name of the object column. |
| column_position | SMALLINT | The position of the column in the index. |
| column_length | SMALLINT | The length of the column. Unit: bytes. |
| char_length | NUMERIC | The length of the column. Unit: characters. |
| descend | CHARACTER(1) | This parameter is supported for compatibility only. The value is Y in descending order. |

## 14.33 DBA_INDEXES

The DBA_INDEXES view provides the information about all indexes in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which an index belongs. |
| schema_name | TEXT | The name of the schema where the index is located. |
| index_name | TEXT | The name of the index. |
| index_type | TEXT | The index type is BTREE. This parameter is supported for compatibility only. |
| table_owner | TEXT | The username of the owner of an indexed table. |
| table_name | TEXT | The name of the indexed table. |
| table_type | TEXT | This parameter is supported for compatibil ity only. The value is TABLE. |
| uniqueness | TEXT | Indicates whether the index is UNIQUE or NONUNIQUE. |
| compression | CHARACTER(1) | The value is N (not compressed). This parameter is supported for compatibility only. |
| tablespace_name | TEXT | The name of the tablespace in which the table is located if this tablespace is not the default tablespace. |
| logging | TEXT | This parameter is supported for compatibil ity only. The value is LOGGING. |
| status | TEXT | Indicates whether the state of the object is valid. Valid values: VALID and INVALID. |
| partitioned | CHARACTER(3) | Indicates that the index is partitioned. The value is NO. |
| temporary | CHARACTER(1) | Indicates that the index is on a temporary table. The value is N. |
| secondary | CHARACTER(1) | This parameter is supported for compatibil ity only. The value is N. |
| join_index | CHARACTER(3) | This parameter is supported for compatibil ity only. The value is NO. |

| Parameter | Type | Description |
|---|---|---|
| dropped | CHARACTER(3) | This parameter is supported for compatibility only. The value is NO. |

## 14.34 DBA_JOBS

The DBA_JOBS view provides the information about all jobs in the database.

| Parameter | Type | Description |
|---|---|---|
| job | INTEGER | The identifier of a job (job ID). |
| log_user | TEXT | The name of the user that submitted the job . |
| priv_user | TEXT | The same as log_user. This parameter is supported for compatibility only. |
| schema_user | TEXT | The name of the schema used to parse the job. |
| last_date | TIMESTAMP WITH TIME ZONE | The last date when this job was executed with the expected result returned. |
| last_sec | TEXT | The same as last_date. |
| this_date | TIMESTAMP WITH TIME ZONE | The date when the system starts to execute the job. |
| this_sec | TEXT | The same as this_date. |
| next_date | TIMESTAMP WITH TIME ZONE | The next date when this job will be executed . |
| next_sec | TEXT | The same as next_date. |
| total_time | INTERVAL | The period in which the job is executed. Unit : seconds. |
| broken | TEXT | • Y: indicates no attempt is made to run this job.<br>• N: indicates this job will attempt to execute. |
| interval | TEXT | The interval at which the job is repeated. |
| failures | BIGINT | The number of times that the job has failed since the last successful execution. |

| Parameter | Type | Description |
|---|---|---|
| what | TEXT | The job definition that runs when the job executes. The job definition appears as a PL /SQL code block. |
| nls_env | CHARACTER VARYING (4000) | The value is NULL. This parameter is supported for compatibility only. |
| misc_env | BYTEA | The value is NULL. This parameter is supported for compatibility only. |
| instance | NUMERIC | The value is 0. This parameter is supported for compatibility only. |

## 14.35 DBA_OBJECTS

The DBA_OBJECTS view provides the information about all objects in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the object belongs. |
| schema_name | TEXT | The name of the schema to which the object belongs. |
| object_name | TEXT | The name of the object. |
| object_type | TEXT | The type of the object. Valid values: INDEX , FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW. |
| status | CHARACTER VARYING | This parameter is supported for compatibility only. The value is VALID. |
| temporary | TEXT | • Y: indicates that the table is a temporary table.<br>• N: indicates that the table is a permanent table. |

## 14.36 DBA_PART_KEY_COLUMNS

The DBA_PART_KEY_COLUMNS view provides the information about key columns of partitioned tables in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The owner of the table. |
| schema_name | TEXT | The name of the schema to which the table belongs. |
| name | TEXT | The name of the table to which the column belongs. |
| object_type | CHARACTER(5) | This parameter is supported for compatibil ity only. The value is TABLE. |
| column_name | TEXT | The name of the column on which the key is defined. |
| column_position | INTEGER | The position of this column. For example, a value of 1 indicates the first column and a value of 2 indicates the second column. All columns follow the same rule. |

## 14.37 DBA_PART_TABLES

The DBA_PART_TABLES view provides the information about all partitioned tables in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The owner of a partitioned table. |
| schema_name | TEXT | The schema to which the table belongs. |
| table_name | TEXT | The name of the table. |
| partitioning_type | TEXT | The partition type used to define table partitions. |
| subpartitioning_type | TEXT | The subpartition type used to define table subpartitions. |
| partition_count | BIGINT | The number of partitions in the table. |
| def_subpartition_count | INTEGER | The number of subpartitions in the table. |

| Parameter | Type | Description |
|---|---|---|
| partitioning_key_count | INTEGER | The number of specified partition keys. |
| subpartitioning_key_count | INTEGER | The number of specified subpartition keys. |
| status | CHARACTER VARYING(8) | This parameter is supported for compatibility only. The value is VALID. |
| def_tablespace_name | CHARACTER VARYING(30) | This parameter is supported for compatibility only. The value is NULL. |
| def_pct_free | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_pct_used | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_ini_trans | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_max_trans | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_initial_extent | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| def_next_extent | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| def_min_extents | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| def_max_extents | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| def_pct_increase | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| def_freelists | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_freelist_groups | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_logging | CHARACTER VARYING(7) | This parameter is supported for compatibility only. The value is YES. |
| def_compression | CHARACTER VARYING(8) | This parameter is supported for compatibility only. The value is NONE. |

| Parameter | Type | Description |
|---|---|---|
| def_buffer_pool | CHARACTER VARYING(7) | This parameter is supported for compatibility only. The value is DEFAULT. |
| ref_ptn_constraint_name | CHARACTER VARYING(30) | This parameter is supported for compatibility only. The value is NULL. |
| interval | CHARACTER VARYING(1000) | This parameter is supported for compatibility only. The value is NULL. |

## 14.38 DBA_PROFILES

The DBA_PROFILES view provides the information about existing profiles. The table includes a row for each profile or resource combination.

| Parameter | Type | Description |
|---|---|---|
| profile | CHARACTER VARYING(128) | The name of the profile. |
| resource_name | CHARACTER VARYING(32) | The name of the resource associated with the profile. |
| resource_type | CHARACTER VARYING(8) | The type of resource managed by the profile; currently PASSWORD for all supported resources. |
| limit | CHARACTER VARYING(128) | The limit values of the resource. |
| common | CHARACTER VARYING(3) | • YES: indicates that the profile is a user-created profile.<br>• NO: indicates that the profile is a system-defined profile. |

## 14.39 DBA_QUEUES

The DBA_QUEUES view provides the information about any defined queues.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which a queue belongs. |
| Parameter | TEXT | The name of the queue. |

| Parameter | Type | Description |
|---|---|---|
| queue_table | TEXT | The name of the queue table to which the queue belongs. |
| qid | OID | The object ID of the queue. This object ID is assigned by the system. |
| queue_type | CHARACTER VARYING | The queue type. Valid values: EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, and NORMAL_QUEUE. |
| max_retries | NUMERIC | The maximum number of dequeuing attempts. |
| retrydelay | NUMERIC | The maximum time allowed between retries . |
| enqueue_enabled | CHARACTER VARYING | • YES: indicates that the queue allows enqueuing.<br>• NO: indicates that the queue does not allow enqueuing. |
| dequeue_enabled | CHARACTER VARYING | • YES: indicates that the queue allows dequeuing.<br>• NO: indicates that the queue does not allow dequeuing. |
| retention | CHARACTER VARYING | The number of seconds that a processed message is retained in the queue. |
| user_comment | CHARACTER VARYING | The user-defined comment. |
| network_name | CHARACTER VARYING | The name of the network in which the queue is. |
| sharded | CHARACTER VARYING | • YES: indicates the queue is in a sharded network.<br>• NO: indicates the queue is not in a sharded network. |

## 14.40 DBA_QUEUE_TABLES

The DBA_QUEUE_TABLES view provides the information about all queue tables in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The role name of the owner of a queue table. |
| queue_table | TEXT | The user-defined name of the queue table. |
| type | CHARACTER VARYING | The type of data stored in the queue table. |
| object_type | TEXT | The type of the user-defined payload. |
| sort_order | CHARACTER VARYING | The order in which the queue table is sorted. |
| recipients | CHARACTER VARYING | The value is SINGLE. |
| message_grouping | CHARACTER VARYING | The value is NONE. |
| compatible | CHARACTER VARYING | The release number of the PolarDB database compatible with Oracle. The queue table is compatible with this release. |
| primary_instance | NUMERIC | The value is 0. |
| secondary_instance | NUMERIC | The value is 0. |
| owner_instance | NUMERIC | The instance number of the instance to which the queue table belongs. |
| user_comment | CHARACTER VARYING | The comment added when the table was created. |
| secure | CHARACTER VARYING | • YES: indicates that the queue table is secure.<br>• NO: indicates that the queue table is not secure. |

## 14.41 DBA_ROLE_PRIVS

The DBA_ROLE_PRIVS view provides the information on all roles that have been granted to users. A row is created for each role to which a user has been granted.

| Parameter | Type | Description |
|---|---|---|
| grantee | TEXT | The username to whom the role is granted. |
| granted_role | TEXT | The name of the role granted to the grantee. |
| admin_option | TEXT | YES: indicates that the role is granted with the admin option. NO: indicates that the role is granted with the other options other than admin. |
| default_role | TEXT | YES: indicates that the role is enabled when the grantee creates a session. |

## 14.42 DBA_ROLES

The DBA_ROLES view provides the information on all roles with the NOLOGIN property (groups).

| Parameter | Type | Description |
|---|---|---|
| role | TEXT | The name of a role with the NOLOGIN property. For example, a group. |
| password_required | TEXT | This parameter is supported for compatibility only. The value is N. |

## 14.43 DBA_SEQUENCES

The DBA_SEQUENCES view provides the information about all user-defined sequences.

| Parameter | Type | Description |
|---|---|---|
| sequence_owner | TEXT | The username of the owner to which the sequence belongs. |
| schema_name | TEXT | The name of the schema to which the sequence belongs. |
| sequence_name | TEXT | The name of the sequence. |
| min_value | NUMERIC | The minimum value that the server assigns to the sequence. |

| Parameter | Type | Description |
|---|---|---|
| max_value | NUMERIC | The maximum value that the server assigns to the sequence. |
| increment_by | NUMERIC | The value added to the current sequence number to create the next sequence number. |
| cycle_flag | CHARACTER VARYING | Indicates whether the sequence wraps if it reaches min_value or max_value. |
| order_flag | CHARACTER VARYING | This parameter always returns the value of Y. |
| cache_size | NUMERIC | The number of preallocated sequence numbers stored in memory. |
| last_number | NUMERIC | The value of the last sequence number saved to the disk. |

## 14.44 DBA_SOURCE

The DBA_SOURCE view provides a list of source code for all objects in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the program belongs. |
| schema_name | TEXT | The name of the schema to which the program belongs. |
| name | TEXT | The name of the program. |
| type | TEXT | The type of the program. Valid values : FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER. |
| line | INTEGER | The line number of the source code in a specified program. |
| text | TEXT | The line of the source code text. |

## 14.45 DBA_SUBPART_KEY_COLUMNS

The DBA_SUBPART_KEY_COLUMNS view provides the information about the key columns of those partitioned tables which are subpartitioned in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The owner of the table. |
| Schema_name | TEXT | The name of the schema to which the table belongs. |
| name | TEXT | The name of the table to which the column belongs. |
| object_type | CHARACTER(5) | This parameter is supported for compatibility only. The value is TABLE. |
| column_name | TEXT | The name of the column on which the key is defined. |
| column_position | INTEGER | The position of this column. For example, a value of 1 indicates the first column and a value of 2 indicates the second column. All columns follow the same rule. |

## 14.46 DBA_SYNONYMS

The DBA_SYNONYM view provides the information about all synonyms in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which a synonym belongs. |
| schema_name | TEXT | The name of the schema to which the synonym belongs. |
| synonym_name | TEXT | The name of the synonym. |
| table_owner | TEXT | The username of the owner of the table on which the synonym is defined. |
| Table_schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table on which the synonym is defined. |

| Parameter | Type | Description |
|---|---|---|
| db_link | TEXT | The name of any associated database link. |

## 14.47 DBA_TAB_COLUMNS

The DBA_TAB_COLUMNS view provides the information about all columns in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | CHARACTER VARYING | The username of the owner of a table or view to which the column belongs . |
| schema_name | CHARACTER VARYING | The name of the schema to which a table or view belongs. |
| table_name | CHARACTER VARYING | The name of the table or view to which the column belongs. |
| column_name | CHARACTER VARYING | The name of the column. |
| data_type | CHARACTER VARYING | The data type of the column. |
| data_length | NUMERIC | The length of text columns. |
| data_precision | NUMERIC | The precision of the NUMBER column . The precision is measured in the number of digits. |
| data_scale | NUMERIC | The scale of the NUMBER columns. |
| nullable | CHARACTER(1) | Indicates whether the column can be null. Valid values:<br>• Y: indicates that the column can be null.<br>• N: indicates that the column cannot be null. |
| column_id | NUMERIC | The relative position of the column within the table or view. |
| data_default | CHARACTER VARYING | The default value assigned to the column. |

## 14.48 DBA_TAB_PARTITIONS

The DBA_TAB_PARTITIONS view provides the information about all partitions that locate in the database.

| Parameter | Type | Description |
|---|---|---|
| table_owner | TEXT | The owner of the table where the partition is located. |
| schema_name | TEXT | The name of the schema where the table is located. |
| table_name | TEXT | The name of the table. |
| composite | TEXT | • YES: The table is subpartitioned.<br>• NO: The table is not subpartitioned. |
| partition_name | TEXT | The name of the partition. |
| subpartition_count | BIGINT | The number of subpartitions in a partition. |
| high_value | TEXT | The high partitioning value specified in the CREATE TABLE statement. |
| high_value_length | INTEGER | The length of the high partitioning value. |
| partition_position | INTEGER | The ordinal position of this partition. For example, a value of 1 indicates the first partition and a value of 2 indicates the second partition. All positions follow the same rule. |
| tablespace_name | TEXT | The name of the tablespace where the partition is located. |
| pct_free | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| pct_used | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| ini_trans | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| max_trans | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| initial_extent | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |

| Parameter | Type | Description |
|---|---|---|
| next_extent | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| min_extent | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| max_extent | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| pct_increase | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| freelists | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| freelist_groups | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| logging | CHARACTER VARYING (7) | This parameter is supported for compatibility only. The value is YES. |
| compression | CHARACTER VARYING (8) | This parameter is supported for compatibility only. The value is NONE. |
| num_rows | NUMERIC | The same as pg_class.reltuples. |
| blocks | INTEGER | The same as pg_class.relpages. |
| empty_blocks | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| avg_space | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| chain_cnt | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| avg_row_len | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| sample_size | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| last_analyzed | TIMESTAMP WITHOUT TIME ZONE | This parameter is supported for compatibility only. The value is NULL. |
| buffer_pool | CHARACTER VARYING (7) | This parameter is supported for compatibility only. The value is NULL. |
| global_stats | CHARACTER VARYING (3) | This parameter is supported for compatibility only. The value is YES. |

| Parameter | Type | Description |
|---|---|---|
| user_stats | CHARACTER VARYING (3) | This parameter is supported for compatibility only. The value is NO. |
| backing_table | REGCLASS | The name of the partition backup table. |

## 14.49 DBA_TAB_SUBPARTITIONS

The DBA_TAB_SUBPARTITIONS view provides the information about all subpartitions that locate in the database.

| Parameter | Type | Description |
|---|---|---|
| table_owner | TEXT | The owner of the table where a subpartition is located. |
| schema_name | TEXT | The name of the schema where the table is located. |
| table_name | TEXT | The name of the table. |
| partition_name | TEXT | The name of the partition. |
| subpartition_name | TEXT | The name of the subpartition. |
| high_value | TEXT | The high subpartitioning value specified in the CREATE TABLE statement. |
| high_value_length | INTEGER | The length of the high subpartitioning value. |
| subpartition_position | INTEGER | The ordinal position of this subpartition. For example, a value of 1 indicates the first subpartition and a value of 2 indicates the second subpartition. All positions of subpartitions follow the same rule. |
| tablespace_name | TEXT | The name of the tablespace where the subpartition is located. |
| pct_free | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| pct_used | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| ini_trans | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| max_trans | NUMERIC | This parameter is supported for compatibility only. The value is 0. |

| Parameter | Type | Description |
|---|---|---|
| initial_extent | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| next_extent | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| min_extent | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| max_extent | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| pct_increase | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| freelists | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| freelist_groups | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| logging | CHARACTER VARYING (7) | This parameter is supported for compatibility only. The value is YES. |
| compression | CHARACTER VARYING (8) | This parameter is supported for compatibility only. The value is NONE. |
| num_rows | NUMERIC | The same as pg_class.reltuples. |
| blocks | INTEGER | The same as pg_class.relpages. |
| empty_blocks | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| avg_space | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| chain_cnt | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| avg_row_len | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| sample_size | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| last_analyzed | TIMESTAMP WITHOUT TIME ZONE | This parameter is supported for compatibility only. The value is NULL. |
| buffer_pool | CHARACTER VARYING (7) | This parameter is supported for compatibility only. The value is NULL. |

| Parameter | Type | Description |
|---|---|---|
| global_stats | CHARACTER VARYING (3) | This parameter is supported for compatibility only. The value is YES. |
| user_stats | CHARACTER VARYING (3) | This parameter is supported for compatibility only. The value is NO. |
| backing_table | REGCLASS | The name of the subpartition backup table. |

## 14.50 DBA_TABLES

The DBA_TABLES view provides the information about all tables in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which the table belongs. |
| schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table. |
| tablespace_name | TEXT | The name of the tablespace where the table is located if this tablespace is not the default tablespace. |
| status | CHARACTER VARYING(5) | This parameter is supported for compatibility only. The value is VALID. |
| temporary | CHARACTER(1) | • Y: indicates that the table is a temporary table.<br>• N: indicates that the table is a permanent table. |

## 14.51 DBA_TRIGGERS

The DBA_TRIGGERS view provides the information about all triggers in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which a trigger belongs. |
| schema_name | TEXT | The name of the schema to which the trigger belongs. |
| trigger_name | TEXT | The name of the trigger. |

| Parameter | Type | Description |
|---|---|---|
| trigger_type | TEXT | The type of the trigger. Valid values:<br>• BEFORE<br>• ROW<br>• BEFORE<br>• STATEMENT<br>• AFTER<br>• ROW<br>• AFTER STATEMENT |
| triggering_event | TEXT | The event that activate the trigger. |
| table_owner | TEXT | The username of the owner of the table on which the trigger is defined. |
| base_object_type | TEXT | This parameter is supported for compatibil ity only. The value is TABLE. |
| table_name | TEXT | The name of the table on which the trigger is defined. |
| referencing_names | TEXT | This parameter is supported for compatibil ity only. The value is REFERENCING NEW AS NEW OLD AS OLD. |
| status | TEXT | Indicates whether the trigger is enabled ( VALID) or disabled (NOTVALID). |
| description | TEXT | This parameter is supported for compatibil ity only. |
| trigger_body | TEXT | The body of the trigger. |
| action_statement | TEXT | The SQL statement that is executed when the trigger is activated. |

## 14.52 DBA_TYPES

The DBA_TYPES view provides the information about all object types in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The owner of an object type. |
| schema_name | TEXT | The name of the schema in which a type is defined. |
| type_name | TEXT | The name of the type. |

| Parameter | Type | Description |
|---|---|---|
| type_oid | OID | The object identifier (OID) of the type. |
| typecode | TEXT | The typecode of the type. Valid values:<br>• OBJECT<br>• COLLECTION<br>• OTHER |
| attributes | INTEGER | The number of attributes in the type. |

## 14.53 DBA_USERS

The DBA_USERS view provides the information about all users of the database.

| Parameter | Type | Description |
|---|---|---|
| username | TEXT | The username of the user. |
| user_id | OID | The ID number of the user. |
| password | CHARACTER VARYING (30) | The encrypted password of the user. |
| account_status | CHARACTER VARYING (32) | The current status of the account. Valid values:<br>• OPEN<br>• EXPIRED<br>• EXPIRED(GRACE)<br>• EXPIRED & LOCKED<br>• EXPIRED & LOCKED(TIMED)<br>• EXPIRED(GRACE) & LOCKED<br>• EXPIRED(GRACE) & LOCKED(TIMED)<br>• LOCKED<br>• LOCKED(TIMED)<br>You can use the edb_get_role_status(role_id) function to retrieve the current status of the account. |
| lock_date | TIMESTAMP WITHOUT TIME ZONE | If the account status is LOCKED, the lock_date parameter indicates the date and time when the account was locked. |

| Parameter | Type | Description |
|---|---|---|
| expiry_date | TIMESTAMP WITHOUT TIME ZONE | The expiration date of the password. You can use the edb_get_password_expiry_date(role_id) function to retrieve the expiration date of the current password. |
| default_tablespace | TEXT | The default tablespace associated with the account. |
| temporary_tablespace | CHARACTER VARYING (30) | This parameter is supported for compatibility only. The value is '' (an empty string). |
| created | TIMESTAMP WITHOUT TIME ZONE | This parameter is supported for compatibility only. The value is NULL. |
| profile | CHARACTER VARYING (30) | The profile associated with the user. |
| initial_rsrc_consumer_group | CHARACTER VARYING (30) | This parameter is supported for compatibility only. The value is NULL. |
| external_name | CHARACTER VARYING (4000) | This parameter is supported for compatibility only. The value is NULL. |

## 14.54 DBA_VIEW_COLUMNS

The DBA_VIEW_COLUMNS view provides the information on all columns in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | CHARACTER VARYING | The username of the owner to which a view belongs. |
| schema_name | CHARACTER VARYING | The name of the schema to which the view belongs. |
| view_name | CHARACTER VARYING | The name of the view. |
| column_name | CHARACTER VARYING | The name of the column. |
| data_type | CHARACTER VARYING | The data type of the column. |
| data_length | NUMERIC | The length of text columns. |
| data_precision | NUMERIC | The precision of NUMBER columns. The precision is measured in the number of digits. |
| data_scale | NUMERIC | The scale of NUMBER columns. |

| Parameter | Type | Description |
|---|---|---|
| nullable | CHARACTER(1) | Indicates whether the column can be null. Valid values:<br>• Y: indicates that the column can be null.<br>• N: indicates that the column cannot be null. |
| column_id | NUMERIC | The relative position of a column within the view. |
| data_default | CHARACTER VARYING | The default value assigned to the column. |

## 14.55 DBA_VIEWS

The DBA_VIEWS view provides the information about all views in the database.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which a view belongs. |
| schema_name | TEXT | The name of the schema to which the view belongs. |
| view_name | TEXT | The name of the view. |
| text | TEXT | The tex of the SELECT statement that defines the view. |

## 14.56 USER_ALL_TABLES

The USER_ALL_TABLES view provides the information about all tables owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table. |
| tablespace_name | TEXT | The name of the tablespace where the table is located if this tablespace is not the default tablespace. |

| Parameter | Type | Description |
|---|---|---|
| status | CHARACTER VARYING (5) | This parameter is supported for compatibility only. The value is VALID. |
| temporary | TEXT | • Y: indicates that the table is a temporary table.<br>• N: indicates that the table is a permanent table. |

## 14.57 USER_CONS_COLUMNS

The USER_CONS_COLUMNS view provides the information about all columns that are included in constraints in tables that are owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The username of the owner to which a constraint belongs. |
| schema_name | TEXT | The name of the schema to which the constraint belongs. |
| constraint_name | TEXT | The name of the constraint. |
| table_name | TEXT | The name of the table to which the constraint belongs. |
| column_name | TEXT | The name of a column referenced in the constraint. |
| position | SMALLINT | The position of the column within the object definition. |
| constraint_def | TEXT | The definition of the constraint. |

## 14.58 USER_CONSTRAINTS

The USER_CONSTRAINTS view provides the information about all constraints placed on tables that are owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| owner | TEXT | The name of the owner of the constraint. |
| schema_name | TEXT | The name of the schema to which the constraint belongs. |
| constraint_name | TEXT | The name of the constraint. |
| constraint_type | TEXT | The type of the constraint. Valid values:<br>• C: the check constraint<br>• F: the foreign key constraint<br>• P: the primary key constraint<br>• U: the unique key constraint<br>• R: the referential integrity constraint<br>• V: the constraint on a view<br>• O: with a read-only property on a view |
| table_name | TEXT | The name of the table to which the constraint belongs. |
| search_condition | TEXT | The search condition that applies to a check constraint. |
| r_owner | TEXT | The owner of a table referenced by a referential constraint. |
| r_constraint_name | TEXT | The name of the constraint definition for a referenced table. |
| delete_rule | TEXT | The delete rule for a referential constraint. Valid values:<br>• C: cascade<br>• R: restrict<br>• N: no action |
| deferrable | BOOLEAN | Indicates whether the constraint is deferrable. Valid values: T and F. |

| Parameter | Type | Description |
|---|---|---|
| deferred | BOOLEAN | Indicates whether the constraint has been deferred. Valid values: T and F. |
| index_owner | TEXT | The username of the owner to which an index belongs. |
| index_name | TEXT | The name of the index. |
| constraint_def | TEXT | The definition of the constraint. |

## 14.59 USER_DB_LINKS

The USER_DB_LINKS view provides the information about all database links owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| db_link | TEXT | The name of the database link. |
| type | CHARACTER VARYING | The type of the remote server. Valid values: REDWOOD and PolarDB. |
| username | TEXT | The username of the user logging in. |
| password | TEXT | The password used for authentication on the remote server. |
| host | TEXT | The name or IP address of the remote server. |

## 14.60 USER_IND_COLUMNS

The USER_IND_COLUMNS view provides the information about all columns included in indexes on the tables that are owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which an index belongs. |
| index_name | TEXT | The name of the index. |
| table_name | TEXT | The name of the table to which the index belongs. |
| column_name | TEXT | The name of the column. |
| column_position | SMALLINT | The position of the column in the index. |

| Parameter | Type | Description |
|---|---|---|
| column_length | SMALLINT | The length of the column. Unit: bytes. |
| char_length | NUMERIC | The length of the column. Unit: characters. |
| descend | CHARACTER(1) | This parameter is supported for compatibility only. The value is Y in descending order. |

## 14.61 USER_INDEXES

The USER_INDEXES view provides the information about all indexes on tables that are owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which an index belongs. |
| index_name | TEXT | The name of the index. |
| index_type | TEXT | This parameter is supported for compatibil ity only. The index type is BTREE. |
| table_owner | TEXT | The username of the owner of an indexed table. |
| table_name | TEXT | The name of the indexed table. |
| table_type | TEXT | This parameter is supported for compatibil ity only. The value is TABLE. |
| uniqueness | TEXT | Indicates whether the index is UNIQUE or NONUNIQUE. |
| compression | CHARACTER(1) | This parameter is supported for compatibil ity only. The value is N (not compressed). |
| tablespace_name | TEXT | The name of the tablespace where the table is located if this tablespace is not the default tablespace. |
| logging | TEXT | This parameter is supported for compatibil ity only. The value is LOGGING. |
| status | TEXT | Indicates whether the state of the object is valid. Valid values: VALID and INVALID. |
| partitioned | CHARACTER(3) | This parameter is supported for compatibil ity only. The value is NO. |

| Parameter | Type | Description |
|---|---|---|
| temporary | CHARACTER(1) | This parameter is supported for compatibility only. The value is N. |
| secondary | CHARACTER(1) | This parameter is supported for compatibility only. The value is N. |
| join_index | CHARACTER(3) | This parameter is supported for compatibility only. The value is NO. |
| dropped | CHARACTER(3) | This parameter is supported for compatibility only. The value is NO. |

## 14.62 USER_JOBS

The USER_JOBS view provides the information about all jobs owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| job | INTEGER | The identifier of a job (Job ID). |
| log_user | TEXT | The name of the user that submitted the job. |
| priv_user | TEXT | The same as log_user. This parameter is supported for compatibility only. |
| schema_user | TEXT | The name of the schema used to parse the job. |
| last_date | TIMESTAMP WITH TIME ZONE | The last date when this job was executed with the expected result returned. |
| last_sec | TEXT | The same as last_date. |
| this_date | TIMESTAMP WITH TIME ZONE | The date when the system starts to execute the job. |
| this_sec | TEXT | The same as this_date. |
| next_date | TIMESTAMP WITH TIME ZONE | The next date when this job will be executed. |
| next_sec | TEXT | The same as next_date. |
| total_time | INTERVAL | The period in which the job is executed. Unit: seconds. |

| Parameter | Type | Description |
|---|---|---|
| broken | TEXT | • Y: indicates that no attempt will be made to run this job.<br>• N: indicates that attempts will be made to run this job. |
| interval | TEXT | The interval at which the job is repeated. |
| failures | BIGINT | The number of times that the job has failed since the last successful execution. |
| what | TEXT | The job definition that runs when the job executes. The job definition appears as a PL/SQL code block. |
| nls_env | CHARACTER VARYING(4000) | The value is NULL. This parameter is supported for compatibility only. |
| misc_env | BYTEA | The value is NULL. This parameter is supported for compatibility only. |
| instance | NUMERIC | The value is 0. This parameter is supported for compatibility only. |

## 14.63 USER_OBJECTS

The USER_OBJECTS view provides the information about all objects that are owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which the object belongs. |
| object_name | TEXT | The name of the object. |
| object_type | TEXT | The type of the object. Valid values: INDEX , FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW. |
| status | CHARACTER VARYING | This parameter is supported for compatibility only. The value is VALID. |
| temporary | TEXT | • Y: indicates the object is temporary.<br>• N: indicates the object is not temporary. |

## 14.64 USER_PART_KEY_COLUMNS

The USER_PART_KEY_COLUMNS view provides the information about the key columns of partitioned tables in a database.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema where the table is located. |
| name | TEXT | The name of the table where the column is located. |
| object_type | CHARACTER(5) | This parameter is supported for compatibility only. The value is TABLE. |
| column_name | TEXT | The name of the column on which the key is defined. |
| column_position | INTEGER | The position of this column. For example, a value of 1 indicates the first column and a value of 2 indicates the second column. All columns follow the same rule. |

## 14.65 USER_PART_TABLES

The USER_PART_TABLES view provides the information about all partitioned tables in the database that are owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema where the table is located. |
| table_name | TEXT | The name of the table. |
| partitioning_type | TEXT | The partition type used to define table partitions. |
| subpartitioning_type | TEXT | The subpartition type used to define table subpartitions. |
| partition_count | BIGINT | The number of partitions in the table. |
| def_subpartition_count | INTEGER | The number of subpartitions in the table. |
| partitioning_key_count | INTEGER | The number of specified partition keys. |

| Parameter | Type | Description |
|---|---|---|
| subpartitioning_key_count | INTEGER | The number of specified subpartition keys. |
| status | CHARACTER VARYING(8) | This parameter is supported for compatibility only. The value is VALID. |
| def_tablespace_name | CHARACTER VARYING(30) | This parameter is supported for compatibility only. The value is NULL. |
| def_pct_free | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_pct_used | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_ini_trans | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_max_trans | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_initial_extent | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| def_min_extents | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| def_max_extents | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| def_pct_increase | CHARACTER VARYING(40) | This parameter is supported for compatibility only. The value is NULL. |
| def_freelists | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_freelist_groups | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| def_logging | CHARACTER VARYING(7) | This parameter is supported for compatibility only. The value is YES. |
| def_compression | CHARACTER VARYING(8) | This parameter is supported for compatibility only. The value is NONE. |
| def_buffer_pool | CHARACTER VARYING(7) | This parameter is supported for compatibility only. The value is DEFAULT. |
| ref_ptn_constraint_name | CHARACTER VARYING(30) | This parameter is supported for compatibility only. The value is NULL. |

| Parameter | Type | Description |
|---|---|---|
| interval | CHARACTER VARYING(1000) | This parameter is supported for compatibility only. The value is NULL. |

## 14.66 USER_QUEUES

The USER_QUEUES view provides the information about a queue on which the current user has usage permissions.

| Parameter | Type | Description |
|---|---|---|
| name | TEXT | The name of a queue. |
| queue_table | TEXT | The name of the queue table where the queue is located. |
| qid | OID | The system-assigned object ID of the queue. |
| queue_type | CHARACTER VARYING | The type of the queue. Valid values : EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, and NORMAL_QUEUE. |
| max_retries | NUMERIC | The maximum number of dequeuing attempts. |
| retrydelay | NUMERIC | The maximum time allowed between retries . |
| enqueue_enabled | CHARACTER VARYING | • YES: indicates that the queue allows enqueuing.<br>• NO: indicates that the queue does not allow enqueuing. |
| dequeue_enabled | CHARACTER VARYING | • YES: indicates that the queue allows dequeuing.<br>• NO: indicates that the queue does not allow dequeuing. |
| retention | CHARACTER VARYING | The number of seconds that a processed message is retained in the queue. |
| user_comment | CHARACTER VARYING | The user-defined comment. |
| network_name | CHARACTER VARYING | The name of the network where the queue is located. |

| Parameter | Type | Description |
|---|---|---|
| sharded | CHARACTER VARYING | • YES: indicates the queue is on a sharded network.<br>• NO: indicates the queue is not on a sharded network. |

## 14.67 USER_QUEUE_TABLES

The USER_QUEUE_TABLES view provides the information about all of the queue tables accessible by the current user.

| Parameter | Type | Description |
|---|---|---|
| queue_table | TEXT | The user-defined name of the queue table. |
| type | CHARACTER VARYING | The type of data stored in the queue table. |
| object_type | TEXT | The user-defined payload type. |
| sort_order | CHARACTER VARYING | The order in which the queue table is sorted. |
| recipients | CHARACTER VARYING | The value is SINGLE. |
| message_grouping | CHARACTER VARYING | The value is NONE. |
| compatible | CHARACTER VARYING | The release number of the PolarDB database compatible with Oracle. The queue table is compatible with this release. |
| primary_instance | NUMERIC | The value is 0. |
| secondary_instance | NUMERIC | The value is 0. |
| owner_instance | NUMERIC | The instance number of the instance to which the queue table belongs. |
| user_comment | CHARACTER VARYING | The comment added when the table was created. |
| secure | CHARACTER VARYING | • YES: indicates that the queue table is secure.<br>• NO: indicates that the queue table is not secure. |

## 14.68 USER_ROLE_PRIVS

The USER_ROLE_PRIVS view provides the information about the permissions that have been granted to the current user. A row is created for each role to which a user has been granted.

| Parameter | Type | Description |
|---|---|---|
| username | TEXT | The name of the user to which the role was granted. |
| granted_role | TEXT | The name of the role granted to the grantee. |
| admin_option | TEXT | YES: The role is granted with the admin option. NO: The role is granted with the other options other than admin. |
| default_role | TEXT | YES: The role is enabled when the grantee creates a session. |
| os_granted | CHARACTER VARYING(3) | This parameter is supported for compatibility only. The value is NO. |

## 14.69 USER_SEQUENCES

The USER_SEQUENCES view provides the information about all user-defined sequences that belong to the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which the sequence belongs. |
| sequence_name | TEXT | The name of the sequence. |
| min_value | NUMERIC | The lowest value that the server assigns to the sequence. |
| max_value | NUMERIC | The highest value that the server assigns to the sequence. |
| increment_by | NUMERIC | The value added to the current sequence number to create the next sequent number. |
| cycle_flag | CHARACTER VARYING | Specifies whether the sequence wraps if it reaches min_value or max_value. |

| Parameter | Type | Description |
|---|---|---|
| order_flag | CHARACTER VARYING | This parameter is supported for compatibility only. The value is Y. |
| cache_size | NUMERIC | The number of pre-allocated sequence numbers in memory. |
| last_number | NUMERIC | The value of the last sequence number saved to the disk. |

## 14.70 USER_SOURCE

The USER_SOURCE view provides the information about all programs owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which the program belongs. |
| name | TEXT | The name of the program. |
| type | TEXT | The type of the program. Valid values: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER. |
| line | INTEGER | The source code line number relative to a specified program. |
| text | TEXT | The line of source code text. |

## 14.71 USER_SUBPART_KEY_COLUMNS

The USER_SUBPART_KEY_COLUMNS view provides the information about key columns of those partitioned tables which are subpartitioned that belong to the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which the table belongs. |
| name | TEXT | The name of the table to which the column belongs. |
| object_type | CHARACTER(5) | This parameter is supported for compatibility only. The value is TABLE. |
| column_name | TEXT | The name of the column on which the key is defined. |

| Parameter | Type | Description |
|---|---|---|
| column_position | INTEGER | The ordinal position of this column. For example, a value of 1 indicates the first column and a value of 2 indicates the second column. All columns follow the same rule. |

## 14.72 USER_SYNONYMS

The USER_SYNONYMS view provides the information about all synonyms owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which the synonym belongs. |
| synonym_name | TEXT | The name of the synonym. |
| table_owner | TEXT | The username of the owner of the table on which the synonym is defined. |
| table_schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table on which the synonym is defined. |
| db_link | TEXT | The name of any associated database link. |

## 14.73 USER_TAB_COLUMNS

The USER_TAB_COLUMNS view provides the information about all columns in tables and views owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | CHARACTER VARYING | The name of the schema to which the table or the view belongs. |
| table_name | CHARACTER VARYING | The name of the table or view to which the column belongs. |
| column_name | CHARACTER VARYING | The name of the column. |
| data_type | CHARACTER VARYING | The data type of the column. |

| Parameter | Type | Description |
|---|---|---|
| data_length | NUMERIC | The length of the text column. |
| data_precision | NUMERIC | The precision of the NUMBER column . The precision is measured with the number of digits. |
| data_scale | NUMERIC | The scale of the NUMBER column. |
| nullable | CHARACTER(1) | Specifies whether the column is nullable or not. Valid values:<br>• Y: The column is nullable.<br>• N: The column cannot be null. |
| column_id | NUMERIC | The relative position of the column within the table. |
| data_default | CHARACTER VARYING | The default value that is assigned to the column. |

## 14.74 USER_TAB_PARTITIONS

The USER_TAB_PARTITIONS view provides the information about all of the partitions that are owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table. |
| composite | TEXT | • YES: The table is subpartitioned.<br>• NO: The table is not subpartitioned. |
| partition_name | TEXT | The name of the partition. |
| subpartition_count | BIGINT | The number of subpartitions in the partition. |
| high_value | TEXT | The high partitioning value specified in the CREATE TABLE statement. |
| high_value_length | INTEGER | The length of the high partitioning value. |
| partition_position | INTEGER | The ordinal position of this partition. For example, a value of 1 indicates the first partition and a value of 2 indicates the second partition. All positions follow the same rules. |

| Parameter | Type | Description |
|---|---|---|
| tablespace_name | TEXT | The name of the tablespace to which the subpartition belongs. |
| pct_free | NUMERIC | This parameter is supported for compatibil ity only. The value is 0. |
| pct_used | NUMERIC | This parameter is supported for compatibil ity only. The value is 0. |
| ini_trans | NUMERIC | This parameter is supported for compatibil ity only. The value is 0. |
| max_trans | NUMERIC | This parameter is supported for compatibil ity only. The value is 0. |
| initial_extent | NUMERIC | This parameter is supported for compatibil ity only. The value is NULL. |
| next_extent | NUMERIC | This parameter is supported for compatibil ity only. The value is NULL. |
| min_extent | NUMERIC | This parameter is supported for compatibil ity only. The value is 0. |
| max_extent | NUMERIC | This parameter is supported for compatibil ity only. The value is 0. |
| pct_increase | NUMERIC | This parameter is supported for compatibil ity only. The value is 0. |
| freelists | NUMERIC | This parameter is supported for compatibil ity only. The value is NULL. |
| freelist_groups | NUMERIC | This parameter is supported for compatibil ity only. The value is NULL. |
| logging | CHARACTER VARYING (7) | This parameter is supported for compatibil ity only. The value is YES. |
| compression | CHARACTER VARYING (8) | This parameter is supported for compatibil ity only. The value is NONE. |
| num_rows | NUMERIC | The same as pg_class.reltuples. |
| blocks | INTEGER | The same as pg_class.relpages. |
| empty_blocks | NUMERIC | This parameter is supported for compatibil ity only. The value is NULL. |
| avg_space | NUMERIC | This parameter is supported for compatibil ity only. The value is NULL. |

| Parameter | Type | Description |
|---|---|---|
| chain_cnt | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| avg_row_len | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| sample_size | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| last_analyzed | TIMESTAMP WITHOUT TIME ZONE | This parameter is supported for compatibility only. The value is NULL. |
| buffer_pool | CHARACTER VARYING (7) | This parameter is supported for compatibility only. The value is NULL. |
| global_stats | CHARACTER VARYING (3) | This parameter is supported for compatibility only. The value is YES. |
| user_stats | CHARACTER VARYING (3) | This parameter is supported for compatibility only. The value is NO. |
| backing_table | REGCLASS | The name of the partition backup table. |

## 14.75 USER_TAB_SUBPARTITIONS

The USER_TAB_SUBPARTITIONS view provides information about all of the subpartitions owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table. |
| partition_name | TEXT | The name of the partition. |
| subpartition_name | TEXT | The name of the subpartition. |
| high_value | TEXT | The high subpartitioning value specified in the CREATE TABLE statement. |
| high_value_length | INTEGER | The length of the high subpartitioning value . |
| subpartition_position | INTEGER | The ordinal position of this subpartition. For example, a value of 1 indicates the first subpartition and a value of 2 indicates the second subpartition. All positions of subpartitions follow the same rule. |

| Parameter | Type | Description |
|---|---|---|
| tablespace_name | TEXT | The name of the tablespace to which the subpartition belongs. |
| pct_free | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| pct_used | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| ini_trans | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| max_trans | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| initial_extent | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| next_extent | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| min_extent | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| max_extent | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| pct_increase | NUMERIC | This parameter is supported for compatibility only. The value is 0. |
| freelists | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| freelist_groups | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| logging | CHARACTER VARYING (7) | This parameter is supported for compatibility only. The value is YES. |
| compression | CHARACTER VARYING (8) | This parameter is supported for compatibility only. The value is NONE. |
| num_rows | NUMERIC | The same as pg_class.reltuples. |
| blocks | INTEGER | The same as pg_class.relpages. |
| empty_blocks | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| avg_space | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |

| Parameter | Type | Description |
|---|---|---|
| chain_cnt | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| avg_row_len | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| sample_size | NUMERIC | This parameter is supported for compatibility only. The value is NULL. |
| last_analyzed | TIMESTAMP WITHOUT TIME ZONE | This parameter is supported for compatibility only. The value is NULL. |
| buffer_pool | CHARACTER VARYING (7) | This parameter is supported for compatibility only. The value is NULL. |
| global_stats | CHARACTER VARYING (3) | This parameter is supported for compatibility only. The value is YES. |
| user_stats | CHARACTER VARYING (3) | This parameter is supported for compatibility only. The value is NO. |
| backing_table | REGCLASS | The name of the partition backup table. |

## 14.76 USER_TABLES

The USER_TABLES view provides the information about all tables owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which the table belongs. |
| table_name | TEXT | The name of the table. |
| tablespace_name | TEXT | The name of the tablespace to which the table belongs if this tablespace is not the default tablespace. |
| status | CHARACTER VARYING (5) | This parameter is supported for compatibility only. The value is VALID. |
| temporary | CHARACTER(1) | • Y: specifies that the table is a temporary table.<br>• N: specifies that the table is not a temporary table. |

## 14.77 USER_TRIGGERS

The USER_TRIGGERS view provides the information about all triggers on tables owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which the trigger belongs. |
| trigger_name | TEXT | The name of the trigger. |
| trigger_type | TEXT | The type of the trigger. Valid values:<br>• BEFORE ROW<br>• BEFORE STATEMENT<br>• AFTER ROW<br>• AFTER STATEMENT |
| triggering_event | TEXT | The event that fires the trigger. |
| table_owner | TEXT | The username of the owner of the table on which the trigger is defined. |
| base_object_type | TEXT | This parameter is supported for compatibility only. The value is TABLE. |
| table_name | TEXT | The name of the table on which the trigger is defined. |
| referencing_names | TEXT | This parameter is supported for compatibility only. The value is REFERENCING. NEW AS NEW OLD AS OLD. |
| status | TEXT | Specifies whether the trigger is enabled (VALID) or disabled ( NOTVALID). |
| description | TEXT | This parameter is supported for compatibility only. |
| trigger_body | TEXT | The body of the trigger. |
| action_statement | TEXT | The SQL statement that is executed when the trigger fires. |

## 14.78 USER_TYPES

The USER_TYPES view provides the information about all object types owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema in which the type is defined. |
| type_name | TEXT | The name of the type. |
| type_oid | OID | The object identifier (OID) of the type. |
| typecode | TEXT | The typecode of the type. Valid values:<br>• OBJECT<br>• COLLECTION<br>• OTHER |
| attributes | INTEGER | The number of properties in the type. |

## 14.79 USER_USERS

The USER_USERS view provides the information about the current user.

| Parameter | Type | Description |
|---|---|---|
| username | TEXT | The username of the user. |
| user_id | OID | The ID of the user. |
| account_status | CHARACTER VARYING (32) | The current status of the account. Valid values:<br>• OPEN<br>• EXPIRED<br>• EXPIRED(GRACE)<br>• EXPIRED & LOCKED<br>• EXPIRED & LOCKED(TIMED)<br>• EXPIRED(GRACE) & LOCKED<br>• EXPIRED(GRACE) & LOCKED(TIMED)<br>• LOCKED<br>• LOCKED(TIMED)<br>Uses the edb_get_role_status(role_id) function to get the current status of the account. |

| Parameter | Type | Description |
|---|---|---|
| lock_date | TIMESTAMP WITHOUT TIME ZONE | If the account status is set to LOCKED, the lock_date parameter displays the date and time when the account is locked |
| expiry_date | TIMESTAMP WITHOUT TIME ZONE | The expiration date of the account. |
| default_tablespace | TEXT | The default tablespace associated with the account. |
| temporary_ tablespace | CHARACTER VARYING (30) | This parameter is supported for compatibil ity only. The value is '' (an empty string). |
| created | TIMESTAMP WITHOUT TIME ZONE | This parameter is supported for compatibil ity only. The value is NULL. |
| initial_rsrc_consume r_group | CHARACTER VARYING (30) | This parameter is supported for compatibil ity only. The value is NULL. |
| external_name | CHARACTER VARYING (4000) | This parameter is supported for compatibil ity only. The value is NULL. |

## 14.80 USER_VIEW_COLUMNS

The USER_VIEW_COLUMNS view provides the information about all columns in views owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | CHARACTER VARYING | The name of the schema to which the view belongs. |
| view_name | CHARACTER VARYING | The name of the view. |
| column_name | CHARACTER VARYING | The name of the column. |
| data_type | CHARACTER VARYING | The data type of the column. |
| data_length | NUMERIC | The length of the text column. |
| data_precision | NUMERIC | The precision of the NUMBER column . The precision is measured with the number of digits. |
| data_scale | NUMERIC | The scale of the NUMBER column. |

| Parameter | Type | Description |
|---|---|---|
| nullable | CHARACTER(1) | Specifies whether the column is nullable or not. Valid values:<br>• Y: The column is nullable.<br>• N: The column cannot be null. |
| column_id | NUMERIC | The relative position of the column within the view. |
| data_default | CHARACTER VARYING | The default value that is assigned to the column. |

## 14.81 USER_VIEWS

The USER_VIEWS view provides the information about all views owned by the current user.

| Parameter | Type | Description |
|---|---|---|
| schema_name | TEXT | The name of the schema to which the view belongs. |
| view_name | TEXT | The name of the view. |
| text | TEXT | The SELECT statement that defines the view. |

## 14.82 V$VERSION

The V$VERSION view provides the information about the product compatibility.

| Parameter | Type | Description |
|---|---|---|
| banner | TEXT | The product compatibility informatio n. |

## 14.83 PRODUCT_COMPONENT_VERSION

The PRODUCT_COMPONENT_VERSION view provides the version information about the product version compatibility.

| Parameter | Type | Description |
|---|---|---|
| product | CHARACTER VARYING(74) | The name of the product. |
| version | CHARACTER VARYING(74) | The version number of the product. |

| Parameter | Type | Description |
|---|---|---|
| status | CHARACTER VARYING(74) | This parameter is provided for compatibility. The value is Available. |

# 15 Table partitioning

## 15.1 Overview

In a partitioned table, a logically large table is divided into smaller physical pieces. This document discusses the aspects of table partitioning that are compatible with Oracle databases and supported by POLARDB compatible with Oracle.

Partitioning can provide the following benefits:

- Query performance can be significantly improved in specific situations, particularly when the most frequently accessed rows of the table are in a single partition or small number of partitions. Partitioning allows you to omit the partition column from the front of an index, reducing index size and making it more likely that the frequently used parts of the index fits in memory.
- You can experience improved performance when accessing (query or update) a large percentage of a single partition. This is because the server will perform a sequential scan of the partition instead of using an index and random access reads scattered across the whole table.
- A bulk load (or unload) can be implemented by adding or removing partitions, if you plan this requirement into the partitioning design. ALTER TABLE is much faster than a bulk operation. It also helps to avoid the VACUUM overhead caused by a bulk DELETE.
- You can migrate seldom-used data to less-expensive (or slower) storage media.

We recommend table partitioning only when a table is very large. The exact point at which a table will benefit from partitioning depends on the application. We recommend that the size of the table exceeds the physical memory of the database server.

## 15.2 Select a partitioning type

This topic describes how to select a partitioning type.

When you create a partitioned table, you can specify LIST or RANGE partitioning rules. The partitioning rules provide a set of constraints that define the data stored in each partition. When new rows are added to the partitioned table, the server uses the partitioning rules to determine which partition will contain each row.

POLARDB compatible with Oracle can also use partitioning rules to enable partition pruning , improving performance when responding to user queries. When selecting a partitioning type and partition keys for a table, you need to consider how the data that is stored in the table will be queried, and include frequently queried columns in the partitioning rules.

**Partitioning types**

- List partitioning

  When creating a list-partitioned table, you must specify a single partition key column. When you add a new row to the table, the server compares the key values specified in the partitioning rule to the corresponding column within the row. If the column value matches a value in the partitioning rule, the row is stored in the partition named in the rule.
- Range partitioning

  When creating a range-partitioned table, you must specify one or more partition key columns. When you add a new row to the table, the server compares the value of the partition key column (or columns) to the corresponding column (or columns) in the table entry. If the column values satisfy the conditions specified in the partitioning rule, the row is stored in the partition named in the rule.
- Subpartitioning

  Subpartitioning breaks a partitioned table into smaller subsets that can be stored on the same server. A table is typically subpartitioned by a different set of columns, and can be of a different subpartitioning type other than that of the parent partition. If one partition is subpartitioned, then each partition must include a minimum of at least one subpartiti on.

  If a table is subpartitioned, no data will be stored in any of the partitions. The data will be instead stored in the corresponding subpartitions.

# 15.3 Use partition pruning

The query planner of POLARDB compatible with Oracle uses partition pruning to compute an effective plan to locate a row that matches the conditions specified in the WHERE clause of a SELECT statement.

The partition pruning mechanism uses the following two optimization methods:

- Constraint exclusion

- Fast pruning

Partition pruning methods limit the search for data to only the partitions where the values for which you are searching can reside. The preceding two pruning methods remove partitions from a query execution plan to increase performance.

The difference between fast pruning and constraint exclusion is that fast pruning understands the relationship between the partitions in an Oracle-partitioned table, whereas exclusion constraint does not. For example, when a query searches for a specific value in a list-partitioned table, fast pruning involves only searching a specific partition. However, constraint exclusion must examine the constraints defined for each partition. Fast pruning occurs early in the planning process to reduce the number of partitions that the planner must consider, whereas constraint exclusion occurs late in the planning process.

## Use constraint exclusion

The constraint_exclusion parameter is used to control constraint exclusion. The value of the constraint_exclusion parameter can be on, off, or partition. To enable constraint exclusion, you must set the constraint_exclusion parameter to either partition or on. By default, the parameter is set to partition.

**Note:**

For more information about constraint exclusion, see Partitioning.

When constraint exclusion is enabled, the server examines the constraints defined for each partition to determine whether the partition can satisfy a query.

When you run a SELECT statement that does not contain a WHERE clause, the query planner must recommend an execution plan that searches through the entire table. When you run a SELECT statement that contains a WHERE clause, the query planner determines in which partition the row can be stored, and sends query fragments to that partition. This prunes the partitions that cannot contain the row from the execution plan. If you are not using partitioned tables, disabling constraint exclusion can improve performance.

## Use fast pruning

Like constraint exclusion, fast pruning can only optimize queries that contain a WHERE (or join) clause, and only when the qualifiers in the WHERE clause match a specific form. In both cases, the query planner will avoid searching for data within partitions that cannot hold the data required by the query.

Fast pruning is controlled by a boolean configuration parameter named edb partition pruning. If edb partition pruning is ON, POLARDB compatible with Oracle will fast prune specific queries. If edb partition pruning is OFF, POLARDB compatible with Oracle will disable fast pruning.

Note that fast pruning cannot optimize queries against subpartitioned tables or optimize queries against range-partitioned tables that are partitioned on more than one column.

For LIST partitioned tables, POLARDB compatible with Oracle can fast prune queries that contain a WHERE clause that constrains a partitioning column to a literal value. For example, given the following LIST partitioned table:

```
CREATE TABLE sales_hist(..., country text, ...)
    PARTITION BY LIST(country) (
    PARTITION americas VALUES('US', 'CA', 'MX'),
    PARTITION europe VALUES('BE', 'NL', 'FR'),
    PARTITION asia VALUES('JP', 'PK', 'CN'),
    PARTITION others VALUES(DEFAULT)
)
```

Fast pruning can reason about WHERE clauses such as:

```
WHERE country = 'US' WHERE country IS NULL;
```

Given the first WHERE clause, fast pruning can eliminate partitions europe, asia, and others because these partitions cannot hold rows that satisfy the qualifier: WHERE country = 'US '. Given the second WHERE clause, fast pruning can eliminate partitions americas, europe , and asia because these partitions cannot hold rows where country IS NULL. The operator specified in the WHERE clause must be an equal sign (=) or the equality operator suitable for the data type of the partitioning column.

For a range-partitioned table, POLARDB compatible with Oracle can fast prune queries that contain a WHERE clause that constrains a partitioning column to a literal value. The operator may be any of the following: greater than (>), greater than or equal to (>=), less than (<), and less than or equal to (<=).

Fast pruning will also reason about more complex expressions, including AND and BETWEEN operators, such as:

```
WHERE size > 100 AND size <= 200 WHERE size BETWEEN 100 AND 200
```

However, fast pruning cannot prune based on expressions that include OR or IN. For example, when querying the following RANGE partitioned table:

```
CREATE TABLE boxes(id int, size int, color text)
```

```
 PARTITION BY RANGE(size)
(
  PARTITION small VALUES LESS THAN(100),
  PARTITION medium VALUES LESS THAN(200),
  PARTITION large VALUES LESS THAN(300)
)
```

Fast pruning can reason about WHERE clauses, such as:

```
WHERE size > 100    -- scan partitions 'medium' and 'large'
WHERE size >= 100   -- scan partitions 'medium' and 'large'
WHERE size = 100    -- scan partition 'medium'
WHERE size <= 100   -- scan partitions 'small' and 'medium'
WHERE size < 100    -- scan partition 'small'
WHERE size > 100 AND size < 199    -- scan partition 'medium'
WHERE size BETWEEN 100 AND 199     -- scan partition 'medium'
WHERE color = 'red' AND size = 100  -- scan 'medium'
WHERE color = 'red' AND (size > 100 AND size < 199) -- scan 'medium'
```

In each case, fast pruning requires that the qualifier be a partitioning column and literal value (or IS NULL/IS NOT NULL). Note that fast pruning can also optimize DELETE and UPDATE statements containing WHERE clauses of the forms described above.

## 15.4 Example - partition pruning

This topic provides an example about how to use partition pruning.

### Examples

The EXPLAIN statement displays the execution plan of a statement. You can use the EXPLAIN statement to confirm that POLARDB compatible with Oracle is pruning partitions from the execution plan of a query. To demonstrate the efficiency of partition pruning, first create a simple table:

```
CREATE TABLE sales
(
 dept_no    number,
 part_no    varchar2,
 country    varchar2(20),
 date       date,
 amount     number
)
```

```
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Then, perform a constrained query that includes the EXPLAIN statement:

```
EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'INDIA';
```

The resulting query plan shows that the server will only scan the sales_asia partition, in which a row with a country value of INDIA can be stored:

```
edb=# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'INDIA';
                QUERY PLAN
---------------------------------------------------

 Append
   ->  Seq Scan on sales
         Filter: ((country)::text = 'INDIA'::text)
   ->  Seq Scan on sales_asia
         Filter: ((country)::text = 'INDIA'::text)
(5 rows)
```

If you perform a query that searches for a row that matches a value not included in the partition key:

```
EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE dept_no = '30';
```

The resulting query plan shows that the server must search through all of the partitions to locate the rows that satisfy the query:

```
edb=# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE dept_no = '30';
            QUERY PLAN
-----------------------------------------
 Append
   ->  Seq Scan on sales
         Filter: (dept_no = 30::numeric)
   ->  Seq Scan on sales_europe
         Filter: (dept_no = 30::numeric)
   ->  Seq Scan on sales_asia
         Filter: (dept_no = 30::numeric)
   ->  Seq Scan on sales_americas
         Filter: (dept_no = 30::numeric)
(9 rows)
```

Constraint exclusion also applies when querying subpartitioned tables:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date) SUBPARTITION BY LIST (country)
(
  PARTITION "2011" VALUES LESS THAN('01-JAN-2012')
  (
   SUBPARTITION europe_2011 VALUES ('ITALY', 'FRANCE'),
   SUBPARTITION asia_2011 VALUES ('PAKISTAN', 'INDIA'),
   SUBPARTITION americas_2011 VALUES ('US', 'CANADA')
  ),
  PARTITION "2012" VALUES LESS THAN('01-JAN-2013')
  (
   SUBPARTITION europe_2012 VALUES ('ITALY', 'FRANCE'),
   SUBPARTITION asia_2012 VALUES ('PAKISTAN', 'INDIA'),
   SUBPARTITION americas_2012 VALUES ('US', 'CANADA')
  ),
  PARTITION "2013" VALUES LESS THAN('01-JAN-2014')
  (
   SUBPARTITION europe_2013 VALUES ('ITALY', 'FRANCE'),
   SUBPARTITION asia_2013 VALUES ('PAKISTAN', 'INDIA'),
   SUBPARTITION americas_2013 VALUES ('US', 'CANADA')
  )
);
```

When you query the table, the query planner prunes any partitions or subpartitions from the search path that cannot contain the result set:

```
edb=# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'US' AND date = 'Dec
12, 2012';
                                QUERY PLAN
-----------------------------------------------------------------------------
 Append
   ->  Seq Scan on sales
         Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12 00:00:00'::timestamp
without time zone))
   ->  Seq Scan on sales_2012
         Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12 00:00:00'::timestamp
without time zone))
   ->  Seq Scan on sales_americas_2012
         Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12 00:00:00'::timestamp
without time zone))
(7 rows)
```

## 15.5 Partitioning commands compatible with Oracle databases

### 15.5.1 CREATE TABLE... PARTITION BY

You can use the PARTITION BY clause of the CREATE TABLE command to create a partitioned table. Data in this partitioned table is distributed among one or more partitions (and subpartitions).

**Overview**

The CREATE TABLE command syntax has the following four forms:

- List partitioning syntax

  The first form is to create a list-partitioned table:

  ```
  CREATE TABLE [ schema. ]table_name table_definition PARTITION BY
      LIST(column)
      [SUBPARTITION BY {RANGE|LIST} (column[, column ]...)]
      (list_partition_definition[, list_partition_definition]...) ;
  ```

  Where list_partition_definition is:

  ```
  PARTITION [partition_name]
  VALUES (value[, value]...) [TABLESPACE tablespace_name] [(subpartition, ...)]
  ```

- Range partitioning syntax

  The second form is to create a range-partitioned table:

  ```
  CREATE TABLE [ schema. ]table_name
     table_definition
     PARTITION BY RANGE(column[, column ]...)
     [SUBPARTITION BY {RANGE|LIST} (column[, column ]...)]
     (range_partition_definition[, range_partition_definition]...) ;
  ```

  Where range_partition_definition is:

  ```
  PARTITION [partition_name]
    VALUES LESS THAN (value[, value]...)
    [TABLESPACE tablespace_name]
    [(subpartition, ...)]
  ```

- Subpartitioning syntax

  subpartition may be one of the following two types:

  ```
  {list_subpartition | range_subpartition}
  ```

  Where list_subpartition is:

  ```
  SUBPARTITION [subpartition_name] VALUES (value[, value]...)
  [TABLESPACE tablespace_name]
  ```

  Where range_subpartition is:

  ```
  SUBPARTITION [subpartition_name]
  VALUES LESS THAN (value[, value]...)
  [TABLESPACE tablespace_name]
  ```

**Description**

The CREATE TABLE... PARTITION BY command creates a table that has one or multiple partitions. Each partition may have one or multiple subpartitions. The number of defined partitions is not limited. If you include the PARTITION BY clause, you must specify a minimum of one partitioning rule. The resulting table is owned by the user who creates the table.

Use the PARTITION BY LIST clause to divide a table into partitions based on the values entered in a specified column. Each partitioning rule must specify a minimum of one literal value. The number of values you may specify is not limited. Include a rule that specifies a matching value of DEFAULT to direct any un-qualified rows to the specified partition.

Use the PARTITION BY RANGE clause to specify boundary rules based on which partitions are created. Each partitioning rule must contain at least one column of a data type that has two operators (for example, a greater-than or equal to operator, and a less-than operator ). Range boundaries are evaluated based on a LESS THAN clause and are non-inclusive . A date boundary of January 1, 2013 only includes the date values that fall on or before December 31, 2012.

Range partitioning rules must be specified in ascending order. If INSERT commands store rows with values that exceed the top boundary of a range-partitioned table, the commands will fail. However, commands will not fail if the partitioning rules include a boundary rule that specifies a value of MAXVALUE. If you do not include a MAXVALUE rule, any row that exceeds the maximum limit specified by the boundary rules will cause an error.

Use the TABLESPACE keyword to specify the name of a tablespace in which a partition or subpartition will reside. If you do not specify a tablespace, the partition or subpartition will be created in the default tablespace.

If you use the CREATE TABLE syntax to create an index on a partitioned table, the index will be created on each partition or subpartition.

If the table definition includes the SUBPARTITION BY clause, each partition in the table will have a minimum of one subpartition. Each subpartition can be explicitly defined or system-defined.

If the subpartition is system-defined, the server-generated subpartition will reside in the default tablespace, and the subpartition name will be assigned by the server. The server will create:

- A DEFAULT subpartition if the SUBPARTITION BY clause specifies LIST.
- A MAXVALUE subpartition if the SUBPARTITION BY clause specifies RANGE.

A subpartition name generated by the server is a combination of the partition name and a unique identifier. You can query the ALL_TAB_SUBPARTITIONS table to view a complete list of subpartition names.

**Parameters**

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the table to be created. |
| table_definition | The column names, data types, and constraint information as described in the PostgreSQL core documentation for the CREATE TABLE statement is available at CREATE TABLE. |
| partition_name | The name of the partition to be created. Partition names must be unique among all partitions and subpartitions, and must follow the naming conventions for object identifiers. |
| subpartition_name | The name of the subpartition to be created. Subpartition names must be unique among all partitions and subpartitions, and must follow the naming conventions for object identifiers. |
| column | The name of the column on which the partitioning rules are based. Each row will be stored in a partition that corresponds to the value of the specified column. |

| Parameter | Description |
|---|---|
| (value[, value ]...) | Use `value` to specify a quoted literal value (or a list of literal values separated by commas) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but the number of values specified in a rule is not limited. `value` may be `null`, `default` (if specifying a `LIST` partition), or `maxvalue` (if specifying a `RANGE` partition).<br>When you specify rules for a list-partitioned table, include the `DEFAULT` keyword in the last partition rule to direct any unmatched rows to the specified partition. If you do not include a value of `DEFAULT`, any `INSERT` statement that attempts to add a row that does not match the specified rules of at least one partition will fail and return an error.<br>When you specify rules for a range-partitioned table, include the `MAXVALUE` keyword in the last partition rule to direct any un-categorized rows to the specified partition. If you do not include a `MAXVALUE` partition, any `INSERT` statement that attempts to add a row where the partition key is greater than the highest value specified will fail and return an error. |
| tablespace _name | The name of the tablespace in which the partition or subpartition resides. |

**Example - PARTITION BY LIST**

The following example uses the PARTITION BY LIST clause to create a partitioned table named sales. The sales table stores information in three partitions (europe, asia, and americas):

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

The resulting table is partitioned based on the value specified in the country column:

```
acctg=# SELECT partition_name, high_value from ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 americas       | 'US', 'CANADA'
 asia           | 'INDIA', 'PAKISTAN'
 europe         | 'FRANCE', 'ITALY'
(3 rows)
```

- Rows with a value of US or CANADA in the country column are stored in the americas partition.
- Rows with a value of INDIA or PAKISTAN in the country column are stored in the asia partition.
- Rows with a value of FRANCE or ITALY in the country column are stored in the europe partition.

The server evaluates the following statement based on the partitioning rules and stores the row in the europe partition:

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012', '650000');
```

### Example - PARTITION BY RANGE

The following example uses the PARTITION BY RANGE clause to create a partitioned table named sales. The sales table stores information in four partitions (q1_2012, q2_2012, q3_2012, and q4_2012).

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012
    VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012
    VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
);
```

The resulting table is partitioned based on the value specified in the date column:

```
acctg=# SELECT partition_name, high_value from ALL_TAB_PARTITIONS;
 partition_name |  high_value
----------------+---------------
 q4_2012        | '2013-Jan-01'
 q3_2012        | '2012-Oct-01'
 q2_2012        | '2012-Jul-01'
 q1_2012        | '2012-Apr-01'
(4 rows)
```

- Rows with a value in the date column before April 1, 2012 are stored in the q1_2012 partition.
- Rows with a value in the date column before July 1, 2012 are stored in the q2_2012 partition.
- Rows with a value in the date column before October 1, 2012 are stored in the q3_2012 partition.
- Rows with a value in the date column before January 1, 2013 are stored in the q4_2012 partition.

The server evaluates the following statement based on the partitioning rules and stores the row in the q3_2012 partition:

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012', '650000');
```

### Example - PARTITION BY RANGE, SUBPARTITION BY LIST

The following example creates a partitioned table (sales) that is first partitioned by using the transaction date. Then, the range partitions (q1_2012, q2_2012, q3_2012, and q4_2012) are list-partitioned by using the value of the country column.

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST(country)
  (
    PARTITION q1_2012
      VALUES LESS THAN('2012-Apr-01')
      (
        SUBPARTITION q1_europe VALUES ('FRANCE', 'ITALY'),
        SUBPARTITION q1_asia VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION q1_americas VALUES ('US', 'CANADA')
       ),
  PARTITION q2_2012
```

```
    VALUES LESS THAN('2012-Jul-01')
      (
        SUBPARTITION q2_europe VALUES ('FRANCE', 'ITALY'),
        SUBPARTITION q2_asia VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION q2_americas VALUES ('US', 'CANADA')
       ),
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01')
      (
        SUBPARTITION q3_europe VALUES ('FRANCE', 'ITALY'),
        SUBPARTITION q3_asia VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION q3_americas VALUES ('US', 'CANADA')
       ),
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
      (
        SUBPARTITION q4_europe VALUES ('FRANCE', 'ITALY'),
        SUBPARTITION q4_asia VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION q4_americas VALUES ('US', 'CANADA')
       )
);
```

The table created by using this statement has four partitions. Each partition has three subpartitions:

```
acctg=# SELECT subpartition_name, high_value, partition_name FROM ALL_TAB_SU
BPARTITIONS;
subpartition_name | high_value | partition_name     +    +
q4_asia    | 'INDIA', 'PAKISTAN' | q4_2012
q4_europe   | 'FRANCE', 'ITALY' | q4_2012
SUBPARTITION q4_ SUBPARTITION q4_ SUBPARTITION q4_
q4_americas    | 'US', 'CANADA'    | q4_2012
q3_americas    | 'US', 'CANADA'    | q3_2012
q3_asia       | 'INDIA', 'PAKISTAN'    | q3_2012
q3_europe    | 'FRANCE', 'ITALY'    | q3_2012
q2_americas    | 'US', 'CANADA'    | q2_2012
q2_asia       | 'INDIA','PAKISTAN'    | q2_2012
q2_europe    | 'FRANCE', 'ITALY'    | q2_2012
q1_americas    | 'US', 'CANADA'    | q1_2012
q1_asia       | 'INDIA', 'PAKISTAN'    | q1_2012
q1_europe    | 'FRANCE', 'ITALY'    | q1_2012
(12 rows)
```

When a row is added to this table, the value in the date column is compared with the values specified in the range partitioning rules. The server selects the partition in which the row will reside. The value in the country column is then compared with the values specified in the list subpartitioning rules. When the server locates a match for the value, the row is stored in the corresponding subpartition.

Any row added to the table is stored in a subpartition. Therefore, all partitions contain no data.

The server evaluates the following statement based on the partitioning and subpartitioning rules and stores the row in the q3_europe partition:

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012', '650000');
```

## 15.5.2 ALTER TABLE... ADD PARTITION

The ALTER TABLE... ADD PARTITION command adds a partition to an existing partitioned table.

### Overview

You can use the ALTER TABLE... ADD PARTITION command to add a partition to an existing partitioned table. Syntax:

```
ALTER TABLE table_name ADD PARTITION partition_definition;
```

Where partition_definition is:

```
{list_partition | range_partition}
```

and list_partition is:

```
PARTITION [partition_name]
  VALUES (value[, value]...)
  [TABLESPACE tablespace_name]
  [(subpartition, ...)]
```

and range_partition is:

```
PARTITION [partition_name]
  VALUES LESS THAN (value[, value]...)
  [TABLESPACE tablespace_name]
  [(subpartition, ...)]
```

Where subpartition is:

```
{list_subpartition | range_subpartition}
```

and list_subpartition is:

```
SUBPARTITION [subpartition_name]
  VALUES (value[, value]...)
  [TABLESPACE tablespace_name]
```

and range_subpartition is:

```
SUBPARTITION [subpartition_name ]
  VALUES LESS THAN (value[, value]...)
```

```
[TABLESPACE tablespace_name]
```

**Description**

The ALTER TABLE... ADD PARTITION command adds a partition to an existing partitioned table. The number of defined partitions in a partitioned table is not limited.

New partitions must be of the same type (LIST or RANGE) as existing partitions. The partitioning rules for new partitions must reference the same column specified in the partitioning rules that define the existing partitions.

You cannot use the ALTER TABLE... ADD PARTITION statement to add partitions to tables that have a MAXVALUE or DEFAULT rule. Alternatively, you can use the ALTER TABLE... SPLIT PARTITION statement to split an existing partition. This allows you to effectively increase the number of partitions in a table.

RANGE partitions must be specified in ascending order. You cannot add a new partition that precedes existing partitions in a RANGE partitioned table.

Include the TABLESPACE clause to specify a tablespace in which a new partition will reside. If you do not specify a tablespace, the partition will be created in the default tablespace.

If the table is indexed, the index will be created on the new partition. To use the ALTER TABLE... ADD SUBPARTITION command, you must be the table owner, or have superuser (or administrative) privileges.

**Parameters**

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the partitioned table. |
| partition_name | The name of the partition to be created. Partition names must be unique among all partitions and subpartitions, and must follow the naming conventions for object identifiers. |
| subpartition_name | The name of the subpartition to be created. Subpartition names must be unique among all partitions and subpartitions, and must follow the naming conventions for object identifiers. |

| Parameter | Description |
|---|---|
| (value[, value ]...) | Use value to specify a quoted literal value (or a list of literal values separated by commas) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but the number of values specified in a rule is not limited. value may be null, default (if specifying a LIST partition), or maxvalue (if specifying a RANGE partition).<br>For more information about creating a default or maxvalue partition, see Handle stray values in a LIST or RANGE partitioned table. |
| tablespace_name | The name of the tablespace in which the partition or subpartition resides. |

**Example - add a partition to a LIST partitioned table**

The following example adds a partition to a list-partitioned table named sales. Run the following command to create the sales table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

The table contains three partitions (americas, asia, and europe):

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 americas       | 'US', 'CANADA'
 asia           | 'INDIA', 'PAKISTAN'
 europe         | 'FRANCE', 'ITALY'
(3 rows)
```

The following command adds a partition named east_asia to the sales table:

```
ALTER TABLE sales ADD PARTITION east_asia
  VALUES ('CHINA', 'KOREA');
```

After this command is called, the table contains the east_asia partition:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
```

```
 partition_name |    high_value
----------------+---------------------
 east_asia      | 'CHINA', 'KOREA'
 americas       | 'US', 'CANADA'
 asia           | 'INDIA', 'PAKISTAN'
 europe         | 'FRANCE', 'ITALY'
(4 rows)
```

**Example - add a partition to a RANGE partitioned table**

The following example adds a partition to a range-partitioned table named sales:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012
    VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012
    VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
);
```

The table contains four partitions (q1_2012, q2_2012, q3_2012, and q4_2012):

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name | high_value
----------------+---------------
 q4_2012        | '2013-Jan-01'
 q3_2012        | '2012-Oct-01'
 q2_2012        | '2012-Jul-01'
 q1_2012        | '2012-Apr-01'
(4 rows)
```

The following command adds a partition named q1_2013 to the sales table:

```
ALTER TABLE sales ADD PARTITION q1_2013
  VALUES LESS THAN('01-APR-2013');
```

After this command is called, the table contains the q1_2013 partition:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name | high_value
----------------+---------------
 q1_2012        | '2012-Apr-01'
 q2_2012        | '2012-Jul-01'
 q3_2012        | '2012-Oct-01'
 q4_2012        | '2013-Jan-01'
 q1_2013        | '01-APR-2013'
```

```
(5 rows)
```

## 15.5.3 ALTER TABLE... ADD SUBPARTITION

The ALTER TABLE... ADD SUBPARTITION command adds a subpartition to an existing subpartitioned partition.

**Overview**

You can use the ALTER TABLE... ADD SUBPARTITION command to add a subpartition to an existing subpartitioned table. Syntax:

```
ALTER TABLE table_name MODIFY PARTITION partition_name
    ADD SUBPARTITION subpartition_definition;
```

Where subpartition_definition is:

```
{list subpartition | range subpartition}
```

and list_subpartition is:

```
SUBPARTITION [subpartition_name]
  VALUES (value[, value]...)
  [TABLESPACE tablespace_name]
```

and range_subpartition is:

```
SUBPARTITION subpartition_name
  VALUES LESS THAN (value[, value]...)
  [TABLESPACE tablespace_name]
```

**Description**

The ALTER TABLE... ADD SUBPARTITION command adds a subpartition to an existing subpartitioned partition. The number of defined subpartitions is not limited.

New subpartitions must be of the same type (LIST or RANGE) as existing subpartitions. The subpartitioning rules for new subpartitions must reference the same column specified in the subpartitioning rules that define the existing subpartitions.

You cannot use the ALTER TABLE... ADD SUBPARTITION statement to add subpartitions to tables that have a MAXVALUE or DEFAULT rule. Alternatively, you can use the ALTER TABLE... SPLIT SUBPARTITION statement to split an existing subpartition. This effectively allows you to add a subpartition to a table.

You cannot add a new subpartition that precedes existing subpartitions in a range-partitioned table. Range subpartitions must be specified in ascending order.

Include the TABLESPACE clause to specify a tablespace in which a new subpartition will reside. If you do not specify a tablespace, the subpartition will be created in the default tablespace.

If the table is indexed, the index will be created on the new subpartition.

To use the ALTER TABLE... ADD SUBPARTITION command, you must be the table owner, or have superuser (or administrative) privileges.

### Parameters

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the partitioned table in which the subpartition resides. |
| partition_name | The name of the partition in which the new subpartition will reside. |
| subpartition_name | The name of the subpartition to be created. Subpartition names must be unique among all partitions and subpartitions, and must follow the naming conventions for object identifiers. |
| (value[, value ]...) | Use value to specify a quoted literal value (or a list of literal values separated by commas) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but the number of values specified in a rule is not limited. value may be null, default (if specifying a LIST partition), or maxvalue (if specifying a RANGE partition).<br>For more information about creating a DEFAULT or MAXVALUE partition, see Handle stray values in a LIST or RANGE partitioned table. |
| tablespace_name | The name of the tablespace in which the subpartition resides. |

### Example - add a subpartition to a LIST-RANGE partitioned table

The following example adds a RANGE subpartition to the list-partitioned sales table. The sales table is created by using the following command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
```

```
 SUBPARTITION BY RANGE(date)
(
 PARTITION europe VALUES('FRANCE', 'ITALY')
  (
   SUBPARTITION europe_2011
    VALUES LESS THAN('2012-Jan-01'),
   SUBPARTITION europe_2012
    VALUES LESS THAN('2013-Jan-01')
  ),
 PARTITION asia VALUES('INDIA', 'PAKISTAN')
  (
   SUBPARTITION asia_2011
    VALUES LESS THAN('2012-Jan-01'),
   SUBPARTITION asia_2012
    VALUES LESS THAN('2013-Jan-01')
  ),
 PARTITION americas VALUES('US', 'CANADA')
  (
   SUBPARTITION americas_2011
    VALUES LESS THAN('2012-Jan-01'),
   SUBPARTITION americas_2012
    VALUES LESS THAN('2013-Jan-01')
  )
);
```

The sales table has three partitions (europe, asia, and americas). Each partition has two range-defined subpartitions:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SU
BPARTITIONS;
 partition_name | subpartition_name |  high_value
----------------+-------------------+---------------
 europe         | europe_2011       | '2012-Jan-01'
 europe         | europe_2012       | '2013-Jan-01'
 asia           | asia_2011         | '2012-Jan-01'
 asia           | asia_2012         | '2013-Jan-01'
 americas       | americas_2011     | '2012-Jan-01'
 americas       | americas_2012     | '2013-Jan-01'
(6 rows)
```

The following command adds a subpartition named europe_2013:

```
ALTER TABLE sales MODIFY PARTITION europe
 ADD SUBPARTITION europe_2013
 VALUES LESS THAN('2015-Jan-01');
```

After this command is called, the table contains the europe_2013 subpartition:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SU
BPARTITIONS;
 partition_name | subpartition_name |  high_value
----------------+-------------------+---------------
 europe         | europe_2011       | '2012-Jan-01'
 europe         | europe_2012       | '2013-Jan-01'
 europe         | europe_2013       | '2015-Jan-01'
 asia           | asia_2011         | '2012-Jan-01'
 asia           | asia_2012         | '2013-Jan-01'
 americas       | americas_2011     | '2012-Jan-01'
 americas       | americas_2012     | '2013-Jan-01'
```

```
(7 rows)
```

Note that when you add a new range subpartition, the subpartitioning rules must specify a range that is located after existing subpartitions.

### Example - add a subpartition to a RANGE-LIST partitioned table

The following example adds a LIST subpartition to the range-partitioned sales table. The sales table is created by using the following command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST (country)
  (
    PARTITION first_half_2012 VALUES LESS THAN('01-JUL-2012')
    (
      SUBPARTITION europe VALUES ('ITALY', 'FRANCE'),
      SUBPARTITION americas VALUES ('US', 'CANADA')
    ),

    PARTITION second_half_2012 VALUES LESS THAN('01-JAN-2013')
    (
      SUBPARTITION asia VALUES ('INDIA', 'PAKISTAN')
    )
  );
```

The sales table has two partitions, named first_half_2012 and second_half_2012, respectively. The first_half_2012 partition has two subpartitions named europe and americas, respectively. The second_half_2012 partition has one subpartition named asia.

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SU
BPARTITIONS;
  partition_name  | subpartition_name |     high_value
------------------+-------------------+---------------------
 first_half_2012  | europe            | 'ITALY', 'FRANCE'
 first_half_2012  | americas          | 'US', 'CANADA'
 second_half_2012 | asia              | 'INDIA', 'PAKISTAN'
(3 rows)
```

The following command adds a subpartition named east_asia to the second_half_2012 partition:

```
ALTER TABLE sales MODIFY PARTITION second_half_2012
```

```
  ADD SUBPARTITION east_asia VALUES ('CHINA');
```

After this command is called, the table contains the east_asia subpartition:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SU
BPARTITIONS;
  partition_name  | subpartition_name |     high_value
------------------+-------------------+---------------------
 first_half_2012  | europe            | 'ITALY', 'FRANCE'
 first_half_2012  | americas          | 'US', 'CANADA'
 second_half_2012 | asia              | 'INDIA', 'PAKISTAN'
 second_half_2012 | east_asia         | 'CHINA'
(4 rows)
```

## 15.5.4 ALTER TABLE... SPLIT PARTITION

The ALTER TABLE... SPLIT PARTITION command adds a partition to an existing partitioned table.

### Overview

You can use the ALTER TABLE... SPLIT PARTITION command to divide a partition into two partitions and redistribute the content of the partition. The ALTER TABLE... SPLIT PARTITION command has two forms.

The first form splits a RANGE partition into two partitions:

```
ALTER TABLE table_name SPLIT PARTITION partition_name
  AT (range_part_value)
  INTO
  (
    PARTITION new_part1
      [TABLESPACE tablespace_name],
    PARTITION new_part2
      [TABLESPACE tablespace_name]
  );
```

The second form splits a LIST partition into two partitions:

```
ALTER TABLE table_name SPLIT PARTITION partition_name
  VALUES (value[, value]...)
  INTO
  (
    PARTITION new_part1
      [TABLESPACE tablespace_name],
    PARTITION new_part2
      [TABLESPACE tablespace_name]
  );
```

### Description

The ALTER TABLE... SPLIT PARTITION command adds a partition to an existing partitioned table. The number of partitions in a partitioned table is not limited.

When you run an ALTER TABLE... SPLIT PARTITION command, POLARDB compatible with Oracle creates two new partitions and redistributes the content of the old partition between the new partitions (as constrained by the partitioning rules).

Include the TABLESPACE clause to specify a tablespace in which a new partition will reside. If you do not specify a tablespace, the partition will be created in the default tablespace.

If the table is indexed, the index will be created on the new partition.

To use the ALTER TABLE... SPLIT PARTITION command, you must be the table owner, or have superuser (or administrative) privileges.

**Parameters**

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the partitioned table. |
| partition_name | The name of the partition to be split. |
| new_part1 | The name of the first new partition to be created. Partition names must be unique among all partitions and subpartitions, and must follow the naming conventions for object identifiers.<br>new_part1 will receive the rows that meet the partitioning constraints specified in the ALTER TABLE... SPLIT PARTITION command. |
| new_part2 | The name of the second new partition to be created. Partition names must be unique among all partitions and subpartitions, and must follow the naming conventions for object identifiers.<br>new_part2 will receive the rows that are not directed to new_part1 by the subpartitioning constraints specified in the ALTER TABLE... SPLIT PARTITION command. |
| range_part_value | Use range_part_value to specify the boundary rules by which the new partition is created. Each partitioning rule must contain at least one column of a data type that has two operators (for example, a greater-than or equal to operator, and a less-than operator). Range boundaries are evaluated based on a LESS THAN clause and are non-inclusive. A date boundary of January 1, 2010 only includes the date values that fall on or before December 31, 2009. |

| Parameter | Description |
|---|---|
| (value[, value ]...) | Use value to specify a quoted literal value (or a list of literal values separated by commas) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but the number of values specified in a rule is not limited.<br>For more information about creating a DEFAULT or MAXVALUE partition, see Handle stray values in a LIST or RANGE partitioned table. |
| tablespace_name | The name of the tablespace in which the partition or subpartition resides. |

### Example - split a LIST partition

The following example splits one partition in the list-partitioned sales table into two new partitions, and redistributes the content of the partition between the two new partitions. The sales table is created by using the following statement:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

The table definition creates three partitions (europe, asia, and americas). The following command adds rows to each partition:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

The rows are distributed among the partitions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date        | amount
----------------+---------+---------+----------+--------------------+-------
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
(17 rows)
```

The following command splits the americas partition into two partitions named us and canada:

```
ALTER TABLE sales SPLIT PARTITION americas
  VALUES ('US')
  INTO (PARTITION us, PARTITION canada);
```

A SELECT statement is used to confirm that the rows are distributed among the partitions as expected:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid   | dept_no | part_no | country  |        date        | amount
--------------+---------+---------+----------+--------------------+--------
 sales_europe |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia   |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia   |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia   |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia   |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia   |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
 sales_us     |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_us     |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_us     |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_us     |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_us     |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
 sales_canada |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_canada |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_canada |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
```

```
(17 rows)
```

### Example - split a RANGE partition

The following example splits the q4_2012 partition in the range-partitioned sales table into two partitions, and redistributes the content of the partition. Run the following command to create the sales table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012
    VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012
    VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
);
```

The table definition creates four partitions (q1_2012, q2_2012, q3_2012, and q4_2012 ). The following command adds rows to each partition:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

A SELECT statement is used to confirm that the rows are distributed among the partitions as expected:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | dept_no | part_no | country  |        date        | amount
---------------+---------+---------+----------+--------------------+--------
 sales_q1_2012 |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_q1_2012 |     20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_q1_2012 |     30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_q2_2012 |     40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_q2_2012 |     20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_q2_2012 |     30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_q2_2012 |     40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_q3_2012 |     10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_q3_2012 |     10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_q3_2012 |     10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_q3_2012 |     20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_q3_2012 |     40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_q4_2012 |     40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_q4_2012 |     30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_q4_2012 |     40 | 4788b   | US       | 09-OCT-12 00:00:00 |   15000
 sales_q4_2012 |     20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_q4_2012 |     20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(17 rows)
```

The following command splits the q4_2012 partition into two partitions named q4_2012_p1 and q4_2012_p2:

```
ALTER TABLE sales SPLIT PARTITION q4_2012
  AT ('15-Nov-2012')
  INTO
  (
    PARTITION q4_2012_p1,
    PARTITION q4_2012_p2
  );
```

A SELECT statement is used to confirm that the rows are distributed among the partitions as expected:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
     tableoid     | dept_no | part_no | country  |        date        |amount
------------------+---------+---------+----------+--------------------+------
 sales_q1_2012    |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 | 45000
 sales_q1_2012    |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 | 75000
 sales_q1_2012    |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 | 75000
 sales_q2_2012    |      40 | 9519b   | US       | 12-APR-12 00:00:00 |145000
 sales_q2_2012    |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 | 37500
 sales_q2_2012    |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 |120000
 sales_q2_2012    |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |  4950
 sales_q3_2012    |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 | 15000
 sales_q3_2012    |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 |650000
 sales_q3_2012    |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 |650000
 sales_q3_2012    |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |  5090
 sales_q3_2012    |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |  4950
 sales_q4_2012_p1 |      40 | 4577b   | US       | 11-NOV-12 00:00:00 | 25000
 sales_q4_2012_p1 |      40 | 4788b   | US       | 09-OCT-12 00:00:00 | 15000
 sales_q4_2012_p1 |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 |650000
 sales_q4_2012_p2 |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 | 50000
 sales_q4_2012_p2 |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |  5090
```

```
(17 rows)
```

## 15.5.5 ALTER TABLE... SPLIT SUBPARTITION

The ALTER TABLE... SPLIT SUBPARTITION command adds a subpartition to an existing subpartitioned table.

### Overview

You can use the ALTER TABLE... SPLIT SUBPARTITION command to divide a subpartition into two subpartitions and redistribute the content of the subpartition. The ALTER TABLE... SPLIT SUBPARTITION command has two forms.

The first form splits a range subpartition into two subpartitions:

```
ALTER TABLE table_name SPLIT SUBPARTITION subpartition_name
  AT (range_part_value)
  INTO
  (
    SUBPARTITION new_subpart1
      [TABLESPACE tablespace_name],
    SUBPARTITION new_subpart2
      [TABLESPACE tablespace_name]
  );
```

The second form splits a list subpartition into two subpartitions:

```
ALTER TABLE table_name SPLIT SUBPARTITION subpartition_name
  VALUES (value[, value]...)
  INTO
  (
    SUBPARTITION new_subpart1
      [TABLESPACE tablespace_name],
    SUBPARTITION new_subpart2
      [TABLESPACE tablespace_name]
  );
```

### Description

The ALTER TABLE... SPLIT SUBPARTITION command adds a subpartition to an existing subpartitioned table. The number of defined subpartitions is not limited. When you run an ALTER TABLE... SPLIT SUBPARTITION command, POLARDB compatible with Oracle creates two new subpartitions. It moves rows that contain values that are constrained by the specified subpartition rules into new_subpart1, and the remaining rows into new_subpart2.

The new subpartition rules must reference the column specified in the rules that define the existing subpartitions.

Include the TABLESPACE clause to specify a tablespace in which a new subpartition will reside. If you do not specify a tablespace, the subpartition will be created in the default tablespace.

If the table is indexed, the index will be created on the new subpartition.

To use the ALTER TABLE... SPLIT SUBPARTITION command, you must be the table owner, or have superuser (or administrative) privileges.

**Parameters**

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the partitioned table. |
| subpartition_name | The name of the subpartition to be split. |
| new_subpart1 | The name of the first new subpartition to be created. Subpartition names must be unique among all partitions and subpartitions, and must follow the naming convention s for object identifiers.<br>new_subpart1 will receive the rows that meet the subpartitioning constraints specified in the ALTER TABLE... SPLIT SUBPARTITION command. |
| new_subpart2 | The name of the second new subpartition to be created. Subpartition names must be unique among all partitions and subpartitions, and must follow the naming convention s for object identifiers.<br>new_subpart2 will receive the rows that are not directed to new_subpart1 by the subpartitioning constraints specified in the ALTER TABLE... SPLIT SUBPARTITION command. |

| Parameter | Description |
|---|---|
| (value[, value]...) | Use value to specify a quoted literal value (or a list of literal values separated by commas) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but the number of values specified in a rule is not limited. value may be null, default (if specifying a LIST subpartition), or maxvalue (if specifying a RANGE subpartition).<br>For more information about creating a DEFAULT or MAXVALUE partition, see Handle stray values in a LIST or RANGE partitioned table. |
| tablespace_name | The name of the tablespace in which the partition or subpartition resides. |

**Example - split a LIST subpartition**

The following example splits a list subpartition and redistributes the content of the subpartition between two new subpartitions. The sample table (sales) is created by using the following command:

```
CREATE TABLE sales
(
 dept_no     number,
 part_no     varchar2,
 country     varchar2(20),
 date        date,
 amount      number
)
PARTITION BY RANGE(date)
 SUBPARTITION BY LIST (country)
 (
  PARTITION first_half_2012 VALUES LESS THAN('01-JUL-2012')
  (
   SUBPARTITION p1_europe VALUES ('ITALY', 'FRANCE'),
   SUBPARTITION p1_americas VALUES ('US', 'CANADA')
  ),
  PARTITION second_half_2012 VALUES LESS THAN('01-JAN-2013')
  (
   SUBPARTITION p2_europe VALUES ('ITALY', 'FRANCE'),
   SUBPARTITION p2_americas VALUES ('US', 'CANADA')
  )
 );
```

The sales table has two partitions, named first_half_2012 and second_half_2012. Each partition has two range-defined subpartitions that distribute the content of the partition into subpartitions based on the value of the country column.

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SU
BPARTITIONS;
  partition_name  | subpartition_name |    high_value
------------------+-------------------+-------------------
 second_half_2012 | p2_europe         | 'ITALY', 'FRANCE'
 first_half_2012  | p1_europe         | 'ITALY', 'FRANCE'
 second_half_2012 | p2_americas       | 'US', 'CANADA'
 first_half_2012  | p1_americas       | 'US', 'CANADA'
(4 rows)
```

The following command adds rows to each subpartition:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000');
```

A SELECT statement is used to confirm that rows are distributed among the subpartitions as expected:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
     tableoid     | dept_no | part_no | country|        date        |amount
------------------+---------+---------+--------+--------------------+------
 sales_p1_europe   |      10 | 4519b   | FRANCE | 17-JAN-12 00:00:00 |  45000
 sales_p1_americas |      40 | 9519b   | US     | 12-APR-12 00:00:00 | 145000
 sales_p1_americas |      30 | 9519b   | CANADA | 01-FEB-12 00:00:00 |  75000
 sales_p1_americas |      30 | 4519b   | CANADA | 08-APR-12 00:00:00 | 120000
 sales_p1_americas |      40 | 3788a   | US     | 12-MAY-12 00:00:00 |   4950
 sales_p2_europe   |      10 | 9519b   | ITALY  | 07-JUL-12 00:00:00 |  15000
 sales_p2_europe   |      10 | 9519a   | FRANCE | 18-AUG-12 00:00:00 | 650000
 sales_p2_europe   |      10 | 9519b   | FRANCE | 18-AUG-12 00:00:00 | 650000
 sales_p2_americas |      40 | 4577b   | US     | 11-NOV-12 00:00:00 |  25000
 sales_p2_americas |      30 | 7588b   | CANADA | 14-DEC-12 00:00:00 |  50000
 sales_p2_americas |      40 | 4788a   | US     | 23-SEP-12 00:00:00 |   4950
 sales_p2_americas |      40 | 4788b   | US     | 09-OCT-12 00:00:00 |  15000
(12 rows)
```

The following command splits the p2_americas subpartition into two new subpartitions and redistributes the content:

```
ALTER TABLE sales SPLIT SUBPARTITION p2_americas
  VALUES ('US')
```

```
 INTO
 (
   SUBPARTITION p2_us,
   SUBPARTITION p2_canada
 );
```

After this command is called, the p2_americas subpartition is deleted. In the place of the subpartition, the server creates two new subpartitions (p2_us and p2_canada):

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SU
BPARTITIONS;
  partition_name  | subpartition_name |    high_value
------------------+-------------------+-------------------
 first_half_2012  | p1_europe         | 'ITALY', 'FRANCE'
 first_half_2012  | p1_americas       | 'US', 'CANADA'
 second_half_2012 | p2_europe         | 'ITALY', 'FRANCE'
 second_half_2012 | p2_canada         | 'CANADA'
 second_half_2012 | p2_us             | 'US'
(5 rows)
```

Querying the sales table shows that the content of the p2_americas subpartition has been redistributed:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
     tableoid      | dept_no | part_no | country |        date        |amount
-------------------+---------+---------+---------+--------------------+------
 sales_p1_europe   |      10 | 4519b   | FRANCE  | 17-JAN-12 00:00:00 | 45000
 sales_p1_americas |      40 | 9519b   | US      | 12-APR-12 00:00:00 |145000
 sales_p1_americas |      30 | 9519b   | CANADA  | 01-FEB-12 00:00:00 | 75000
 sales_p1_americas |      30 | 4519b   | CANADA  | 08-APR-12 00:00:00 |120000
 sales_p1_americas |      40 | 3788a   | US      | 12-MAY-12 00:00:00 |  4950
 sales_p2_europe   |      10 | 9519b   | ITALY   | 07-JUL-12 00:00:00 | 15000
 sales_p2_europe   |      10 | 9519a   | FRANCE  | 18-AUG-12 00:00:00 |650000
 sales_p2_europe   |      10 | 9519b   | FRANCE  | 18-AUG-12 00:00:00 |650000
 sales_p2_us       |      40 | 4577b   | US      | 11-NOV-12 00:00:00 | 25000
 sales_p2_us       |      40 | 4788a   | US      | 23-SEP-12 00:00:00 |  4950
 sales_p2_us       |      40 | 4788b   | US      | 09-OCT-12 00:00:00 | 15000
 sales_p2_canada   |      30 | 7588b   | CANADA  | 14-DEC-12 00:00:00 | 50000
(12 rows)
```

### Example - split a RANGE subpartition

The following example splits a range subpartition and redistributes the content of the subpartition between two new subpartitions. The sample table (sales) is created by using the following command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
  SUBPARTITION BY RANGE(date)
(
  PARTITION europe VALUES('FRANCE', 'ITALY')
    (
      SUBPARTITION europe_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION europe_2012
        VALUES LESS THAN('2013-Jan-01')
    ),
  PARTITION asia VALUES('INDIA', 'PAKISTAN')
    (
      SUBPARTITION asia_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION asia_2012
        VALUES LESS THAN('2013-Jan-01')
    ),
  PARTITION americas VALUES('US', 'CANADA')
    (
      SUBPARTITION americas_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION americas_2012
        VALUES LESS THAN('2013-Jan-01')
    )
);
```

The sales table has three partitions (europe, asia, and americas). Each partition has two range-defined subpartitions that distribute the content of the partition into subpartitions based on the value of the date column.

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SU
BPARTITIONS;
 partition_name | subpartition_name |  high_value
----------------+-------------------+---------------
 europe         | europe_2011       | '2012-Jan-01'
 europe         | europe_2012       | '2013-Jan-01'
 asia           | asia_2011         | '2012-Jan-01'
 asia           | asia_2012         | '2013-Jan-01'
 americas       | americas_2011     | '2012-Jan-01'
 americas       | americas_2012     | '2013-Jan-01'
(6 rows)
```

The following command adds rows to each subpartition:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

A SELECT statement is used to confirm that rows are distributed among the subpartitions as expected:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
      tableoid       | dept_no|part_no| country |       date        |amount
---------------------+--------+-------+---------+-------------------+---
 sales_europe_2012   |      10| 4519b | FRANCE  | 17-JAN-12 00:00:00 | 45000
 sales_europe_2012   |      10| 9519b | ITALY   | 07-JUL-12 00:00:00 | 15000
 sales_europe_2012   |      10| 9519a | FRANCE  | 18-AUG-12 00:00:00 | 650000
 sales_europe_2012   |      10| 9519b | FRANCE  | 18-AUG-12 00:00:00 | 650000
 sales_asia_2012     |      20| 3788a | INDIA   | 01-MAR-12 00:00:00 | 75000
 sales_asia_2012     |      20| 3788a | PAKISTAN| 04-JUN-12 00:00:00 | 37500
 sales_asia_2012     |      20| 3788b | INDIA   | 21-SEP-12 00:00:00 | 5090
 sales_asia_2012     |      20| 4519a | INDIA   | 18-OCT-12 00:00:00 | 650000
 sales_asia_2012     |      20| 4519b | INDIA   | 02-DEC-12 00:00:00 | 5090
 sales_americas_2012 |      40| 9519b | US      | 12-APR-12 00:00:00 | 145000
 sales_americas_2012 |      40| 4577b | US      | 11-NOV-12 00:00:00 | 25000
 sales_americas_2012 |      30| 7588b | CANADA  | 14-DEC-12 00:00:00 | 50000
 sales_americas_2012 |      30| 9519b | CANADA  | 01-FEB-12 00:00:00 | 75000
 sales_americas_2012 |      30| 4519b | CANADA  | 08-APR-12 00:00:00 | 120000
 sales_americas_2012 |      40| 3788a | US      | 12-MAY-12 00:00:00 | 4950
 sales_americas_2012 |      40| 4788a | US      | 23-SEP-12 00:00:00 | 4950
 sales_americas_2012 |      40| 4788b | US      | 09-OCT-12 00:00:00 | 15000
(17 rows)
```

The following command splits the americas_2012 subpartition into two new subpartitions and redistributes the content:

```
ALTER TABLE sales
  SPLIT SUBPARTITION americas_2012
  AT('2012-Jun-01')
  INTO
  (
    SUBPARTITION americas_p1_2012,
    SUBPARTITION americas_p2_2012
  );
```

After this command is called, the americas_2012 subpartition is deleted. In the place of the subpartition, the server creates two new subpartitions (americas_p1_2012 and americas_p2_2012):

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SU
BPARTITIONS;
 partition_name | subpartition_name |  high_value
----------------+-------------------+---------------
 europe         | europe_2012       | '2013-Jan-01'
 europe         | europe_2011       | '2012-Jan-01'
 americas       | americas_2011     | '2012-Jan-01'
 americas       | americas_p2_2012  | '2013-Jan-01'
 americas       | americas_p1_2012  | '2012-Jun-01'
 asia           | asia_2012         | '2013-Jan-01'
 asia           | asia_2011         | '2012-Jan-01'
```

```
(7 rows)
```

Querying the sales table shows that the content of the americas_2012 subpartition has been redistributed:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
      tableoid        | dept_no|part_no|country |      date        |amount
----------------------+--------+-------+--------+------------------+------- sales_euro
pe_2012     |      10| 4519b |FRANCE  | 17-JAN-12 00:00:00|  45000
 sales_europe_2012    |      10| 9519b |ITALY   | 07-JUL-12 00:00:00|  15000
 sales_europe_2012    |      10| 9519a |FRANCE  | 18-AUG-12 00:00:00| 650000
 sales_europe_2012    |      10| 9519b |FRANCE  | 18-AUG-12 00:00:00| 650000
 sales_asia_2012      |      20| 3788a |INDIA   | 01-MAR-12 00:00:00|  75000
 sales_asia_2012      |      20| 3788a |PAKISTAN| 04-JUN-12 00:00:00|  37500
 sales_asia_2012      |      20| 3788b |INDIA   | 21-SEP-12 00:00:00|   5090
 sales_asia_2012      |      20| 4519a |INDIA   | 18-OCT-12 00:00:00| 650000
 sales_asia_2012      |      20| 4519b |INDIA   | 02-DEC-12 00:00:00|   5090
 sales_americas_p1_2012|      40| 9519b |US      | 12-APR-12 00:00:00| 145000
 sales_americas_p1_2012|      30| 9519b |CANADA  | 01-FEB-12 00:00:00|  75000
 sales_americas_p1_2012|      30| 4519b |CANADA  | 08-APR-12 00:00:00| 120000
 sales_americas_p1_2012|      40| 3788a |US      | 12-MAY-12 00:00:00|   4950
 sales_americas_p2_2012|      40| 4577b |US      | 11-NOV-12 00:00:00|  25000
 sales_americas_p2_2012|      30| 7588b |CANADA  | 14-DEC-12 00:00:00|  50000
 sales_americas_p2_2012|      40| 4788a |US      | 23-SEP-12 00:00:00|   4950
 sales_americas_p2_2012|      40| 4788b |US      | 09-OCT-12 00:00:00|  15000
(17 rows)
```

## 15.5.6 ALTER TABLE... EXCHANGE PARTITION

The ALTER TABLE... EXCHANGE PARTITION command swaps an existing table with a partition or subpartition.

**Overview**

If you plan to add a large quantity of data to a partitioned table, you can use the ALTER TABLE... EXCHANGE PARTITION command to transfer a bulk load of data. You can also use the ALTER TABLE... EXCHANGE PARTITION command to remove outdated or redundant data from storage.

The ALTER TABLE... EXCHANGE PARTITION command has two forms.

- The first form swaps a table for a partition:

```
ALTER TABLE target_table
  EXCHANGE PARTITION target_partition
  WITH TABLE source_table
  [(WITH | WITHOUT) VALIDATION];
```

- The second form swaps a table for a subpartition:

```
ALTER TABLE target_table
  EXCHANGE SUBPARTITION target_subpartition
  WITH TABLE source_table
```

```
[(WITH | WITHOUT) VALIDATION];
```

The ALTER TABLE... EXCHANGE PARTITION command makes no distinctions between a partition and a subpartition:

- You can exchange a partition by using the EXCHANGE PARTITION or EXCHANGE SUBPARTITION clause.
- You can exchange a subpartition by using EXCHANGE PARTITION or EXCHANGE SUBPARTITION clause.

### Description

When the ALTER TABLE... EXCHANGE PARTITION command completes, the data is swapped . The data that originally resides in the target partition resides in the source table, and the data that originally resides in the source table resides in the target partition.

The structure of the source table must match the structure of the target table (both tables must have matching columns and data types). The data contained within the table must adhere to the partitioning constraints.

POLARDB compatible with Oracle accepts the WITHOUT VALIDATION clause, but ignores it. The new table is always validated.

You must own a table to call ALTER TABLE... EXCHANGE PARTITION or ALTER TABLE... EXCHANGE SUBPARTITION against that table.

### Parameters

| Parameter | Description |
|---|---|
| target_table | The name (optionally schema-qualified) of the table in which the partition resides. |
| target_partition | The name of the partition or subpartition to be replaced. |
| source_table | The name of the table that will replace the target_partition. |

### Example - exchange a table for a partition

The following example demonstrates exchanging a table for a partition (americas) of the sales table. You can run the following command to create the sales table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Run the following command to add sample data to the sales table:

```
INSERT INTO sales VALUES
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the sales table shows that only one row resides in the americas partition:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no| part_no | country |        date         | amount
---------------+--------+---------+---------+--------------------+-----------
 sales_europe  |      10| 4519b   | FRANCE  | 17-JAN-12 00:00:00|      45000
 sales_europe  |      10| 9519b   | ITALY   | 07-JUL-12 00:00:00|      15000
 sales_europe  |      10| 9519a   | FRANCE  | 18-AUG-12 00:00:00|     650000
 sales_europe  |      10| 9519b   | FRANCE  | 18-AUG-12 00:00:00|     650000
 sales_asia    |      20| 3788a   | INDIA   | 01-MAR-12 00:00:00|      75000
 sales_asia    |      20| 3788a   | PAKISTAN| 04-JUN-12 00:00:00|      37500
 sales_asia    |      20| 3788b   | INDIA   | 21-SEP-12 00:00:00|       5090
 sales_asia    |      20| 4519a   | INDIA   | 18-OCT-12 00:00:00|     650000
 sales_asia    |      20| 4519b   | INDIA   | 02-DEC-12 00:00:00|       5090
 sales_americas|      40| 9519b   | US      | 12-APR-12 00:00:00|     145000
(10 rows)
```

The following command creates a table (n_america) that matches the definition of the sales table:

```
CREATE TABLE n_america
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
);
```

The following command adds data to the n_america table. The data conforms to the partitioning rules of the americas partition:

```
INSERT INTO n_america VALUES
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000');
```

The following command swaps the table into the partitioned table:

```
ALTER TABLE sales
  EXCHANGE PARTITION americas
  WITH TABLE n_america;
```

Querying the sales table shows that the content of the n_america table has been exchanged for the content of the americas partition:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no| part_no | country |        date        | amount
---------------+--------+--------+----------+--------------------+-----------
 sales_europe  |      10| 4519b   | FRANCE  | 17-JAN-12 00:00:00 |      45000
 sales_europe  |      10| 9519b   | ITALY   | 07-JUL-12 00:00:00 |      15000
 sales_europe  |      10| 9519a   | FRANCE  | 18-AUG-12 00:00:00 |     650000
 sales_europe  |      10| 9519b   | FRANCE  | 18-AUG-12 00:00:00 |     650000
 sales_asia    |      20| 3788a   | INDIA   | 01-MAR-12 00:00:00 |      75000
 sales_asia    |      20| 3788a   | PAKISTAN| 04-JUN-12 00:00:00 |      37500
 sales_asia    |      20| 3788b   | INDIA   | 21-SEP-12 00:00:00 |       5090
 sales_asia    |      20| 4519a   | INDIA   | 18-OCT-12 00:00:00 |     650000
 sales_asia    |      20| 4519b   | INDIA   | 02-DEC-12 00:00:00 |       5090
 sales_americas|      40| 9519b   | US      | 12-APR-12 00:00:00 |     145000
 sales_americas|      40| 4577b   | US      | 11-NOV-12 00:00:00 |      25000
 sales_americas|      30| 7588b   | CANADA  | 14-DEC-12 00:00:00 |      50000
 sales_americas|      30| 9519b   | CANADA  | 01-FEB-12 00:00:00 |      75000
 sales_americas|      30| 4519b   | CANADA  | 08-APR-12 00:00:00 |     120000
 sales_americas|      40| 3788a   | US      | 12-MAY-12 00:00:00 |       4950
 sales_americas|      40| 4788a   | US      | 23-SEP-12 00:00:00 |       4950
 sales_americas|      40| 4788b   | US      | 09-OCT-12 00:00:00 |      15000
(17 rows)
```

Querying the n_america table shows that the row that was previously stored in the americas partition has been moved to the n_america table:

```
acctg=# SELECT tableoid::regclass, * FROM n_america;
 tableoid  | dept_no | part_no | country |        date        | amount
-----------+---------+---------+---------+--------------------+------------
 n_america |      40 | 9519b   | US      | 12-APR-12 00:00:00 |     145000
(1 row)
```

## 15.5.7 ALTER TABLE... MOVE PARTITION

### Overview

You can use the ALTER TABLE... MOVE PARTITION command to move a partition or subpartition to a different tablespace. The ALTER TABLE... MOVE PARTITION command has two forms.

- The first form is to move a partition to a new tablespace:

```
ALTER TABLE table_name
  MOVE PARTITION partition_name
   TABLESPACE tablespace_name;
```

- The second form is to move a subpartition to a new tablespace:

```
ALTER TABLE table_name
  MOVE SUBPARTITION subpartition_name
   TABLESPACE tablespace_name;
```

The syntax of the ALTER TABLE... MOVE PARTITION command makes no distinctions between a partition and a subpartition:

- You can move a partition by using the MOVE PARTITION or MOVE SUBPARTITION clause.
- You can move a subpartition by using the MOVE PARTITION or MOVE SUBPARTITION clause.

**Description**

The ALTER TABLE... MOVE PARTITION command moves a partition or subpartition from its current tablespace to a different tablespace. You must own a table to call ALTER TABLE... MOVE PARTITION or ALTER TABLE... MOVE SUBPARTITION.

**Parameters**

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the table in which the partition resides. |
| partition_name | The name of the partition or subpartition to be moved. |
| tablespace_name | The name of the tablespace to which the partition or subpartition will be moved. |

**Example - move a partition to a different tablespace**

The following example moves a partition of the sales table from one tablespace to another. First, run the following command to create the sales table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012 VALUES LESS THAN ('2012-Apr-01'),
  PARTITION q2_2012 VALUES LESS THAN ('2012-Jul-01'),
  PARTITION q3_2012 VALUES LESS THAN ('2012-Oct-01'),
  PARTITION q4_2012 VALUES LESS THAN ('2013-Jan-01') TABLESPACE ts_1,
  PARTITION q1_2013 VALUES LESS THAN ('2013-Mar-01') TABLESPACE ts_2
);
```

Querying the ALL_TAB_PARTITIONS view confirms that the partitions reside on the expected servers and tablespaces:

```
acctg=# SELECT partition_name, tablespace_name FROM ALL_TAB_PARTITIONS;
 partition_name | tablespace_name
----------------+-----------------+-----------------
 q1_2013        | ts_2
 q4_2012        | ts_1
 q3_2012        |
 q2_2012        |
 q1_2012        |
(5 rows)
```

After preparing the target tablespace, call the ALTER TABLE... MOVE PARTITION command to move the q1_2013 partition from a tablespace named ts_2 to a tablespace named ts_3:

```
ALTER TABLE sales MOVE PARTITION q1_2013 TABLESPACE ts_3;
```

Querying the ALL_TAB_PARTITIONS view shows that the move was successful:

```
acctg=# SELECT partition_name, tablespace_name FROM ALL_TAB_PARTITIONS;
 partition_name | tablespace_name
----------------+-----------------
 q1_2013        | ts_3
 q4_2012        | ts_1
 q3_2012        |
 q2_2012        |
 q1_2012        |
(5 rows)
```

## 15.5.8 ALTER TABLE... RENAME PARTITION

### Overview

You can use the ALTER TABLE... RENAME PARTITION command to rename a table partition. The command has two forms.

-
  ```
  ALTER TABLE table_name
  RENAME PARTITION partition_name
  TO new_name;
  ```

-
  ```
  ALTER TABLE table_name
  RENAME SUBPARTITION subpartition_name
   TO new_name;
  ```

The ALTER TABLE... RENAME PARTITION command makes no distinctions between a partition and a subpartition:

- You can rename a partition by using the RENAME PARTITION or RENAME SUBPARTITION clause.
- You can rename a subpartition by using the RENAME PARTITION or RENAME SUBPARTITI ON clause.

### Description

The ALTER TABLE... RENAME PARTITION and ALTER TABLE... RENAME SUBPARTITION commands renames a partition or subpartition. You must own the specified table to run ALTER TABLE... RENAME PARTITION or ALTER TABLE... RENAME SUBPARTITION.

### Parameters

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the table in which the partition resides. |
| partition_name | The name of the partition or subpartition to be renamed. |
| new_name | The new name of the partition or subpartition. |

### Example - rename a partition

The following command creates a list-partitioned table named sales:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Querying the ALL_TAB_PARTITIONS view displays the partition names:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 europe         | 'FRANCE', 'ITALY'
 asia           | 'INDIA', 'PAKISTAN'
 americas       | 'US', 'CANADA'
```

```
(3 rows)
```

The following command renames the americas partition to n_america:

```
ALTER TABLE sales
RENAME PARTITION americas TO n_america;
```

Querying the ALL_TAB_PARTITIONS view will show that the partition is renamed:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 europe         | 'FRANCE', 'ITALY'
 asia           | 'INDIA', 'PAKISTAN'
 n_america      | 'US', 'CANADA'
(3 rows)
```

## 15.5.9 DROP TABLE

### Overview

You can use the PostgreSQL DROP TABLE command to delete a partitioned table definition, the partitions and subpartitions of that table, and the table content. Syntax:

```
DROP TABLE table_name
```

### Description

The DROP TABLE command deletes an entire table and the data stored in the table. When you delete a table, all partitions and subpartitions of the table are also deleted.

To use the DROP TABLE command, you must be the owner of the partitioning root, a member of a group that owns the table, the schema owner, or a database superuser.

### Parameters

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the partitioned table. |

### Example

To delete a table, connect to the controller node (the host of the partitioning root), and run the DROP TABLE command. For example, to delete the sales table, run the following command:

```
DROP TABLE sales;
```

The server will confirm that the table has been dropped:

```
acctg=# drop table sales;
DROP TABLE
acctg=#
```

For more information about the DROP TABLE command, see the PostgreSQL core documentation.

## 15.5.10 ALTER TABLE... DROP PARTITION

### Overview

You can use the ALTER TABLE... DROP PARTITION command to delete a partition definition and the data stored in that partition. Syntax:

```
ALTER TABLE table_name DROP PARTITION partition_name;
```

### Description

The ALTER TABLE... DROP PARTITION command deletes a partition and the data stored in the partition. When you delete a partition, all subpartitions of the partition are also deleted.

To use the DROP PARTITION clause, you must be the owner of the partitioning root, a member of a group that owns the table, or have superuser or administrative privileges.

### Parameters

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the partitioned table. |
| partition_name | The name of the partition to be deleted. |

### Example - delete a partition

The following example deletes a partition of the sales table. Run the following command to create the sales table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Querying the ALL_TAB_PARTITIONS view displays the partition names:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 europe         | 'FRANCE', 'ITALY'
 asia           | 'INDIA', 'PAKISTAN'
 americas       | 'US', 'CANADA'
(3 rows)
```

To delete the americas partition from the sales table, invoke the following command:

```
ALTER TABLE sales DROP PARTITION americas;
```

Querying the ALL_TAB_PARTITIONS view shows that the partition has been successfully deleted:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 asia           | 'INDIA', 'PAKISTAN'
 europe         | 'FRANCE', 'ITALY'
(2 rows)
```

## 15.5.11 ALTER TABLE... DROP SUBPARTITION

### Overview

You can use the ALTER TABLE... DROP SUBPARTITION command to delete a subpartition definition and the data stored in that subpartition. Syntax:

```
ALTER TABLE table_name DROP SUBPARTITION subpartition_name;
```

### Description

The ALTER TABLE... DROP SUBPARTITION command deletes a subpartition and the data stored in the subpartition. To use the DROP SUBPARTITION clause, you must be the owner of the partitioning root, a member of a group that owns the table, or have superuser or administrative privileges.

### Parameters

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the partitioned table. |
| subpartition_name | The name of the subpartition to be deleted. |

### Example - delete a subpartition

The following example deletes a subpartition of the sales table. Run the following command to create the sales table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST (country)
  (
    PARTITION first_half_2012 VALUES LESS THAN('01-JUL-2012')
    (
      SUBPARTITION europe VALUES ('ITALY', 'FRANCE'),
      SUBPARTITION americas VALUES ('CANADA', 'US'),
      SUBPARTITION asia VALUES ('PAKISTAN', 'INDIA')
    ),
    PARTITION second_half_2012 VALUES LESS THAN('01-JAN-2013')
  );
```

Querying the ALL_TAB_SUBPARTITIONS view displays the subpartition names:

```
acctg=# SELECT subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name |     high_value
-------------------+---------------------
 europe            | 'ITALY', 'FRANCE'
 americas          | 'CANADA', 'US'
 asia              | 'PAKISTAN', 'INDIA'
(3 rows)
```

To delete the americas subpartition from the sales table, run the following command:

```
ALTER TABLE sales DROP SUBPARTITION americas;
```

Querying the ALL_TAB_SUBPARTITIONS view shows that the subpartition has been successfully deleted:

```
acctg=# SELECT subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name |     high_value
-------------------+---------------------
 europe            | 'ITALY', 'FRANCE'
 asia              | 'PAKISTAN', 'INDIA'
```

```
(2 rows)
```

## 15.5.12 TRUNCATE TABLE

### Overview

You can use the TRUNCATE TABLE command to remove the content of a table, while preserving the table definition. When you truncate a table, all partitions and subpartitions of the table are also truncated. Syntax:

```
TRUNCATE TABLE table_name;
```

### Description

The TRUNCATE TABLE command removes an entire table and the data stored in the table. When you truncate a table, all partitions and subpartitions of the table are also truncated.

To use the TRUNCATE TABLE command, you must be the owner of the partitioning root, a member of a group that owns the table, the schema owner, or a database superuser.

### Parameters

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the partitioned table. |

### Example - empty a table

The following example removes data from the sales table. Run the following command to create the sales table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Run the following command to add values to the sales table:

```
INSERT INTO sales VALUES
(10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
(20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
(40, '9519b', 'US', '12-Apr-2012', '145000'),
(20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
(40, '4577b', 'US', '11-Nov-2012', '25000'),
(30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
(30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
(30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
(40, '3788a', 'US', '12-May-2012', '4950'),
(10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
(10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
(10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
(20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
(40, '4788a', 'US', '23-Sept-2012', '4950'),
(40, '4788b', 'US', '09-Oct-2012', '15000'),
(20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
(20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the sales table shows that the partitions are populated with data:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid   |dept_no | part_no | country |        date        | amount
--------------+--------+---------+----------+--------------------+-----------
sales_europe |     10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |    45000
sales_europe |     10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |    15000
sales_europe |     10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 |   650000
 sales_europe |     10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 |   650000
sales_asia    |     20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |    75000
sales_asia    |     20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |    37500
sales_asia    |     20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |     5090
sales_asia    |     20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 |   650000
sales_asia    |     20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |     5090
sales_americas|     40 | 9519b   | US       | 12-APR-12 00:00:00 |   145000
sales_americas|     40 | 4577b   | US       | 11-NOV-12 00:00:00 |    25000
sales_americas|     30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |    50000
sales_americas|     30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |    75000
sales_americas|     30 | 4519b   | CANADA   | 08-APR-12 00:00:00 |   120000
sales_americas|     40 | 3788a   | US       | 12-MAY-12 00:00:00 |     4950
sales_americas|     40 | 4788a   | US       | 23-SEP-12 00:00:00 |     4950
sales_americas|     40 | 4788b   | US       | 09-OCT-12 00:00:00 |    15000
(17 rows)
```

To delete the content of the sales table, run the following command:

```
TRUNCATE TABLE sales;
```

Querying the sales table will show that the data is removed, but the structure is intact:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
 tableoid | dept_no | part_no | country |  date   | amount
----------+---------+---------+---------+----------+------------
(0 rows)
```

For more information about the TRUNCATE TABLE command, see TRUNCATE.

## 15.5.13 ALTER TABLE... TRUNCATE PARTITION

### Overview

You can use the ALTER TABLE... TRUNCATE PARTITION command to remove all data from a specified partition, leaving the partition structure intact. Syntax:

```
ALTER TABLE table_name TRUNCATE PARTITION partition_name
  [{DROP|REUSE} STORAGE]
```

### Description

You can use the ALTER TABLE... TRUNCATE PARTITION command to remove all data from a specified partition, leaving the partition structure intact. When you truncate a partition, all subpartitions of the partition are also truncated.

The ALTER TABLE... TRUNCATE PARTITION command will not fire any ON DELETE triggers that may exist for the table. However, the command will fire ON TRUNCATE triggers. If an ON TRUNCATE trigger is defined for the partition, all BEFORE TRUNCATE triggers are fired before any truncation occurs, and all AFTER TRUNCATE triggers are fired after the last truncation is performed.

You must have the TRUNCATE permission on a table to invoke ALTER TABLE... TRUNCATE PARTITION.

### Parameters

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the partitioned table. |
| partition_name | The name of the partition to be removed. |

**Note:**

DROP STORAGE and REUSE STORAGE are only included for compatibility. These clauses are parsed and ignored.

### Example - empty a partition

The following example removes the data from a partition of the sales table. Run the following command to create the sales table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date       date,
  amount     number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Run the following command to add values to the sales table:

```
INSERT INTO sales VALUES
(10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
(20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
(40, '9519b', 'US', '12-Apr-2012', '145000'),
(20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
(40, '4577b', 'US', '11-Nov-2012', '25000'),
(30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
(30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
(30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
(40, '3788a', 'US', '12-May-2012', '4950'),
(10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
(10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
(10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
(20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
(40, '4788a', 'US', '23-Sept-2012', '4950'),
(40, '4788b', 'US', '09-Oct-2012', '15000'),
(20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
(20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the sales table shows that the partitions are populated with data:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date        | amount
----------------+---------+---------+----------+--------------------+--------
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
```

```
(17 rows)
```

To delete the content of the americas partition, run the following command:

```
ALTER TABLE sales TRUNCATE PARTITION americas;
```

Querying the sales table will show that the content of the americas partition is removed:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid   | dept_no | part_no | country  |        date        | amount
--------------+---------+---------+----------+--------------------+--------
 sales_europe |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia   |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia   |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia   |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia   |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia   |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(9 rows)
```

Although the rows have been removed, the structure of the americas partition is still intact:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 europe         | 'FRANCE', 'ITALY'
 asia           | 'INDIA', 'PAKISTAN'
 americas       | 'US', 'CANADA'
(3 rows)
```

## 15.5.14 ALTER TABLE... TRUNCATE SUBPARTITION

### Overview

You can use the ALTER TABLE... TRUNCATE SUBPARTITION command to remove all data from a specified subpartition, leaving the subpartition structure intact. Syntax:

```
ALTER TABLE table_name
  TRUNCATE SUBPARTITION subpartition_name
  [{DROP|REUSE} STORAGE]
```

### Description

The ALTER TABLE... TRUNCATE SUBPARTITION command removes all data from a specified subpartition, leaving the subpartition structure intact.

The ALTER TABLE... TRUNCATE SUBPARTITION command will not fire any ON DELETE triggers that may exist for the table. However, the command will fire ON TRUNCATE triggers. If an ON TRUNCATE trigger is defined for the subpartition, all BEFORE TRUNCATE triggers are fired before any truncation occurs, and all AFTER TRUNCATE triggers are fired after the last truncation is performed.

You must have the TRUNCATE permission on a table to run ALTER TABLE... TRUNCATE SUBPARTITION.

### Parameters

| Parameter | Description |
|---|---|
| table_name | The name (optionally schema-qualified) of the partitioned table. |
| subpartition_name | The name of the subpartition to be truncated. |

**Note:**

DROP STORAGE and REUSE STORAGE are only included for compatibility. These clauses are parsed and ignored.

### Example - empty a subpartition

The following example removes the data from a subpartition of the sales table. Run the following command to create the sales table:

```
CREATE TABLE sales
(
 dept_no     number,
 part_no     varchar2,
 country     varchar2(20),
 date        date,
 amount      number
)
PARTITION BY RANGE(date) SUBPARTITION BY LIST (country)
(
 PARTITION "2011" VALUES LESS THAN('01-JAN-2012')
 (
  SUBPARTITION europe_2011 VALUES ('ITALY', 'FRANCE'),
  SUBPARTITION asia_2011 VALUES ('PAKISTAN', 'INDIA'),
  SUBPARTITION americas_2011 VALUES ('US', 'CANADA')
 ),
 PARTITION "2012" VALUES LESS THAN('01-JAN-2013')
 (
  SUBPARTITION europe_2012 VALUES ('ITALY', 'FRANCE'),
  SUBPARTITION asia_2012 VALUES ('PAKISTAN', 'INDIA'),
  SUBPARTITION americas_2012 VALUES ('US', 'CANADA')
 ),
 PARTITION "2013" VALUES LESS THAN('01-JAN-2015')
 (
  SUBPARTITION europe_2013 VALUES ('ITALY', 'FRANCE'),
  SUBPARTITION asia_2013 VALUES ('PAKISTAN', 'INDIA'),
  SUBPARTITION americas_2013 VALUES ('US', 'CANADA')
 )
);
```

Run the following command to add values to the sales table:

```
INSERT INTO sales VALUES
(10, '4519b', 'FRANCE', '17-Jan-2011', '45000'),
(20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
(40, '9519b', 'US', '12-Apr-2012', '145000'),
(20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
(40, '4577b', 'US', '11-Nov-2012', '25000'),
(30, '7588b', 'CANADA', '14-Dec-2011', '50000'),
(30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
(40, '3788a', 'US', '12-May-2011', '4950'),
(20, '3788a', 'US', '04-Apr-2012', '37500'),
(40, '4577b', 'INDIA', '11-Jun-2011', '25000'),
(10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
(20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the sales table shows that the rows have been distributed among the subpartitions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
     tableoid      | dept_no| part_no| country  |       date        |amount
--------------------+--------+--------+----------+-------------------+-------
 sales_2011_europe  |      10| 4519b  | FRANCE   | 17-JAN-11 00:00:00|  45000
 sales_2011_asia    |      40| 4577b  | INDIA    | 11-JUN-11 00:00:00|  25000
 sales_2011_americas|      30| 7588b  | CANADA   | 14-DEC-11 00:00:00|  50000
 sales_2011_americas|      40| 3788a  | US       | 12-MAY-11 00:00:00|   4950
 sales_2012_europe  |      10| 9519b  | ITALY    | 07-JUL-12 00:00:00|  15000
 sales_2012_asia    |      20| 3788a  | INDIA    | 01-MAR-12 00:00:00|  75000
 sales_2012_asia    |      20| 3788a  | PAKISTAN | 04-JUN-12 00:00:00|  37500
 sales_2012_asia    |      20| 4519b  | INDIA    | 02-DEC-12 00:00:00|   5090
 sales_2012_americas|      40| 9519b  | US       | 12-APR-12 00:00:00| 145000
 sales_2012_americas|      40| 4577b  | US       | 11-NOV-12 00:00:00| 25000
 sales_2012_americas|      30| 4519b  | CANADA   | 08-APR-12 00:00:00| 120000
 sales_2012_americas|      20| 3788a  | US       | 04-APR-12 00:00:00|  37500
(12 rows)
```

To delete the content of the 2012_americas partition, run the following command:

```
ALTER TABLE sales TRUNCATE SUBPARTITION "americas_2012";
```

Querying the sales table shows that the content of the americas_2012 partition has been removed:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
     tableoid      | dept_no|part_no| country  |       date         | amount
--------------------+--------+-------+----------+--------------------+-------
 sales_2011_europe  |      10| 4519b | FRANCE   | 17-JAN-11 00:00:00 |  45000
 sales_2011_asia    |      40| 4577b | INDIA    | 11-JUN-11 00:00:00 |  25000
 sales_2011_americas|      30| 7588b | CANADA   | 14-DEC-11 00:00:00 |  50000
 sales_2011_americas|      40| 3788a | US       | 12-MAY-11 00:00:00 |   4950
 sales_2012_europe  |      10| 9519b | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_2012_asia    |      20| 3788a | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_2012_asia    |      20| 3788a | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_2012_asia    |      20| 4519b | INDIA    | 02-DEC-12 00:00:00 |   5090
```

```
(8 rows)
```

Although the rows have been removed, the structure of the 2012_americas partition is still intact:

```
acctg=# SELECT subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name |     high_value
-------------------+---------------------
 2013_europe       | 'ITALY', 'FRANCE'
 2012_europe       | 'ITALY', 'FRANCE'
 2011_europe       | 'ITALY', 'FRANCE'
 2013_asia         | 'PAKISTAN', 'INDIA'
 2012_asia         | 'PAKISTAN', 'INDIA'
 2011_asia         | 'PAKISTAN', 'INDIA'
 2013_americas     | 'US', 'CANADA'
 2012_americas     | 'US', 'CANADA'
 2011_americas     | 'US', 'CANADA'
(9
rows)
```

## 15.6 Handle stray values in a LIST or RANGE partitioned table

A DEFAULT or MAXVALUE partition or subpartition captures any rows that do not meet the other partitioning rules defined for a table.

### Define a DEFAULT partition

A DEFAULT partition captures any rows that do not fit into any other partition in a LIST partitioned (or subpartitioned) table. If you do not include a DEFAULT rule, any row that does not match one of the values in the partitioning constraints will cause an error. Each LIST partition or subpartition may have its own DEFAULT rule.

The syntax of a DEFAULT rule is as follows:

```
PARTITION partition_name VALUES (DEFAULT)
```

Where partition_name specifies the name of the partition or subpartition used to store any rows that do not match the partitioning rules specified for other partitions.

In the last example, a list-partitioned table is created. The server determines which partition in this partitioned table to store the data based on the value of the country column. If you attempt to add a row in which the value of the country column is not listed in the partitioning rules, POLARDB compatible with Oracle reports an error:

```
acctg=# INSERT INTO sales VALUES
acctg-#   (40, '3000x', 'IRELAND', '01-Mar-2012', '45000');
```

```
ERROR:  inserted partition key does not map to any partition
```

The following example creates the same table, but adds a DEFAULT partition. The server will store any rows that do not match a value specified in the partitioning rules for the europe, asia, or americas partition in the others partition.

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA'),
  PARTITION others VALUES (DEFAULT)
);
```

To test the DEFAULT partition, add a row with a value in the country column that does not match any country specified in the partitioning constraints:

```
INSERT INTO sales VALUES
  (40, '3000x', 'IRELAND', '01-Mar-2012', '45000');
```

Querying the sales table confirms that the previously rejected row is now stored in the sales_others partition:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date        | amount
----------------+---------+---------+----------+--------------------+--------
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
 sales_others   |      40 | 3000x   | IRELAND  | 01-MAR-12 00:00:00 |  45000
(18 rows)
```

Note that POLARDB compatible with Oracle does not include a method to reassign the content of a DEFAULT partition or subpartition.

- You cannot use the ALTER TABLE... ADD PARTITION command to add a partition to a table with a DEFAULT rule. However, you can use the ALTER TABLE... SPLIT PARTITION command to split an existing partition.
- You cannot use the ALTER TABLE... ADD SUBPARTITION command to add a subpartition to a table with a DEFAULT rule. However, you can use the ALTER TABLE... SPLIT SUBPARTITION command to split an existing subpartition.

**Define a MAXVALUE partition**

A MAXVALUE partition or subpartition captures any rows that do not fit into any other partition in a range-partitioned or subpartitioned table. If you do not include a MAXVALUE rule, any row that exceeds the maximum limit specified by the partitioning rules will cause an error. Each partition or subpartition may have its own MAXVALUE partition.

Note that POLARDB compatible with Oracle does not include a method to reassign the content of a MAXVALUE partition or subpartition:

- You cannot use the ALTER TABLE... ADD PARTITION command to add a partition to a table with a MAXVALUE rule. However, you can use the ALTER TABLE... SPLIT PARTITION command to split an existing partition.
- You cannot use the ALTER TABLE... ADD SUBPARTITION command to add a subpartition to a table with a MAXVALUE rule. However, you can use the ALTER TABLE... SPLIT SUBPARTITION command to split an existing subpartition.

The syntax of a MAXVALUE rule is as follows:

```
PARTITION partition_name VALUES LESS THAN (MAXVALUE)
```

Where partition_name specifies the name of the partition used to store any rows that do not match the partitioning rules specified for other partitions.

In the last example, a range-partitioned table is created. The data in this table is partitioned based on the value of date column. If you attempt to add a row in which the value of the date column exceeds a date listed in the partitioning constraints, POLARDB compatible with Oracle reports an error:

```
acctg=# INSERT INTO sales VALUES
acctg-#   (40, '3000x', 'IRELAND', '01-Mar-2013', '45000');
```

```
ERROR:  inserted partition key does not map to any partition
```

The following CREATE TABLE command creates the same table, but this table has a MAXVALUE partition. Instead of reporting an error, the server will store any rows that do not match the previous partitioning constraints in the others partition:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012 VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012 VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012 VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012 VALUES LESS THAN('2013-Jan-01'),
  PARTITION others VALUES LESS THAN (MAXVALUE)
);
```

To test the MAXVALUE partition, add a row with a value in the date column that exceeds the last date value listed in each partitioning rule. The server will store this row in the others partition:

```
INSERT INTO sales VALUES
                (40, '3000x', 'IRELAND', '2015-Oct-01', '45000');
```

Querying the sales table confirms that the previously rejected row is now stored in the sales_others partition:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid   | dept_no | part_no | country  |        date        | amount
--------------+---------+---------+----------+--------------------+---------
 sales_q1_2012 |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |   45000
 sales_q1_2012 |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |   75000
 sales_q1_2012 |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |   75000
 sales_q2_2012 |      40 | 9519b   | US       | 12-APR-12 00:00:00 |  145000
 sales_q2_2012 |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |   37500
 sales_q2_2012 |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 |  120000
 sales_q2_2012 |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |    4950
 sales_q3_2012 |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |   15000
 sales_q3_2012 |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 |  650000
 sales_q3_2012 |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 |  650000
 sales_q3_2012 |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |    5090
 sales_q3_2012 |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |    4950
 sales_q4_2012 |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |   25000
 sales_q4_2012 |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |   50000
 sales_q4_2012 |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |   15000
 sales_q4_2012 |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 |  650000
 sales_q4_2012 |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |    5090
 sales_others  |      40 | 3000x   | IRELAND  | 01-MAR-13 00:00:00 |   45000
```

```
(18 rows)
```

## 15.7 Specify multiple partition key columns in a RANGE partitioned table

You can improve performance by specifying multiple key columns for a RANGE partitioned table. If you often select rows by using comparison operators (based on a greater-than or less-than value) on a small set of columns, consider using these columns in RANGE partitioning rules.

### Specify multiple key columns in a range-partitioned table

A range-partitioned table definition may include multiple columns in the partition key. To specify multiple partition key columns for a range-partitioned table, you must include the column names in a comma-separated list after the PARTITION BY RANGE clause:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  sale_year    number,
  sale_month   number,
  sale_day     number,
  amount      number
)
PARTITION BY RANGE(sale_year, sale_month)
(
  PARTITION q1_2012
    VALUES LESS THAN(2012, 4),
  PARTITION q2_2012
    VALUES LESS THAN(2012, 7),
  PARTITION q3_2012
    VALUES LESS THAN(2012, 10),
  PARTITION q4_2012
    VALUES LESS THAN(2013, 1)
);
```

If a table has multiple partition key columns, you must specify multiple key values when querying the table to take full advantage of partition pruning.

```
acctg=# EXPLAIN SELECT * FROM sales WHERE sale_year = 2012 AND sale_month = 8;
                              QUERY PLAN
----------------------------------------------------------------------------- Result  (cost=0.
00..14.35 rows=2 width=250)
   ->  Append  (cost=0.00..14.35 rows=2 width=250)
         ->  Seq Scan on sales  (cost=0.00..0.00 rows=1 width=250)
               Filter: ((sale_year = 2012::numeric) AND (sale_month = 8::numeric))
         ->  Seq Scan on sales_q3_2012 sales  (cost=0.00..14.35 rows=1 width=250)
               Filter: ((sale_year = 2012::numeric) AND (sale_month = 8::numeric))
```

```
(6 rows)
```

All rows with a value of 8 in the sale_month column and a value of 2012 in the sale_year column will be stored in the q3_2012 partition. POLARDB compatible with Oracle will only search this partition.

# 15.8 Retrieve information about a partitioned table

## 15.8.1 Overview

POLARDB compatible with Oracle provides five system catalog views. You can use these catalog views to view information about the structure of partitioned tables.

**Query the partitioning views**

You can query the following views to retrieve information about partitioned and subpartitioned tables.

- ALL_PART_TABLES
- ALL_TAB_PARTITIONS
- ALL_TAB_SUBPARTITIONS
- ALL_PART_KEY_COLUMNS
- ALL_SUBPART_KEY_COLUMNS

In the Table partitioning views - reference topic, the structure of each view is explained. If you are using the EDB-PSQL client, you can also discover the structure of a view by entering the following:

```
\d view name
```

The view_name specifies the name of the table partitioning view.

Querying a view can provide information about the structure of a partitioned or subpartitioned table. For example, the following code snippet displays the system-assigned names of a subpartitioned table:

```
acctg=# SELECT subpartition_name, partition_name FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name | partition_name
-------------------+----------------
 SYS_SUBP107       | americas
 SYS_SUBP104       | asia
 SYS_SUBP101       | europe
 SYS_SUBP108       | americas
 SYS_SUBP105       | asia
 SYS_SUBP102     | europe
 SYS_SUBP109     | americas
 SYS_SUBP106     | asia
 SYS_SUBP103     | europe
(9 rows)
```

## 15.8.2 Table partitioning views - reference

Query the following catalog views (compatible with Oracle databases) to review detailed information about your partitioned tables.

### ALL PART TABLES

The following table lists the available information in the ALL_PART_TABLES view:

| Column | Type | Description |
|---|---|---|
| owner | name | The owner of the table. |
| table_name | name | The name of the table. |
| schema_name | name | The schema in which the table resides. |
| partitioning_type | text | RANGE or LIST |
| subpartitioning_type | text | RANGE, LIST, or NONE |
| partition_count | bigint | The number of partitions. |
| def_subpartition_count | integer | The default subpartition count. This column is set to 0. |
| partitioning_key_count | integer | The number of columns listed in the partition by clause. |
| subpartitioning_key_count | integer | The number of columns in the subpartition by clause. |
| status | character varying(8) | This column is set to VALID and provided for Oracle compatibility. |
| def_tablespace_name | character varying(30) | This column is set to NULL and provided for Oracle compatibility. |
| def_pct_free | numeric | This column is set to NULL and provided for Oracle compatibility. |
| def_pct_used | numeric | This column is set to NULL and provided for Oracle compatibility. |
| def_ini_trans | numeric | This column is set to NULL and provided for Oracle compatibility. |
| def_max_trans | numeric | This column is set to NULL and provided for Oracle compatibility. |

| Column | Type | Description |
|---|---|---|
| def_initial_extent | character varying(40) | This column is set to NULL and provided for Oracle compatibility. |
| def_next_extent | character varying(40) | This column is set to NULL and provided for Oracle compatibility. |
| def_min_extents | character varying(40) | This column is set to NULL and provided for Oracle compatibility. |
| def_max_extents | character varying(40) | This column is set to NULL and provided for Oracle compatibility. |
| def_pct_increase | character varying(40) | This column is set to NULL and provided for Oracle compatibility. |
| def_freelists | numeric | This column is set to NULL and provided for Oracle compatibility. |
| def_freelist_groups | numeric | This column is set to NULL and provided for Oracle compatibility. |
| def_logging | character varying(7) | This column is set to YES and provided for Oracle compatibility. |
| def_compression | character varying(8) | This column is set to NONE and provided for Oracle compatibility. |
| def_buffer_pool | character varying(7) | This column is set to DEFAULT and provided for Oracle compatibility. |
| ref_ptn_constraint_name | character varying(30) | This column is set to NULL and provided for Oracle compatibility. |
| interval | character varying(1000) | This column is set to NULL and provided for Oracle compatibility. |

**ALL_TAB_PARTITIONS**

The following table lists the available information in the ALL_TAB_PARTITIONS view:

| Column | Type | Description |
|---|---|---|
| table_owner | name | The owner of the table. |
| table_name | name | The name of the table. |
| schema_name | name | The schema in which the table resides. |
| composite | text | This column is set to YES if the table is subpartitioned and set to NO if the table is not subpartitioned. |

| Column | Type | Description |
|---|---|---|
| partition_name | name | The name of the partition. |
| subpartition_count | bigint | The number of subpartitions for this partition. |
| high_value | text | The partition limit for RANGE partitions, or the partition value for LIST partitions. |
| high_value_length | integer | The length of high_value. |
| partition_position | integer | The ordinal position of this partition. |
| tablespace_name | name | The table space in which this partition resides. |
| pct_free | numeric | This column is set to 0 and provided for Oracle compatibility. |
| pct_used | numeric | This column is set to 0 and provided for Oracle compatibility. |
| ini_trans | numeric | This column is set to 0 and provided for Oracle compatibility. |
| max_trans | numeric | This column is set to 0 and provided for Oracle compatibility. |
| initial_extent | numeric | This column is set to NULL and provided for Oracle compatibility. |
| next_extent | numeric | This column is set to NULL and provided for Oracle compatibility. |
| min_extent | numeric | This column is set to 0 and provided for Oracle compatibility. |
| max_extent | numeric | This column is set to 0 and provided for Oracle compatibility. |
| pct_increase | numeric | This column is set to 0 and provided for Oracle compatibility. |
| freelists | numeric | This column is set to NULL and provided for Oracle compatibility. |
| freelist_groups | numeric | This column is set to NULL and provided for Oracle compatibility. |
| logging | character varying(7) | This column is set to YES and provided for Oracle compatibility. |

| Column | Type | Description |
|---|---|---|
| compression | character varying(8) | This column is set to NONE and provided for Oracle compatibility. |
| num_rows | numeric | The approximate number of rows in this partition. |
| blocks | integer | The approximate number of blocks in this partition. |
| empty_blocks | numeric | This column is set to NULL and provided for Oracle compatibility. |
| avg_space | numeric | This column is set to NULL and provided for Oracle compatibility. |
| chain_cnt | numeric | This column is set to NULL and provided for Oracle compatibility. |
| avg_row_len | numeric | This column is set to NULL and provided for Oracle compatibility. |
| sample_size | numeric | This column is set to NULL and provided for Oracle compatibility. |
| last_analyzed | timestamp without time zone | This column is set to NULL and provided for Oracle compatibility. |
| buffer_pool | character varying(7) | This column is set to NULL and provided for Oracle compatibility. |
| global_stats | character varying(3) | This column is set to YES and provided for Oracle compatibility. |
| user_stats | character varying(3) | This column is set to NO and provided for Oracle compatibility. |
| backing_table | regclass | The OID of the backing table for this partition. |
| server_name | name | The name of the server on which the partition resides. |

#### ALL TAB SUBPARTITIONS

The following table lists the available information in the ALL_TAB_SUBPARTITIONS view:

| Column | Type | Description |
|---|---|---|
| table_owner | name | The owner of the table. |
| table_name | name | The name of the table. |
| schema_name | name | The schema in which the table resides. |

| Column | Type | Description |
|---|---|---|
| partition_name | name | The name of the partition. |
| high_value | text | The subpartition limit for RANGE subpartitions, or the subpartition value for LIST subpartitions. |
| high_value_length | integer | The length of high value. |
| subpartition_name | name | The name of the subpartition. |
| subpartition_position | integer | The ordinal position of this subpartition. |
| tablespace_name | name | The tablespace in which this partition resides. |
| pct_free | numeric | This column is set to 0 and provided for Oracle compatibility. |
| pct_used | numeric | This column is set to 0 and provided for Oracle compatibility. |
| ini_trans | numeric | This column is set to 0 and provided for Oracle compatibility. |
| max_trans | numeric | This column is set to 0 and provided for Oracle compatibility. |
| initial_extent | numeric | This column is set to NULL and provided for Oracle compatibility. |
| next_extent | numeric | This column is set to NULL and provided for Oracle compatibility. |
| min_extent | numeric | This column is set to 0 and provided for Oracle compatibility. |
| max_extent | numeric | This column is set to 0 and provided for Oracle compatibility. |
| pct_increase | numeric | This column is set to 0 and provided for Oracle compatibility. |
| freelists | numeric | This column is set to NULL and provided for Oracle compatibility. |
| freelist_groups | numeric | This column is set to NULL and provided for Oracle compatibility. |
| logging | character varying(7) | This column is set to YES and provided for Oracle compatibility. |

| Column | Type | Description |
|---|---|---|
| compression | character varying(8) | This column is set to NONE and provided for Oracle compatibility. |
| num_rows | numeric | The approximate number of rows in this subpartition. |
| blocks | integer | The approximate number of blocks in this subpartition. |
| empty_blocks | numeric | This column is set to NULL and provided for Oracle compatibility. |
| avg_space | numeric | This column is set to NULL and provided for Oracle compatibility. |
| chain_cnt | numeric | This column is set to NULL and provided for Oracle compatibility. |
| avg_row_len | numeric | This column is set to NULL and provided for Oracle compatibility. |
| sample_size | numeric | This column is set to NULL and provided for Oracle compatibility. |
| last_analyzed | timestamp without time zone | This column is set to NULL and provided for Oracle compatibility. |
| buffer_pool | character varying(7) | This column is set to NULL and provided for Oracle compatibility. |
| global_stats | character varying(3) | This column is set to YES and provided for Oracle compatibility. |
| user_stats | character varying(3) | This column is set to NO and provided for Oracle compatibility. |
| backing_table | regclass | The OID of the backing table for this subpartition . |
| server_name | name | The name of the server on which the subpartition resides. |

### ALL PART KEY COLUMNS

The following table lists the available information in the ALL_PART_KEY_COLUMNS view:

| Column | Type | Description |
|---|---|---|
| owner | name | The owner of the table. |
| name | name | The name of the table. |

| Column | Type | Description |
|---|---|---|
| schema | name | The schema in which the table resides. |
| object_type | character(5) | This column is set to TABLE and provided for Oracle compatibility. |
| column_name | name | The name of the partition key column. |
| column_position | integer | The position of this column within the partition key. Each column has a corresponding column position (for example, the first column has a column position of 1, the second column has a column position of 2). |

**ALL SUBPART KEY COLUMNS**

The following table lists the available information in the ALL_SUBPART_KEY_COLUMNS view.

| Column | Type | Description |
|---|---|---|
| owner | name | The owner of the table. |
| name | name | The name of the table. |
| schema | name | The schema in which the table resides. |
| object_type | character(5) | This column is set to TABLE and provided for Oracle compatibility. |
| column_name | name | The name of the partition key column. |
| column_position | integer | The position of this column within the subpartition key. Each column has a corresponding column position (for example, the first column has a column position of 1, the second column has a column position of 2). |

# 16 Packages

## 16.1 Overview

A package is a named collection of functions, procedures, variables, cursors, user-defined record types, and records that are referenced with the package identifier.

The following section shows the characteristics of packages:

- Packages provide a convenient method to organize the functions and procedures to achieve a certain purpose. The permission to use the package functions and procedures is based on one privilege granted to the entire package. All of the package programs must be referenced with a common name.
- Certain functions, procedures, variables, and types in the package can be declared as public. Public entities are visible and can be referenced by other programs that are granted the EXECUTE permission on the package. For public functions and procedures, only the signatures such as program names, parameters, and return types are visible. The Structured Process Language (SPL) code of these functions and procedures cannot be accessed by others. Therefore, applications that utilize a package depend on only the information available in the signatures instead of the procedural logic.
- Other functions, procedures, variables, types in the package can be declared as private . Private entities can be referenced and used by functions and procedures within the package, but cannot be referenced and used by external applications. Private entities are used only by programs within the package.
- Function names and procedure names can be reloaded within a package. One or more functions and procedures can be defined with the same name but different signatures . This allows you to create multiple programs with the same name. These programs run the same jobs based on different types of input.

## 16.2 Package components

## 16.2.1 Package specification syntax

The package specification defines the user interface for a package (the API). The specification lists the functions, procedures, types, exceptions, and cursors that are visible to a user of the package.

The following syntax is used to define an interface for a package:

```
CREATE [ OR REPLACE ] PACKAGE package_name
  [ authorization_clause ]
  { IS | AS }
  [ declaration; ] ...
 [ procedure_or_function_declaration; ] ...
 [ package_name ] ;
```

Where

```
 authorization_clause :=
{ AUTHID DEFINER } | { AUTHID CURRENT_USER }
```

Where

```
procedure_or_function_declaration :=
procedure_declaration | function_declaration
```

Where

```
procedure_declaration :=
PROCEDURE proc_name[ argument_list ] [restriction_pragma];
```

Where

```
function_declaration :=
FUNCTION func_name [ argument_list ]
  RETURN rettype [ restriction_pragma ];
```

Where

```
argument_list :=
( argument_declaration [, ...] )
```

Where

```
argument_declaration :=
argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
```

Where

```
restriction_pragma :=
PRAGMA RESTRICT_REFERENCES(name, restrictions)
```

Where

```
restrictions :=
restriction [, ... ]
```

**Parameters**

| Parameter | Description |
|---|---|
| package_name | The identifier assigned to the package. Each package must have a unique name within the schema. |
| AUTHID DEFINER | If you omit the AUTHID clause or specify the AUTHID DEFINER parameter, permissions of the package owner are used to determine permissions of accessing database objects. |
| AUTHID CURRENT_USER | If you specify the AUTHID CURRENT_USER parameter, permissions of the current user who runs a program in the package are used to determine access permissions. |
| declaration | The identifier of a public variable. A public variable can be accessed from outside the package by using the package_name.variable syntax. There can be zero, one, or more public variables. You must define public variables before you declare procedures or functions.<br>Valid values:<br>• Variable declaration<br>• Record declaration<br>• Collection declaration<br>• REF CURSOR and CURSOR variable declaration<br>• TYPE definitions for records, collections, and REF CURSORs<br>• Exception<br>• Object variable declaration |
| proc_name | The name of a public procedure. |

| Parameter | Description |
|---|---|
| argname | The name of an argument. The argument is referenced by this name within a function or procedure body. |
| IN \| IN OUT \| OUT | The argument mode. IN: The argument is only used for input. This is the default value. IN OUT: The argument is used to receive a value and return a value. OUT: The argument is only used for output. |
| argtype | The data types of an argument. An argument type can be a basic data type, a copy of the type of an existing column that uses %TYPE, or a user-defined type such as a nested table or an object type. The basic data type cannot be specified a length. For example, you must specify VARCHAR2 instead of VARCHAR2(10) as the data type.<br>You can write tablename.columnname%TYPE to reference the type of a column. This enables a procedure be independent of changes of the definition of a table sometimes. |
| DEFAULT value | If the input argument is not provided when you call the procedure, the DEFAULT clause provides a default value for the input argument. You cannot specify DEFAULT for arguments that are in the IN OUT or OUT modes. |
| func_name | The name of a public function. |
| rettype | The data type returned. |

| Parameter | Description |
|---|---|
| DETERMINISTIC | DETERMINISTIC a synonym for IMMUTABLE. A DETERMINISTIC function cannot be used to modify the database and always returns the same result when you input the same argument values. This function is not used to query database and does not use the information that is not in the argument list . If you include this clause, any call of the function with all-constant arguments can be replaced with the function value. |
| restriction | The following keywords are supported for compatibility and can be ignored.<br>• RNDS<br>• RNPS<br>• TRUST<br>• WNDS<br>• WNPS |

## 16.2.2 Package body syntax

The implementation details of the package are in the package body. The package body may contain objects that are not visible to the package user.

The following syntax is used to define a package body.

```
CREATE [ OR REPLACE ] PACKAGE BODY package_name
  { IS | AS }
  [ private_declaration; ] ...
  [ procedure_or_function_definition ] ...
  [ package_initializer ]
 [ package_name ] ;
```

Where

```
procedure_or_function_definition :=
procedure_definition | function_definition
```

Where

```
procedure_definition :=
PROCEDURE proc_name[ argument_list ]
  [ options_list ]
  { IS | AS }
 procedure_body
 END [ proc_name ] ;
```

Where

```
procedure_body :=
[ declaration; ] [, ...]
BEGIN
 statement; [...]
[ EXCEPTION
   { WHEN exception [OR exception] [...]] THEN statement; }
   [...]
]
```

Where

```
function_definition :=
FUNCTION func_name [ argument_list ]
  RETURN rettype [DETERMINISTIC]
  [ options_list ]
  { IS | AS }
 function_body
 END [ func_name ] ;
```

Where

```
function_body :=
[ declaration; ] [, ...]
BEGIN
 statement; [...]
[ EXCEPTION
  { WHEN exception [ OR exception ] [...] THEN statement; }
  [...]
]
```

Where

```
argument_list :=
( argument_declaration [, ...] )
```

Where

```
argument_declaration :=
argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
```

Where

```
options_list :=
option [ ... ]
```

Where

```
option :=
STRICT
LEAKPROOF
COST execution_cost
ROWS result_rows
```

```
SET config_param { TO value | = value | FROM CURRENT }
```

Where

```
package_initializer :=
BEGIN
 statement; [...]
END;
```

**Parameters**

| Parameter | Description |
|---|---|
| package_name | The name of the package to which this package body belongs. You must have defined the package specification with this name. |
| private_declaration | The identifier of a private variable that can be accessed by any procedure or function within the package. There can be zero, one, or more private variables. Valid values:<br>• Variable declaration<br>• Record declaration<br>• Collection declaration<br>• REF CURSOR and CURSOR variable declaration<br>• TYPE definitions for records, collections, and REF CURSORs<br>• Exception<br>• Object variable declaration |
| proc_name | The name of the procedure to be created. |
| PRAGMA AUTONOMOUS_TRANSACTION | The command that sets the function as an autonomous transaction. |
| declaration | A variable, type, REF CURSOR, or subprogram declaration. If subprogram declarations are included, they must be declared after all other variable, type, and REF CURSOR declarations. |
| statement | A Structured Process Language (SPL) program statement. The DECLARE - BEGIN - END block is considered as a part of the SPL statement. Therefore, the function body may contain nested blocks. |

| Parameter | Description |
|---|---|
| exception | The exception condition name. For example : NO_DATA_FOUND, OTHERS. |
| func_name | The name of the function to be created. |
| rettype | The data type returned, which may be any of the types listed by argtype. As for argtype , a length cannot be specified for rettype. |
| DETERMINISTIC | Specifies that the function always returns the same result when you input the same argument values. You cannot use the DETERMINISTIC function to modify the database.<br>**Note:**<br>• The DETERMINISTIC keyword is equivalent to the IMMUTABLE option in PostgreSQL.<br>• If you specify DETERMINISTIC for a public function in the package body, you must also specify DETERMINISTIC for the function declaration in the package specification. For private functions, there is no function declaration in the package specification. |
| PRAGMA AUTONOMOUS_TRANSACTION | The command that sets the function as an autonomous transaction. |
| declaration | A variable, type, REF CURSOR, or subprogram declaration. If subprogram declarations are included, they must be declared after all other variable, type, and REF CURSOR declarations. |
| argname | The name of a formal argument. The argument is referenced by this name within a procedure body. |
| IN \| IN OUT \| OUT | The argument mode. IN: The argument is only used for input. This is the default value. IN OUT: The argument is used to receive a value and return a value. OUT: The argument is only used for output. |

| Parameter | Description |
|---|---|
| argtype | The data types of an argument. An argument type can be a basic data type, a copy of the type of an existing column that uses %TYPE, or a user-defined type such as a nested table or an object type. The basic data type cannot be specified a length. For example, you must specify VARCHAR2 instead of VARCHAR2(10) as the data type.<br>You can write tablename.columnname% TYPE to reference the type of a column. This enables a procedure be independen t of changes of the definition of a table sometimes. |
| DEFAULT value | If the input argument is not provided when you call the procedure, the DEFAULT clause provides a default value for the input argument. You cannot specify DEFAULT for arguments that are in the IN OUT or OUT modes.<br>**Note:**<br>The following options are not compatible with Oracle databases. They are only extensions to Oracle package syntax provided by ApsaraDB PolarDB. |
| STRICT | The STRICT keyword specifies that the function is not executed if you call the function with a NULL argument. The function returns NULL instead. |
| LEAKPROOF | The LEAKPROOF keyword specifies that except for the return value, the function will not reveal any information about arguments. |

| Parameter | Description |
|---|---|
| PARALLEL { UNSAFE \| RESTRICTED \| SAFE } | The PARALLEL clause allows you to use parallel sequential scans in the parallel mode. Different from a serial sequential scan, a parallel sequential scan uses multiple workers to scan a relation in parallel during a query. Valid values:<br>• UNSAFE: The procedure or function cannot be executed in parallel mode. In this case, a serial execution plan is implemented. This is the default value if you omit the PARALLEL clause.<br>• RESTRICTED: The procedure or function can be executed in the parallel mode , but the execution is restricted to the parallel group leader. If the qualification for any particular relation has and parallel restrictions, that relation cannot be chosen for parallelism.<br>• SAFE: The procedure or function can be executed in the parallel mode without any restriction. |
| execution_cost | This parameter specifies estimated execution cost for the function. The value must be a positive number. Unit: cpu_operator_cost. If the function returns a set, this is the collection of execution costs for per returned row. The default value is 0.0025. |
| result_rows | The estimated number of rows that the query planner expects the function to return . The default value is 1000. |
| SET | The SET clause helps you set a parameter value for the duration of the function.<br>• config_param: Specifies the parameter name.<br>• value: Specifies the value of the parameter.<br>• FROM CURRENT: Guarantees that the parameter value is restored when the function ends. |

| Parameter | Description |
|---|---|
| package_initializer | The statements in the package_initializer are executed once for each of your session when the package is first referenced<br>**Note:**<br>The STRICT, LEAKPROOF, PARALLEL, COST, ROWS and SET keywords can provide extended functionality for ApsaraDB PolarDB but are not supported by Oracle. |

## 16.3 Create a package

### 16.3.1 Create a package specification

The package specification contains the definitions of all the elements in the package that can be referenced from outside of the package. These definitions are called the public elements of the package, and they act as the package interface. The following example shows how to create package specification.

```
--
-- The [ackage specification for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE emp_admin
IS

  FUNCTION get_dept_name (
    p_deptno     NUMBER DEFAULT 10
  )
  RETURN VARCHAR2;
  FUNCTION update_emp_sal (
    p_empno      NUMBER,
    p_raise      NUMBER
  )
  RETURN NUMBER;
  PROCEDURE hire_emp (
    p_empno      NUMBER,
    p_ename      VARCHAR2,
    p_job        VARCHAR2,
    p_sal        NUMBER,
    p_hiredate   DATE DEFAULT sysdate,
    p_comm       NUMBER DEFAULT 0,
    p_mgr        NUMBER,
    p_deptno     NUMBER DEFAULT 10
  );
  PROCEDURE fire_emp (
    p_empno      NUMBER
  );
```

```
END emp_admin;
```

This example shows how to create the emp_admin package specification. This package specification consists of two functions and two stored procedures. You can also add the OR REPLACE clause to the CREATE PACKAGE statement for convenience.

## 16.3.2 Create a package body

The package body contains the actual implementation behind the package specification. For the preceding emp_admin package specification, you must create a package body that implements the specification. The body contains the implementation of the functions and stored procedures in the specification.

```
--
--  The package body for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
  --
  --  The function that queries the 'dept' table based on the department.
  --  number and returns the corresponding department name.
  --
  FUNCTION get_dept_name (
    p_deptno       IN NUMBER DEFAULT 10
  )
  RETURN VARCHAR2
  IS
    v_dname        VARCHAR2(14);
  BEGIN
    SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
    RETURN v_dname;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Invalid department number ' ‖ p_deptno);
      RETURN '';
  END;
  --
  --  Function that updates an employee's salary based on the
  --  employee number and salary increment/decrement passed
  --  as IN parameters.  Upon successful completion the function
  --  returns the new updated salary.
  --
  FUNCTION update_emp_sal (
    p_empno        IN NUMBER,
    p_raise        IN NUMBER
  )
  RETURN NUMBER
  IS
    v_sal          NUMBER := 0;
  BEGIN
    SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
    v_sal := v_sal + p_raise;
    UPDATE emp SET sal = v_sal WHERE empno = p_empno;
    RETURN v_sal;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Employee ' ‖ p_empno ‖ ' not found');
      RETURN -1;
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
      DBMS_OUTPUT.PUT_LINE(SQLERRM);
      DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
      DBMS_OUTPUT.PUT_LINE(SQLCODE);
      RETURN -1;
  END;
  --
  -- Procedure that inserts a new employee record into the 'emp' table.
  --
  PROCEDURE hire_emp (
    p_empno         NUMBER,
    p_ename         VARCHAR2,
    p_job           VARCHAR2,
    p_sal           NUMBER,
    p_hiredate      DATE    DEFAULT sysdate,
    p_comm          NUMBER  DEFAULT 0,
    p_mgr           NUMBER,
    p_deptno        NUMBER  DEFAULT 10
  )
  AS
  BEGIN
    INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
      VALUES(p_empno, p_ename, p_job, p_sal,
           p_hiredate, p_comm, p_mgr, p_deptno);
  END;
  --
  -- Procedure that deletes an employee record from the 'emp' table based
  -- on the employee number.
  --
  PROCEDURE fire_emp (
    p_empno         NUMBER
  )
  AS
  BEGIN
    DELETE FROM emp WHERE empno = p_empno;
  END;
END;
```

## 16.4 Reference a package

To reference the types, items and subprograms that are declared within a package specification, you must use the dot notation. For example:

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
```

You can execute the following SQL statement to invoke a function from the emp_admin package specification.

```
SELECT emp_admin.get_dept_name(10) FROM DUAL;
```

This statement invokes the get_dept_name function that is declared within the package emp_admin, and passes the department number as an argument to the function. The function returns the name of the department. The returned value is ACCOUNTING, which corresponds to department number 10.

## 16.5 Use packages with user-defined types

The following example incorporates various user-defined types that are discussed in previous chapters within the context of a package.

The emp_rpt package specification shows the declaration for a record type emprec_typ, a weakly-typed REF CURSOR emp_refcur, as publicly accessible along with two functions and two procedures. The open_emp_by_dept function returns EMP_REFCUR of the REF CURSOR type. Both fetch_emp and close_refcur procedures declare a weakly-typed REF CURSOR as a formal parameter.

```
CREATE OR REPLACE PACKAGE emp_rpt
IS
    TYPE emprec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR(10)
    );
    TYPE emp_refcur IS REF CURSOR;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2;
    FUNCTION open_emp_by_dept (
        p_deptno    IN emp.deptno%TYPE
    ) RETURN EMP_REFCUR;
    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    );
    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    );
END emp_rpt;
```

The package body shows the declaration of several private variables, such as the dept_cur static cursor, the depttab_typ table type, the t_dept table variable, the t_dept_max integer variable, and the r_emp record variable.

```
CREATE OR REPLACE PACKAGE BODY emp_rpt
IS
    CURSOR dept_cur IS SELECT * FROM dept;
    TYPE depttab_typ IS TABLE of dept%ROWTYPE
        INDEX BY BINARY_INTEGER;
    t_dept          DEPTTAB_TYP;
    t_dept_max      INTEGER := 1;
    r_emp           EMPREC_TYP;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2
    IS
    BEGIN
        FOR i IN 1..t_dept_max LOOP
            IF p_deptno = t_dept(i).deptno THEN
                RETURN t_dept(i).dname;
            END IF;
        END LOOP;
        RETURN 'Unknown';
    END;

    FUNCTION open_emp_by_dept(
        p_deptno    IN emp.deptno%TYPE
    ) RETURN EMP_REFCUR
    IS
        emp_by_dept EMP_REFCUR;
    BEGIN
        OPEN emp_by_dept FOR SELECT empno, ename FROM emp
            WHERE deptno = p_deptno;
        RETURN emp_by_dept;
    END;

    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
        DBMS_OUTPUT.PUT_LINE('-----    -------');
        LOOP
            FETCH p_refcur INTO r_emp;
            EXIT WHEN p_refcur%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(r_emp.empno ‖ '     ' ‖ r_emp.ename);
        END LOOP;
    END;

    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        CLOSE p_refcur;
    END;
BEGIN
    OPEN dept_cur;
    LOOP
        FETCH dept_cur INTO t_dept(t_dept_max);
        EXIT WHEN dept_cur%NOTFOUND;
        t_dept_max := t_dept_max + 1;
    END LOOP;
    CLOSE dept_cur;
    t_dept_max := t_dept_max - 1;
END emp_rpt;
```

This package contains an initialization section that loads the private table variable t_dept and uses the private static cursor dept_cur. The t_dept variable is used as the table from which you query the department name in the get_dept_name function.

The open_emp_by_dept function returns a REF CURSOR variable for a result set of employee numbers and names for a specified department argument. This REF CURSOR variable can be passed to the fetch_emp procedure to retrieve and list the individual rows of the result set. The close_refcur procedure can be used to close the REF CURSOR variable associated with this result set.

The following anonymous block is used to run package functions and procedures. In the declaration of the anonymous block, the public REF CURSOR type EMP_REFCUR of the package is used to record the declaration of the v_emp_cur cursor variable. The v_emp_cur cursor variable contains the pointer to the result set that is passed between the package function and procedures.

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    emp_rpt.fetch_emp(v_emp_cur);
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    emp_rpt.close_refcur(v_emp_cur);
END;
```

The following example shows the result of this anonymous block.

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

The following anonymous block illustrates another way of returning the same result. Instead of using the fetch_emp and close_refcur package procedures, the logic of these programs is coded directly into the anonymous block. In the declaration of this anonymous block, add the r_emp record variable. The variable is declared by using the EMPREC_TYP public record type of the package.

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
    r_emp           emp_rpt.EMPREC_TYP;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH v_emp_cur INTO r_emp;
        EXIT WHEN v_emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' ||
            r_emp.ename);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    CLOSE v_emp_cur;
END;
```

The following example shows the result of this anonymous block.

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

## 16.6 Drop a package

The following statement shows the syntax to drop the entire package or only drop the body of the package.

```
DROP PACKAGE [ BODY ] package_name;
```

If you omit the BODY keyword, both the package specification and the package body are dropped. In this case, the entire package is dropped. If you specify the BODY keyword, only the package body is dropped. The package specification remains intact. The package_name parameter specifies the identifier of the package to be dropped.

You can use the following statement to drop only the package body of emp_admin.

```
DROP PACKAGE BODY emp_admin;
```

You can use the following statement to drop the entire emp_admin package.

```
DROP PACKAGE emp_admin;
```

# 17 Built-in packages

## 17.1 Overview

This chapter describes the built-in packages that are provided with POLARDB compatible with Oracle. For certain packages, non-superusers must be explicitly granted the EXECUTE privilege on the package before using any of the functions or stored procedures in the package. For most of the built-in packages, the EXECUTE privilege is granted to PUBLIC by default. For more information about using the GRANT command to provide access to a package, see the GRANT command.

All built-in packages are owned by the special sys user. You must specify this user when granting or revoking privileges on built-in packages.

```
GRANT EXECUTE ON PACKAGE SYS.UTL_FILE TO john;
```

## 17.2 DBMS_ALERT

The DBMS_ALERT package provides the capability to register for, send, and receive alerts.

**Table 17-1: DBMS_ALERT functions and stored procedures**

| Function/stored procedure | Return type | Description |
|---|---|---|
| REGISTER(name) | N/A | Registers to be able to receive alerts named, name. |
| REMOVE(name) | N/A | Removes registration for the alert named, name. |
| REMOVEALL | N/A | Removes registration for all alerts. |
| SIGNAL(name, message) | N/A | Signals the alert named, name, with message. |
| WAITANY(name OUT, message OUT, status OUT, timeout) | N/A | Waits for any registered alert to occur. |
| WAITONE(name, message OUT, status OUT, timeout) | N/A | Waits for the specified alert, name, to occur. |

The DBMS_ALERT package in POLARDB compatible with Oracle is partially implemented when compared to Oracle's version. POLARDB compatible with Oracle only supports the functions and stored procedures that are listed in the preceding table.

POLARDB compatible with Oracle allows a maximum of 500 concurrent alerts. You can use the dbms_alert.max_alerts GUC variable (located in the postgresql.conf file) to specify the maximum number of concurrent alerts allowed on a system.

To set a value for the dbms_alert.max_alerts variable, open the postgresql.conf file ( default location: /opt/PostgresPlus/9.3AS/data) with your choice of editor. Then edit the dbms_alert.max_alerts parameter, as shown in the following example:

```
dbms_alert.max_alerts = alert_count
```

**Note:**

alert_count specifies the maximum number of concurrent alerts. The default value of dbms_alert.max_alerts is 100. To disable this feature, set dbms_alert.max_alerts to 0.

For the dbms_alert.max_alerts GUC variable to function as expected, the custom_variable_classes parameter must contain dbms_alerts:

```
custom_variable_classes = 'dbms_alert, ...'
```

After editing the postgresql.conf file parameters, you must restart the server for the changes to take effect.

## REGISTER

The REGISTER stored procedure enables the current session to be notified of the specified alert.

### Syntax

```
REGISTER(name VARCHAR2)
```

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the alert to be registered. |

### Examples

The following anonymous block registers for an alert named alert_test, and then waits for the signal.

```
DECLARE
    v_name          VARCHAR2(30) := 'alert_test';
    v_msg           VARCHAR2(80);
    v_status        INTEGER;
    v_timeout       NUMBER(3) := 120;
BEGIN
    DBMS_ALERT.REGISTER(v_name);
    DBMS_OUTPUT.PUT_LINE('Registered for alert ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Waiting for signal...') ;
    DBMS_ALERT.WAITONE(v_name,v_msg,v_status,v_timeout);
    DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
    DBMS_ALERT.REMOVE(v_name);
END;

Registered for alert alert_test
Waiting for signal...
```

### REMOVE

The REMOVE stored procedure unregisters the session for the named alert.

**Syntax**

```
REMOVE(name VARCHAR2)
```

**Parameters**

| Parameter | Description |
|---|---|
| name | The name of the alert to be unregistered. |

### REMOVEALL

The REMOVEALL stored procedure unregisters the session for all alerts.

**Syntax**

```
REMOVEALL
```

### SIGNAL

The SIGNAL stored procedure signals the occurrence of the named alert.

**Syntax**

```
SIGNAL(name VARCHAR2, message VARCHAR2)
```

**Parameters**

| Parameter | Description |
|---|---|
| name | The name of the alert. |
| message | The information to pass with this alert. |

### Examples

The following anonymous block signals an alert for alert_test.

```
DECLARE
    v_name   VARCHAR2(30) := 'alert_test';
BEGIN
    DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END;

Issued alert for alert_test
```

## WAITANY

The WAITANY stored procedure waits for any of the registered alerts to occur.

### Syntax

```
WAITANY(name OUT VARCHAR2, message OUT VARCHAR2, status OUT INTEGER, timeout
NUMBER)
```

### Parameters

| Parameter | Description |
|---|---|
| name | The variable that receives the name of the alert. |
| message | The variable that receives the message sent by the SIGNAL stored procedure. |
| status | The status code returned by the operation . Valid values: 0 and 1. 0 indicates that an alert occurred. 1 indicates that a timeout occurred. |
| timeout | The time to wait for an alert. Unit: second. |

### Examples

The following anonymous block uses the WAITANY stored procedure to receive an alert named alert_test or any_alert:

```
DECLARE
    v_name          VARCHAR2(30);
    v_msg          VARCHAR2(80);
    v_status        INTEGER;
    v_timeout       NUMBER(3) := 120;
BEGIN
    DBMS_ALERT.REGISTER('alert_test');
    DBMS_ALERT.REGISTER('any_alert');
    DBMS_OUTPUT.PUT_LINE('Registered for alert alert_test and any_alert');
    DBMS_OUTPUT.PUT_LINE('Waiting for signal...') ;
    DBMS_ALERT.WAITANY(v_name,v_msg,v_status,v_timeout);
    DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
    DBMS_ALERT.REMOVEALL;
END;

Registered for alert alert_test and any_alert
Waiting for signal...
```

The following anonymous block issues a signal for any_alert:

```
DECLARE
    v_name   VARCHAR2(30) := 'any_alert';
BEGIN
    DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END;

Issued alert for any_alert
```

The following output shows that control returns to the first anonymous block and the remaining code is executed:

```
Registered for alert alert_test and any_alert
Waiting for signal...
Alert name   : any_alert
Alert msg    : This is the message from any_alert
Alert status : 0
Alert timeout: 120 seconds
```

## WAITONE

The WAITONE stored procedure waits for the specified registered alert to occur.

### Syntax

```
WAITONE(name VARCHAR2, message OUT VARCHAR2, status OUT INTEGER, timeout
NUMBER)
```

### Parameters

| Parameter | Description |
|---|---|
| name | The name of the alert. |
| message | The variable that receives the message sent by the SIGNAL stored procedure. |

| Parameter | Description |
|---|---|
| status | The status code returned by the operation . Valid values: 0 and 1. 0 indicates that an alert occurred. 1 indicates that a timeout occurred. |
| timeout | The time to wait for an alert. Unit: second. |

**Examples**

The following anonymous block is similar to the one used in the WAITANY example except that the WAITONE stored procedure is used to receive the alert named alert_test.

```
DECLARE
    v_name          VARCHAR2(30) := 'alert_test';
    v_msg           VARCHAR2(80);
    v_status        INTEGER;
    v_timeout       NUMBER(3) := 120;
BEGIN
    DBMS_ALERT.REGISTER(v_name);
    DBMS_OUTPUT.PUT_LINE('Registered for alert ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Waiting for signal...') ;
    DBMS_ALERT.WAITONE(v_name,v_msg,v_status,v_timeout);
    DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
    DBMS_ALERT.REMOVE(v_name);
END;

Registered for alert alert_test
Waiting for signal...
```

The following anonymous block issues a signal for alert_test:

```
DECLARE
    v_name   VARCHAR2(30) := 'alert_test';
BEGIN
    DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END;

Issued alert for alert_test
```

The following output shows that the first session is alerted, control returns to the first anonymous block, and the remaining code is executed:

```
Registered for alert alert_test
Waiting for signal...
Alert name   : alert_test
Alert msg    : This is the message from alert_test
Alert status : 0
Alert timeout: 120 seconds
```

### Comprehensive example

The following example uses two triggers to send alerts when the dept table or the emp table is changed. An anonymous block listens for these alerts and displays messages when an alert is received.

The triggers on the dept and emp tables are defined as follows:

```
CREATE OR REPLACE TRIGGER dept_alert_trig
    AFTER INSERT OR UPDATE OR DELETE ON dept
DECLARE
    v_action        VARCHAR2(25);
BEGIN
    IF INSERTING THEN
        v_action := ' added department(s) ';
    ELSIF UPDATING THEN
        v_action := ' updated department(s) ';
    ELSIF DELETING THEN
        v_action := ' deleted department(s) ';
    END IF;
    DBMS_ALERT.SIGNAL('dept_alert',USER || v_action || 'on ' ||
        SYSDATE);
END;

CREATE OR REPLACE TRIGGER emp_alert_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_action        VARCHAR2(25);
BEGIN
    IF INSERTING THEN
        v_action := ' added employee(s) ';
    ELSIF UPDATING THEN
        v_action := ' updated employee(s) ';
    ELSIF DELETING THEN
        v_action := ' deleted employee(s) ';
    END IF;
    DBMS_ALERT.SIGNAL('emp_alert',USER || v_action || 'on ' ||
        SYSDATE);
END;
```

The following anonymous block is executed in a session while the dept and emp tables are being updated in other sessions:

```
DECLARE
    v_dept_alert      VARCHAR2(30) := 'dept_alert';
    v_emp_alert       VARCHAR2(30) := 'emp_alert';
    v_name            VARCHAR2(30);
    v_msg             VARCHAR2(80);
    v_status          INTEGER;
    v_timeout         NUMBER(3) := 60;
BEGIN
    DBMS_ALERT.REGISTER(v_dept_alert);
    DBMS_ALERT.REGISTER(v_emp_alert);
    DBMS_OUTPUT.PUT_LINE('Registered for alerts dept_alert and emp_alert');
    DBMS_OUTPUT.PUT_LINE('Waiting for signal...') ;
    LOOP
        DBMS_ALERT.WAITANY(v_name,v_msg,v_status,v_timeout);
        EXIT WHEN v_status ! = 0;
        DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
        DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
        DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
        DBMS_OUTPUT.PUT_LINE('----------------------------------------' ||
            '-------------------------');
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_ALERT.REMOVEALL;
END;

Registered for alerts dept_alert and emp_alert
Waiting for signal...
```

The following changes are made by the user named, mary:

```
INSERT INTO dept VALUES (50,'FINANCE,,,CHICAG0');
INSERT INTO emp (empno,ename,deptno) VALUES (9001,'J0NES',50);
INSERT INTO emp (empno,ename,deptno) VALUES (9002,'ALICE',50);
```

The following change is made by user, john:

```
INSERT INTO dept VALUES (60,'HR','L0S ANGELES');
```

The following example shows the output displayed by the anonymous block that receives the signals from the triggers:

```
Registered for alerts dept_alert and emp_alert
Waiting for signal...
Alert name   : dept_alert
Alert msg    : mary added department(s) on 25-OCT-07 16:41:01
Alert status : 0
-----------------------------------------------------------------
Alert name   : emp_alert
Alert msg    : mary added employee(s) on 25-OCT-07 16:41:02
Alert status : 0
-----------------------------------------------------------------
Alert name   : dept_alert
Alert msg    : john added department(s) on 25-OCT-07 16:41:22
Alert status : 0
-----------------------------------------------------------------
Alert status : 1
```

## 17.3 DBMS_AQ

PolarDB databases compatible with Oracle provide message queuing and message processing. User-defined messages are stored in a queue. A collection of queues is stored in a queue table. Procedures in the DBMS_AQADM package to can be used to create and manage message queues and queue tables. You can use the DBMS_AQ package to add messages to a queue or remove messages from a queue, or register or unregister a PL/SQL callback procedure.

PolarDB databases compatible with Oracle also provide extended non-compatible features for the DBMS_AQ package by running the following SQL commands:

- ALTER QUEUE
- ALTER QUEUE TABLE
- CREATE QUEUE
- CREATE QUEUE TABLE
- DROP QUEUE
- DROP QUEUE TABLE

The DBMS_AQ package provides procedures that allow you to enqueue a message, dequeue a message, and manage callback procedures. The following table lists the supported procedures.

| **Function/Procedure** | **Return Type** | **Description** |
|---|---|---|
| ENQUEUE | N/A | Posts a message to a queue. |
| DEQUEUE | N/A | Retrieves a message from a queue immediately after a message is available. |
| REGISTER | N/A | Registers a callback procedure. |
| UNREGISTER | N/A | Unregisters a callback procedure. |

The implementation of DBMS_AQ in PolarDB databases compatible with Oracle is a partial implementation when compared with native Oracle. Only those procedures listed in the preceding table are supported.

The following table lists the constants supported by PolarDB databases compatible with Oracle.

| **Constant** | **Description** | **Applicable parameter** |
|---|---|---|
| DBMS_AQ.BROWSE (0) | Reads a message without locking. | dequeue_options_t.<br>dequeue_mode |
| DBMS_AQ.LOCKED (1) | This constant is defined. An error message is returned if this constant is used. | dequeue_options_t.<br>dequeue_mode |

| Constant | Description | Applicable parameter |
|---|---|---|
| DBMS_AQ.REMOVE (2) | Deletes a message after reading. This is the default value. | dequeue_options_t. dequeue_mode |
| DBMS_AQ.REMOVE_NODATA (3) | This constant is defined. An error message is returned if this constant is used. | dequeue_options_t. dequeue_mode |
| DBMS_AQ.FIRST_MESSAGE (0) | Returns the first available message that matches the search criteria. | dequeue_options_t. navigation |
| DBMS_AQ.NEXT_MESSAGE (1) | Returns the next available message that matches the search criteria. | dequeue_options_t. navigation |
| DBMS_AQ.NEXT_TRANSACTION (2) | This constant is defined. An error message is returned if this constant is used. | dequeue_options_t. navigation |
| DBMS_AQ.FOREVER (0) | Keeps waiting if a message that matches the search criteria is not found. This is the default value. | dequeue_options_t.wait |
| DBMS_AQ.NO_WAIT (1) | Does not wait if a message that matches the search criteria is not found. | dequeue_options_t.wait |
| DBMS_AQ.ON_COMMIT (0) | Dequeuing is part of the current transaction. | enqueue_options_t.visibility, dequeue_options_t. visibility |
| DBMS_AQ.IMMEDIATE (1) | This constant is defined. An error message is returned if this constant is used. | enqueue_options_t.visibility, dequeue_options_t. visibility |
| DBMS_AQ.PERSISTENT (0) | The message must be stored in a table. | enqueue_options_t. delivery_mode |
| DBMS_AQ.BUFFERED (1) | This constant is defined. An error message is returned if this constant is used. | enqueue_options_t. delivery_mode |
| DBMS_AQ.READY (0) | Specifies that the message is ready to be processed. | message_properties_t.state |
| DBMS_AQ.WAITING (1) | Specifies that the message is waiting to be processed. | message_properties_t.state |

| Constant | Description | Applicable parameter |
|---|---|---|
| DBMS_AQ.PROCESSED (2) | Specifies that the message has been processed. | message_properties_t.state |
| DBMS_AQ.EXPIRED (3) | Specifies that the message is in an exception queue. | message_properties_t.state |
| DBMS_AQ.NO_DELAY (0) | This constant is defined. An error message is returned if this constant is used. | message_properties_t.delay |
| DBMS_AQ.NEVER (NULL) | This constant is defined. An error message is returned if this constant is used. | message_properties_t. expiration |
| DBMS_AQ.NAMESPACE_AQ (0 ) | Accepts notifications from DBMS_AQ queues. | sys.aq$_reg_info.namespace |
| DBMS_AQ.NAMESPACE_ ANONYMOUS (1) | This constant is defined. An error message is returned if this constant is used. | sys.aq$_reg_info.namespace |

**ENQUEUE**

You can use the ENQUEUE procedure to add an entry to a queue. The procedure has the following signature:

```
ENQUEUE(
  queue_name IN VARCHAR2,
  enqueue_options IN DBMS_AQ.ENQUEUE_OPTIONS_T,
  message_properties IN DBMS_AQ.MESSAGE_PROPERTIES_T,
  payload IN <type_name>,
  msgid OUT RAW)
```

**Parameters**

- queue_name

  The name of an existing queue. This may be a schema-qualified name. If you omit the schema name, the server uses the schema specified by SEARCH_PATH. Different from native Oracle, unquoted identifiers are converted to be lowercase before the identifiers are stored. To include special characters or use a case-sensitive name, enclose the name in double quotation marks.

- enqueue_options

  The enqueue_options parameter is a parameter of the enqueue_options_t type. The following example shows the structure of enqueue_options_t:

```
DBMS_AQ.ENQUEUE_OPTIONS_T IS RECORD(
  visibility BINARY_INTEGER DEFAULT ON_COMMIT,
  relative_msgid RAW(16) DEFAULT NULL,
  sequence_deviation BINARY INTEGER DEFAULT NULL,
  transformation VARCHAR2(61) DEFAULT NULL,
  delivery_mode PLS_INTEGER NOT NULL DEFAULT PERSISTENT);
```

  The following table lists the only parameter values supported by enqueue_options_t.

| | |
|---|---|
| visibility | ON_COMMIT |
| delivery_mode | PERSISTENT |
| sequence_deviation | NULL |
| transformation | NULL |
| relative_msgid | NULL |

- message_properties

  The message_properties parameter is a parameter of the message_properties_t type. The following example shows the structure of message_properties_t:

```
    message_properties_t IS RECORD(
  priority INTEGER,
  delay INTEGER,
  expiration INTEGER,
  correlation CHARACTER VARYING(128) COLLATE pg_catalog." C",
  attempts INTEGER,
  recipient_list"AQ$_RECIPIENT_LIST_T",
  exception_queue CHARACTER VARYING(61) COLLATE pg_catalog." C",
  enqueue_time TIMESTAMP WITHOUT TIME ZONE,
    state INTEGER,
   original_msgid BYTEA,
    transaction_group CHARACTER VARYING(30) COLLATE pg_catalog." C",
    delivery_mode INTEGER
  DBMS_AQ.PERSISTENT);
```

  The following table lists the values supported by message_properties_t.

| Parameter | Description |
|---|---|
| priority | If the queue table definition includes sort_list that references priority, this parameter affects the order in which messages are dequeued. A lower value specifies a higher dequeuing priority. |

| Parameter | Description |
|---|---|
| delay | The number of seconds elapsed before a message is available for dequeuing. The NO_DELAY constant specifies that a message is dequeued immediately after the message is available. |
| expiration | The number of seconds elapsed before a message expires. |
| correlation | The message associated with the entry. The default value is NULL. |
| attempts | The number of attempts to dequeue the message. This parameter is maintained by the system. |
| recipient_list | This parameter is not supported. |
| exception_queue | The name of an exception queue to which a message is moved if the message expires or is dequeued by a transaction that rolls back excessive times. |
| enqueue_time | The time when the entry was added to the queue. This value is provided by the system. |
| state | This parameter is maintained by DBMS_AQ. Valid values:<br>- DBMS_AQ.READY: The delay has not been reached.<br>- DBMS_AQ.WAITING: The queue entry is ready for processing.<br>- DBMS_AQ.PROCESSED: The queue entry has been processed.<br>- DBMS_AQ.EXPIRED: The queue entry has been moved to the exception queue . |
| original_msgid | This parameter is ignored, but is included for compatibility. |
| transaction_group | This parameter is ignored, but is included for compatibility. |
| delivery_mode | This parameter is not supported. Specify a value of DBMS_AQ.PERSISTENT. |

- payload

  You can use the payload parameter to provide the data associated with the queue entry . The payload type must match the type specified when you create the corresponding queue table. For more information, see DBMS_AQADM.CREATE_QUEUE_TABLE.

- msgid

  You can use the msgid parameter to retrieve a unique message identifier generated by the system.

**Examples**

The following anonymous block calls DBMS_AQ.ENQUEUE to add a message to a queue named work_order:

```
DECLARE

  enqueue_options    DBMS_AQ.ENQUEUE_OPTIONS_T;
  message_properties DBMS_AQ.MESSAGE_PROPERTIES_T;
  message_handle     raw(16);
  payload            work_order;

BEGIN

  payload := work_order('Smith', 'system upgrade');

DBMS_AQ.ENQUEUE(
  queue_name         => 'work_order',
  enqueue_options    => enqueue_options,
  message_properties => message_properties,
  payload            => payload,
  msgid              => message_handle
    );
 END;
```

**DEQUEUE**

You can use the DEQUEUE procedure to dequeue a message. The procedure has the following signature:

```
DEQUEUE(
  queue_name IN VARCHAR2,
  dequeue_options IN DBMS_AQ.DEQUEUE_OPTIONS_T,
  message_properties OUT DBMS_AQ.MESSAGE_PROPERTIES_T,
  payload OUT type_name,
  msgid OUT RAW)
```

**Parameters**

- queue_name

  The name of an existing queue. This may be a schema-qualified name. If you omit the schema name, the server uses the schema specified by SEARCH_PATH. Different from native Oracle, unquoted identifiers are converted to be lowercase before the identifiers are stored. To include special characters or use a case-sensitive name, enclose the name in double quotation marks.

- dequeue_options

  The dequeue _options parameter is a parameter of the dequeue_options_t type. The following example shows the structure of dequeue_options_t:

```
DEQUEUE_OPTIONS_T IS RECORD(
  consumer_name CHARACTER VARYING(30),
  dequeue_mode INTEGER,
  navigation INTEGER,
  visibility INTEGER,
  wait INTEGER,
  msgid BYTEA,
  correlation CHARACTER VARYING(128),
  deq_condition CHARACTER VARYING(4000),
  transformation CHARACTER VARYING(61),
  delivery_mode INTEGER);
```

  The following table lists the only parameter values supported by dequeue_options_t.

| Parameter | Description |
|---|---|
| consumer_name | Must be NULL. |
| dequeue_mode | The locking behavior of the dequeuing operation. Valid values:<br>- DBMS_AQ.BROWSE: reads a message without obtaining a lock.<br>- DBMS_AQ.LOCKED: reads a message after acquiring a lock.<br>- DBMS_AQ.REMOVE: reads a message before deleting the message.<br>- DBMS_AQ.REMOVE_NODATA: reads a message but does not delete the message. |
| navigation | Specifies the message to be retrieved. Valid values:<br>- FIRST_MESSAGE: the first message within the queue that matches the search criteria.<br>- NEXT_MESSAGE: the next available message that matches the first term. |

| Parameter | Description |
|---|---|
| visibility | Must be ON_COMMIT: If you roll back the current transaction, the dequeued item remains in the queue. |
| wait | Must be a number larger than 0, or be set to:<br>- DBMS_AQ.FOREVER: waits indefinitely.<br>- DBMS_AQ.NO_WAIT: does not wait. |
| msgid | The ID of the message to be dequeued. |
| correlation | This parameter is ignored, but is included for compatibility. |
| deq_condition | A VARCHAR2 expression that calculates a BOOLEAN value and specifies whether a message must be dequeued. |
| transformation | This parameter is ignored, but is included for compatibility. |
| delivery_mode | Must be PERSISTENT. Buffered messages are not supported in this mode. |

- message_properties

  The message_properties parameter is a parameter of the message_properties_t type.

  The following example shows the structure of message_properties_t:

```
message_properties_t IS RECORD(
priority INTEGER,
delay INTEGER,
expiration INTEGER,
correlation CHARACTER VARYING(128) COLLATE pg_catalog." C",
attempts INTEGER,
recipient_list"AQ$_RECIPIENT_LIST_T",
exception_queue CHARACTER VARYING(61) COLLATE pg_catalog." C",
enqueue_time TIMESTAMP WITHOUT TIME ZONE,
state INTEGER,
original_msgid BYTEA,
transaction_group CHARACTER VARYING(30) COLLATE pg_catalog." C",
delivery_mode INTEGER
```

```
  DBMS_AQ.PERSISTENT);
```

The following table lists the parameters supported by message_properties_t:

| Parameter | Description |
|---|---|
| priority | If the queue table definition includes sort_list that references priority, this parameter affects the order in which messages are dequeued. A lower value specifies a higher dequeuing priority. |
| delay | The number of seconds elapsed before a message is available for dequeuing. The NO_DELAY constant specifies that a message is dequeued immediately after the message is available. |
| expiration | The number of seconds elapsed before a message expires. |
| correlation | The message associated with the entry. The default value is NULL. |
| attempts | The number of attempts to dequeue the message. This parameter is maintained by the system. |
| recipient_list | This parameter is not supported. |
| exception_queue | The name of an exception queue to which a message is moved if the message expires or is dequeued by a transaction that rolls back excessive times. |
| enqueue_time | The time when the entry was added to the queue. This value is provided by the system. |

| Parameter | Description |
|---|---|
| state | This parameter is maintained by DBMS_AQ. Valid values:<br>- DBMS_AQ.WAITING: The delay has not been reached.<br>- DBMS_AQ.READY: The queue entry is ready for processing.<br>- DBMS_AQ.PROCESSED: The queue entry has been processed.<br>- DBMS_AQ.EXPIRED: The queue entry has been moved to the exception queue . |
| original_msgid | This parameter is ignored, but is included for compatibility. |
| transaction_group | This parameter is ignored, but is included for compatibility. |
| delivery_mode | This parameter is not supported. Specify a value of DBMS_AQ.PERSISTENT. |

- payload

  You can use the payload parameter to retrieve the payload of a message that is involved in a dequeuing operation. The payload type must match the type specified when you create the queue table.

- msgid

  You can use the msgid parameter to retrieve a unique message identifier.

**Examples**

The following anonymous block calls DBMS_AQ.DEQUEUE to retrieve a message from the queue and payload:

```
DECLARE

  dequeue_options    DBMS_AQ.DEQUEUE_OPTIONS_T;
  message_properties DBMS_AQ.MESSAGE_PROPERTIES_T;
  message_handle     raw(16);
  payload            work_order;

BEGIN
  dequeue_options.dequeue_mode := DBMS_AQ.BROWSE;

  DBMS_AQ.DEQUEUE(
    queue_name         => 'work_queue',
    dequeue_options    => dequeue_options,
    message_properties => message_properties,
   payload          => payload,
   msgid           => message_handle
  );

  DBMS_OUTPUT.PUT_LINE(
  'The next work order is [' || payload.subject || '].'
  );
END;
```

The payload is displayed by DBMS_OUTPUT.PUT_LINE.

### REGISTER

You can use the REGISTER procedure to register an email address, procedure, or URL used for notification when an item is enqueued or dequeued. The procedure has the following signature:

```
REGISTER(
  reg_list IN SYS.AQ$_REG_INFO_LIST,
  count IN NUMBER)
```

**Parameters**

- reg_list

  The reg_list parameter specifies a list of the AQ$_REG_INFO_LIST type. This list provides information about each subscription that you want to register. Each entry within the list is of the AQ$_REG_INFO type. The following table lists the attributes included in each entry.

| Attribute | Type | Description |
|---|---|---|
| name | VARCHAR2 (128) | The name of a subscription. This may be a schema-qualified name. |
| namespace | NUMERIC | The only supported value is DBMS_AQ.NAMESPACE_AQ (0). |

| Attribute | Type | Description |
|---|---|---|
| callback | VARCHAR2 (4000) | Describes the action to be performed upon notification. Only PL/SQL procedures are supported. The procedures are called in this format: `plsql://schema.procedure`, where:<br>- The schema field specifies the schema where the procedure is located.<br>- The procedure field specifies the name of the procedure to be notified. |
| context | RAW (16) | Any user-defined value required by the callback procedure. |

- `count`

  The `count` parameter specifies the number of entries in `reg_list`.

**Examples**

The following anonymous block calls DBMS_AQ.REGISTER to register procedures that are notified when an item is added to or removed from a queue. A set of attributes of the `sys.aq$_reg_info` type is provided for each subscription identified in the DECLARE section:

```
DECLARE
   subscription1 sys.aq$_reg_info;
   subscription2 sys.aq$_reg_info;
   subscription3 sys.aq$_reg_info;
   subscriptionlist sys.aq$_reg_info_list;
BEGIN
   subscription1 := sys.aq$_reg_info('q', DBMS_AQ.NAMESPACE_AQ, 'plsql://assign_wor
ker? PR=0',HEXTORAW('FFFF'));
   subscription2 := sys.aq$_reg_info('q', DBMS_AQ.NAMESPACE_AQ, 'plsql://add_to_his
tory? PR=1',HEXTORAW('FFFF'));
   subscription3 := sys.aq$_reg_info('q', DBMS_AQ.NAMESPACE_AQ, 'plsql://reserve_parts
? PR=2',HEXTORAW('FFFF'));

   subscriptionlist := sys.aq$_reg_info_list(subscription1, subscription2, subscription3);
   dbms_aq.register(subscriptionlist, 3);
   commit;
  END;
```

```
/
```

The subscriptionlist parameter is a parameter of the sys.aq$_reg_info_list type, and contains the sys.aq$_reg_info objects described in this example. The list name and object count are passed to dbms_aq.register.

## UNREGISTER

You can use the UNREGISTER procedure to disable notifications related to enqueuing and dequeuing. The procedure has the following signature:

```
UNREGISTER(
  reg_list IN SYS.AQ$_REG_INFO_LIST,
  count
IN NUMBER)
```

**Parameters**

- reg_list

  The reg_list parameter specifies a list of the AQ$_REG_INFO_LIST type, and provides the information about each subscription that you want to register. Each entry within the list is of the AQ$_REG_INFO type. The following table lists the attributes included in each entry.

| Attribute | Type | Description |
|---|---|---|
| name | VARCHAR2 (128) | The name of a subscription. This may be a schema-qualified name. |
| namespace | NUMERIC | The only supported value is DBMS_AQ.NAMESPACE_AQ (0). |

| Attribute | Type | Description |
|---|---|---|
| callback | VARCHAR2 (4000) | Describes the action to be performed upon notification. Only PL/SQL procedures are supported. The procedures are called in this format: plsql://schema.procedure, where:<br>- The schema field specifies the schema where the procedure is located.<br>- The procedure field specifies the name of the procedure to be notified. |
| context | RAW (16) | Any user-defined value required by the procedure. |

- count

  The count parameter specifies the number of entries in reg_list.

**Examples**

The following anonymous block calls DBMS_AQ.UNREGISTER to disable the notifications specified in the example for DBMS_AQ.REGISTER:

```
DECLARE
  subscription1 sys.aq$_reg_info;
  subscription2 sys.aq$_reg_info;
  subscription3 sys.aq$_reg_info;
  subscriptionlist sys.aq$_reg_info_list;
BEGIN
  subscription1 := sys.aq$_reg_info('q', DBMS_AQ.NAMESPACE_AQ, 'plsql://assign_wor
ker? PR=0',HEXTORAW('FFFF'));
  subscription2 := sys.aq$_reg_info('q', DBMS_AQ.NAMESPACE_AQ, 'plsql://add_to_his
tory? PR=1',HEXTORAW('FFFF'));
  subscription3 := sys.aq$_reg_info('q', DBMS_AQ.NAMESPACE_AQ, 'plsql://reserve_parts
? PR=2',HEXTORAW('FFFF'));

  subscriptionlist := sys.aq$_reg_info_list(subscription1, subscription2, subscription3);
  dbms_aq.unregister(subscriptionlist, 3);
  commit;
 END;
  /
```

The subscriptionlist parameter is a parameter of the sys.aq$_reg_info_list type, and contains the sys.aq$_reg_info objects described in this example. The list name and object count are passed to dbms_aq.unregister.

## 17.4 DBMS_AQADM

PolarDB databases compatible with Oracle provide message queueing and message processing. User-defined messages are stored in a queue. A collection of queues is stored in a queue table. Procedures in the DBMS_AQADM package can be used to create and manage message queues and queue tables. You can use the DBMS_AQ package to add messages to a queue or remove messages from a queue, or register or unregister a PL/SQL callback procedure.

PolarDB databases compatible with Oracle also provide extended non-compatible features for the DBMS_AQ package by running the following SQL commands:

- ALTER QUEUE
- ALTER QUEUE TABLE
- CREATE QUEUE
- CREATE QUEUE TABLE
- DROP QUEUE
- DROP QUEUE TABLE

The DBMS_AQADM package provides stored procedures that allow you to create and manage queues and queue tables.

| Function/Procedure | Return type | Description |
|---|---|---|
| ALTER_QUEUE | N/A | Modifies an existing queue. |
| ALTER_QUEUE_TABLE | N/A | Modifies an existing queue table. |
| CREATE_QUEUE | N/A | Creates a queue. |
| CREATE_QUEUE_TABLE | N/A | Creates a queue table. |
| DROP_QUEUE | N/A | Drops an existing queue. |
| DROP_QUEUE_TABLE | N/A | Drops an existing queue table. |
| PURGE_QUEUE_TABLE | N/A | Removes one or more messages from a queue table. |
| START_QUEUE | N/A | Makes a queue available for enqueuing and dequeuing procedures. |

| Function/Procedure | Return type | Description |
|---|---|---|
| STOP_QUEUE | N/A | Makes a queue unavailable for enqueuing and dequeuing procedures. |

The implementation of DBMS_AQADM in PolarDB databases compatible with Oracle is a partial implementation when compared with native Oracle. Only those functions and procedures listed in the preceding table are supported.

The following table lists the constants supported by PolarDB databases compatible with Oracle.

| Constant | Description | Applicable parameter |
|---|---|---|
| DBMS_AQADM.TRANSACTIONAL(1) | This constant is defined. An error message is returned if this constant is used. | message_grouping |
| DBMS_AQADM.NONE(0) | Specifies message grouping for a queue table. | message_grouping |
| DBMS_AQADM.NORMAL_QUEUE(0) | Used with create_queue to specify queue_type. | queue_type |
| DBMS_AQADM.EXCEPTION_QUEUE (1) | Used with create_queue to specify queue_type. | queue_type |
| DBMS_AQADM.INFINITE(-1) | Used with create_queue to specify retention_time. | retention_time |
| DBMS_AQADM.PERSISTENT (0) | The message must be stored in a table. | enqueue_options_t.delivery_mode |
| DBMS_AQADM.BUFFERED (1) | This constant is defined. An error message is returned if this constant is used. | enqueue_options_t.delivery_mode |
| DBMS_AQADM.PERSISTENT_OR_BUFFERED (2) | This constant is defined. An error message is returned if this constant is used. | enqueue_options_t.delivery_mode |

### ALTER_QUEUE

You can use the ALTER_QUEUE procedure to modify an existing queue. The procedure has the following signature:

```
ALTER_QUEUE(
  max_retries IN NUMBER DEFAULT NULL,
  retry_delay IN NUMBER DEFAULT 0
  retention_time IN NUMBER DEFAULT 0,
  auto_commit IN BOOLEAN DEFAULT TRUE)
  comment IN VARCHAR2 DEFAULT NULL,
```

**Parameters**

| Parameter | Description |
|---|---|
| queue_name | The name of the new queue. |
| max_retries | The maximum number of failed attempts allowed before a message is removed with the DEQUEUE statement. The value of max_retries is incremented with each ROLLBACK statement. When the number of failed attempts reaches the value specified by max_retries, the message is moved to the exception queue. A value of 0 means that no retries are allowed. |
| retry_delay | The number of seconds elapsed between a rollback and message scheduling for re-processing. A value of 0 means that the message must be re-processed immediately. This is the default value. |
| retention_time | The number of seconds elapsed between dequeuing and storage for a message. A value of 0 means that the message cannot be retained after being dequeued. A value of INFINITE means that a message is retained forever. Default value: 0. |
| auto_commit | This parameter is ignored, but is included for compatibility. |
| comment | A comment associated with a queue. |

**Examples**

The following command is used to alter a queue named work_order and set the retry_delay parameter to 5 seconds:

```
EXEC DBMS_AQADM.ALTER_QUEUE(queue_name => 'work_order', retry_delay => 5);
```

### ALTER_QUEUE_TABLE

You can use the ALTER_QUEUE_TABLE procedure to modify an existing queue table. The procedure has the following signature:

```
ALTER_QUEUE_TABLE (
  queue_table IN VARCHAR2,
  comment IN VARCHAR2 DEFAULT NULL,
  primary_instance IN BINARY_INTEGER DEFAULT 0,
  secondary_instance IN BINARY_INTEGER DEFAULT 0,
```

**Parameters**

| Parameter | Description |
|---|---|
| queue_table | The name of a queue table. This may be a schema-qualified name. |
| comment | A comment about a queue table. |
| primary_instance | This parameter is ignored, but is included for compatibility. |
| secondary_instance | This parameter is ignored, but is included for compatibility. |

**Examples**

The following command is used to modify a queue table named work_order_table:

```
EXEC DBMS_AQADM.ALTER_QUEUE_TABLE
     (queue_table => 'work_order_table', comment => 'This queue table contains work
orders for the shipping department.') ;
```

The name of the queue table is work_order_table. The command is used to add a comment to the definition of the queue table.

### CREATE_QUEUE

You can use the CREATE_QUEUE procedure to create a queue in an existing queue table. The procedure has the following signature:

```
CREATE_QUEUE(
  queue_name IN VARCHAR2
  queue_table IN VARCHAR2,
  queue_type IN BINARY_INTEGER DEFAULT NORMAL_QUEUE,
  max_retries IN NUMBER DEFAULT 5,
retry_delay IN NUMBER DEFAULT 0
retention_time IN NUMBER DEFAULT 0,
dependency_tracking IN BOOLEAN DEFAULT FALSE,
comment IN VARCHAR2 DEFAULT NULL,
auto_commit IN BOOLEAN DEFAULT TRUE)
```

**Parameters**

| Parameter | Description |
|---|---|
| queue_name | The name of the new queue. |
| queue_table | The name of the table where the new queue is located. |
| queue_type | The type of the new queue. Valid values:<br>• DBMS_AQADM.NORMAL_QUEUE: a normal queue. This is the default value.<br>• DBMS_AQADM.EXCEPTION_QUEUE: an exception queue. An exception queue only supports dequeue operations. |
| max_retries | The maximum number of failed attempts allowed before a message is removed with the DEQUEUE statement. The value of max_retries is incremented with each ROLLBACK statement. When the number of failed attempts reaches the value specified by max_retries, the message is moved to the exception queue. The default value for a system table is 0. The default value for a user-defined table is 5. |
| retry_delay | The number of seconds elapsed between a rollback and message scheduling for re-processing. A value of 0 means that the message must be re-processed immediately. This is the default value. |
| retention_time | The number of seconds elapsed between dequeuing and storage for a message. A value of 0 means that the message cannot be retained after being dequeued. A value of INFINITE means that a message is retained forever. Default value: 0. |
| dependency_tracking | This parameter is ignored, but is included for compatibility. |
| comment | A comment associated with a queue. |

| Parameter | Description |
|---|---|
| auto_commit | This parameter is ignored, but is included for compatibility. |

### Examples

The following anonymous block is used to create a queue named work_order in the work_order_table table:

```
BEGIN
DBMS_AQADM.CREATE_QUEUE ( queue_name => 'work_order', queue_table => '
work_order_table', comment => 'This queue contains pending work orders.') ;
END;
```

## CREATE_QUEUE_TABLE

You can use the CREATE_QUEUE_TABLE procedure to create a queue table. The procedure has the following signature:

```
CREATE_QUEUE_TABLE (
  queue_table IN VARCHAR2,
  queue_payload_type IN VARCHAR2,
  storage_clause IN VARCHAR2 DEFAULT NULL,
  sort_list IN VARCHAR2 DEFAULT NULL,
  multiple_consumers IN BOOLEAN DEFAULT FALSE,
  message_grouping IN BINARY_INTEGER DEFAULT NONE,
  comment IN VARCHAR2 DEFAULT NULL,
  auto_commit IN BOOLEAN DEFAULT TRUE,
  primary_instance IN BINARY_INTEGER DEFAULT 0,
  secondary_instance IN BINARY_INTEGER DEFAULT 0,
  compatible IN VARCHAR2 DEFAULT NULL,
  secure IN BOOLEAN DEFAULT FALSE)
```

### Parameters

| Parameter | Description |
|---|---|
| queue_table | The name of a queue table. This may be a schema-qualified name. |
| queue_payload_type | The user-defined type of the data to be stored in the queue table. To specify a RAW data type, you must create a user-defined type that identifies a RAW type. |

| Parameter | Description |
|---|---|
| storage_clause | Specifies the attributes for the queue table. Only the TABLESPACE option is enforced. All other options are ignored, but are included for compatibility. You can use the TABLESPACE clause to specify the name of a tablespace in which a table is created.<br>• storage_clause can be set to one or more of the following options:<br>TABLESPACE tablespace_name, PCTFREE integer, PCTUSED integer, INITRANS integer, MAXTRANS integer, and STORAGE storage_option.<br>• storage_option can be set to one or more of the following options:<br>MINEXTENTS integer, MAXEXTENTS integer, PCTINCREASE integer, INITIAL size_clause, NEXT, FREELISTS integer, OPTIMAL size_clause, and BUFFER_POOL {KEEP\|RECYCLE\|DEFAULT}. |
| sort_list | This parameter controls the dequeueing order of the queue and specifies the names of the columns that are used to sort the queue in ascending order. The following combinations of enq_time and priority are supported:<br>• enq_time, priority<br>• priority, enq_time<br>• priority<br>• enq_time |
| multiple_consumers | This parameter must be set to FALSE if required. |
| message_grouping | This parameter must be set to NONE if required. |
| comment | A comment about a queue table. |
| auto_commit | This parameter is ignored, but is included for compatibility. |

| Parameter | Description |
|---|---|
| primary_instance | This parameter is ignored, but is included for compatibility. |
| secondary_instance | This parameter is ignored, but is included for compatibility. |
| compatible | This parameter is ignored, but is included for compatibility. |
| secure | This parameter is ignored, but is included for compatibility. |

**Examples**

The following anonymous block is used to create the work_order type with the attributes that hold the VARCHAR2 name and the TEXT project description. Then, the block uses this type to create a queue table.

```
BEGIN

CREATE TYPE work_order AS (name VARCHAR2, project TEXT, completed BOOLEAN);

EXEC DBMS_AQADM.CREATE_QUEUE_TABLE
      (queue_table => 'work_order_table',
       queue_payload_type => 'work_order',
       comment => 'Work order message queue table');
END;
```

The queue table is named work_order_table and contains a payload of the work_order type . A comment is added to indicate that this is the work order message queue table.

### DROP_QUEUE

You can use the DROP_QUEUE procedure to drop a queue. The procedure has the following signature:

```
DROP_QUEUE(
  queue_name  IN VARCHAR2,
  auto_commit IN BOOLEAN DEFAULT TRUE)
```

**Parameters**

| Parameter | Description |
|---|---|
| queue_name | The name of the queue that you want to drop. |
| auto_commit | This parameter is ignored, but is included for compatibility. |

### Examples

The following anonymous block drops the queue named work_order:

```
BEGIN
DBMS_AQADM.DROP_QUEUE(queue_name => 'work_order');
END;
```

## DROP_QUEUE_TABLE

You can use the DROP_QUEUE_TABLE procedure to drop a queue table. The procedure has the following signature:

```
DROP_QUEUE_TABLE(
  queue_table IN VARCHAR2,
  force IN BOOLEAN default FALSE,
  auto_commit IN BOOLEAN default TRUE)
```

### Parameters

| Parameter | Description |
|---|---|
| queue_table | The name of a queue table. This may be a schema-qualified name. |
| force | The force keyword specifies the behavior of the DROP_QUEUE_TABLE command when the command is used to drop a table that contain entries:<br>• If the target table contains entries and force is set to FALSE, the command fails and an error message is returned.<br>• If the target table contains entries and force is set to TRUE, the command drops the table and all dependent objects. |
| auto_commit | This parameter is ignored, but is included for compatibility. |

### Examples

The following anonymous block is used to drop a table named work_order_table:

```
BEGIN
   DBMS_AQADM.DROP_QUEUE_TABLE ('work_order_table', force => TRUE);
```

```
END;
```

### PURGE_QUEUE_TABLE

You can use the PURGE_QUEUE_TABLE procedure to delete messages from a queue table.

The procedure has the following signature:

```
PURGE_QUEUE_TABLE(
  queue_table IN VARCHAR2,
  purge_condition IN VARCHAR2,
  purge_options IN aq$_purge_options_t)
```

#### Parameters

| Parameter | Description |
|---|---|
| queue_table | The name of the queue table from which you want to delete a message. |
| purge_condition | Specifies as the condition that the server evaluates when the server determines the messages to be deleted. The condition is specified in a SQL WHERE clause. |
| purge_options | An object of the aq$_purge_options_t type. An aq$_purge_options_t object contains certain attributes. For more information, see Table 17-2: aq$_purge_options_t. |

Table 17-2: aq$_purge_options_t

| Attribute | Type | Description |
|---|---|---|
| Block | Boolean | A value of TRUE means that an exclusive lock must be held on all queues within the table. Default value: FALSE. |
| delivery_mode | INTEGER | Specifies the type of message to be deleted. The only supported value is dbms_aq.percent. |

#### Examples

The following anonymous block is used to remove any messages from work_order_table where the value of the column named completed is YES:

```
DECLARE
  purge_options dbms_aqadm.aq$_purge_options_t;
```

```
BEGIN
   dbms_aqadm.purge_queue_table('work_order_table', 'completed = YES', purge_opti
ons);
  END;
```

### START_QUEUE

You can use the START_QUEUE procedure to make a queue available for enqueuing and dequeuing. The procedure has the following signature:

```
START_QUEUE(
  queue_name IN VARCHAR2,
  enqueue IN BOOLEAN DEFAULT TRUE,
  dequeue IN BOOLEAN DEFAULT TRUE)
```

**Parameters**

| Parameter | Description |
|---|---|
| queue_name | The name of the queue that you want to start. |
| enqueue | A value of TRUE means that enqueuing is enabled. A value of FALSE means that the current setting is unchanged. Default value: TRUE. |
| dequeue | A value of TRUE means that dequeuing is enabled. A value of FALSE means that the current setting is unchanged. Default value: TRUE. |

**Examples**

The following anonymous block is used to make a queue named work_order available for enqueuing:

```
BEGIN
DBMS_AQADM.START_QUEUE
(queue_name => 'work_order);
END;
```

### STOP_QUEUE

You can use the STOP_QUEUE procedure to disable enqueuing or dequeuing on a specified queue. The procedure has the following signature:

```
STOP_QUEUE(
  queue_name IN VARCHAR2,
  enqueue IN BOOLEAN DEFAULT TRUE,
  dequeue IN BOOLEAN DEFAULT TRUE,
  wait IN BOOLEAN DEFAULT TRUE)
```

**Parameters**

| Parameter | Description |
|---|---|
| queue_name | The name of the queue that you want to stop. |
| enqueue | A value of TRUE means that enqueuing is disabled. A value of FALSE means that the current setting is unchanged. Default value: TRUE. |
| dequeue | A value of TRUE means that dequeuing is disabled. A value of FALSE means that the current setting is unchanged. Default value: TRUE. |
| wait | A value of TRUE means that the server waits for any uncompleted transactions to complete before the server applies the specified changes. When the server waits to stop the queue, no transactions are allowed to be enqueued to or dequeued from the specified queue. A value of FALSE means that the queue is stopped immediately. |

**Examples**

The following anonymous block is used disable enqueuing to and dequeuing from the queue named work_order:

```
BEGIN
DBMS_AQADM.STOP_QUEUE(queue_name =>'work_order', enqueue=>TRUE, dequeue=>
TRUE, wait=>TRUE);
END;
```

Enqueuing and dequeuing are stopped after all outstanding transactions are completed.

## 17.5 DBMS_CRYPTO

You can use the functions and stored procedures in the DBMS_CRYPTO package to encrypt or decrypt RAW, BLOB, or CLOB data. You can also use DBMS_CRYPTO functions to generate cryptographically secure random values.

**Table 17-3: DBMS_CRYPTO functions and stored procedures**

| Function/stored procedure | Return type | Description |
|---|---|---|
| DECRYPT(src, typ, key, iv) | RAW | Decrypts RAW data. |
| DECRYPT(dst INOUT, src, typ, key, iv) | N/A | Decrypts BLOB data. |
| DECRYPT(dst INOUT, src, typ, key, iv) | N/A | Decrypts CLOB data. |
| ENCRYPT(src, typ, key, iv) | RAW | Encrypts RAW data. |
| ENCRYPT(dst INOUT, src, typ, key, iv) | N/A | Encrypts BLOB data. |
| ENCRYPT(dst INOUT, src, typ, key, iv) | N/A | Encrypts CLOB data. |
| HASH(src, typ) | RAW | Applies a hash algorithm to RAW data. |
| HASH(src) | RAW | Applies a hash algorithm to CLOB data. |
| MAC(src, typ, key) | RAW | Returns the hashed MAC value of the given RAW data . The hash algorithm and key are user-specified. |
| MAC(src, typ, key) | RAW | Returns the hashed MAC value of the given CLOB data . The hash algorithm and key are user-specified. |
| RANDOMBYTES(number bytes) | RAW | Returns a specified number of cryptographically secure random bytes. |
| RANDOMINTEGER() | INTEGER | Returns a random integer. |
| RANDOMNUMBER() | NUMBER | Returns a random number. |

Similar to Oracle databases, POLARDB compatible with Oracle supports the following error messages:

```
ORA-28239-DBMS_CRYPTO.KeyNull
ORA-28829-DBMS_CRYPTO.CipherSuiteNull
ORA-28827-DBMS_CRYPTO.CipherSuiteInvalid
```

Different from Oracle databases, POLARDB compatible with Oracle will not return error ORA-28233 if you re-encrypt previously encrypted information.

Note that RAW and BLOB are synonyms of PostgreSQL BYTEA data types, while CLOB is a synonym of TEXT.

**DECRYPT**

You can use the DECRYPT function or stored procedure to decrypt data based on a specified encryption algorithm, key, and optional initialization vector. The following code describes the syntax of the DECRYPT function:

```
DECRYPT
  (src IN RAW, typ IN INTEGER, key IN RAW, iv IN RAW
   DEFAULT NULL) RETURN RAW
```

The following code describes the syntax of the DECRYPT stored procedure:

```
DECRYPT
  (dst INOUT BLOB, src IN BLOB, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL)
```

Or

```
DECRYPT
  (dst INOUT CLOB, src IN CLOB, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL)
```

When DECRYPT is called as a stored procedure, DECRYPT returns BLOB or CLOB data to the user-specified BLOB.

**Parameters**

dst

Specifies the name of a BLOB. The DECRYPT stored procedure writes the output into the BLOB and overwrites any existing data in the BLOB.

src

Specifies the source data to be decrypted. If you call DECRYPT as a function, you must specify RAW data. If you call DECRYPT as a stored procedure, you must specify BLOB or CLOB data.

typ

Specifies the block cipher type and any modifiers. The value of the parameter must match the type specified when the source data was encrypted. POLARDB compatible with Oracle supports the following block cipher algorithms, modifiers, and cipher suites.

| **Block cipher algorithms** | |
|---|---|
| ENCRYPT_DES | CONSTANT INTEGER := 1; |
| ENCRYPT_3DES | CONSTANT INTEGER := 3; |
| ENCRYPT_AES | CONSTANT INTEGER := 4; |
| ENCRYPT_AES128 | CONSTANT INTEGER := 6; |
| **Block cipher modifiers** | |
| CHAIN_CBC | CONSTANT INTEGER := 256; |
| CHAIN_ECB | CONSTANT INTEGER := 768; |
| **Block cipher padding modifiers** | |
| PAD_PKCS5 | CONSTANT INTEGER := 4096; |
| PAD_NONE | CONSTANT INTEGER := 8192; |
| **Block cipher suites** | |
| DES_CBC_PKCS5 | CONSTANT INTEGER := ENCRYPT_DES + CHAIN_CBC + PAD_PKCS5; |
| DES3_CBC_PKCS5 | CONSTANT INTEGER := ENCRYPT_3DES + CHAIN_CBC + PAD_PKCS5; |
| AES_CBC_PKCS5 | CONSTANT INTEGER := ENCRYPT_AES + CHAIN_CBC + PAD_PKCS5; |

key

Specifies the user-defined decryption key. The value of the parameter must match the key specified when the source data was encrypted.

iv

Optional. Specifies the initialization vector. If you specify an initialization vector when encrypting the source data, you must specify the parameter when decrypting the source data. The default value is NULL.

**Example**

The following example uses the DBMS_CRYPTO.DECRYPT function to decrypt the encrypted password that is retrieved from the passwords table.

```
CREATE TABLE passwords
(
  principal  VARCHAR2(90) PRIMARY KEY,  -- username
  ciphertext RAW(9)                     -- encrypted password
);
CREATE FUNCTION get_password(username VARCHAR2) RETURN RAW AS
 typ       INTEGER := DBMS_CRYPTO.DES_CBC_PKCS5;
 key       RAW(128) := 'my secret key';
 iv        RAW(100) := 'my initialization vector';
 password   RAW(2048);
BEGIN

  SELECT ciphertext INTO password FROM passwords WHERE principal = username;

  RETURN dbms_crypto.decrypt(password, typ, key, iv);
END;
```

Note that when you call DECRYPT, you must pass the same password type, key value, and initialization vector used when you encrypted the object.

### ENCRYPT

You can use the ENCRYPT function or stored procedure to encrypt RAW, BLOB, or CLOB data based on a user-defined algorithm, key, and optional initialization vector. The following code describes the syntax of the DECRYPT function:

```
ENCRYPT
  (src IN RAW, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL) RETURN RAW
```

The following code describes the syntax of the DECRYPT stored procedure:

```
ENCRYPT
  (dst INOUT BLOB, src IN BLOB, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL)
```

Or

```
ENCRYPT
  (dst INOUT BLOB, src IN CLOB, typ IN INTEGER, key IN RAW,
    iv IN RAW DEFAULT NULL)
```

When you call ENCRYPT as a stored procedure, ENCRYPT returns BLOB or CLOB data to the user-specified BLOB.

**Parameters**

dst

Specifies the name of a BLOB. The ENCRYPT stored procedure writes the output into the BLOB and overwrites any existing data in the BLOB.

src

Specifies the source data to be encrypted. If you call ENCRYPT as a function, you must specify RAW data. If you call ENCRYPT as a stored procedure, you must specify BLOB or CLOB data.

typ

Specifies the block cipher type and any modifiers. POLARDB compatible with Oracle supports the following block cipher algorithms, modifiers, and cipher suites.

| **Block cipher algorithms** | |
|---|---|
| ENCRYPT_DES | CONSTANT INTEGER := 1; |
| ENCRYPT_3DES | CONSTANT INTEGER := 3; |
| ENCRYPT_AES | CONSTANT INTEGER := 4; |
| ENCRYPT_AES128 | CONSTANT INTEGER := 6; |
| **Block cipher modifiers** | |
| CHAIN_CBC | CONSTANT INTEGER := 256; |
| CHAIN_ECB | CONSTANT INTEGER := 768; |
| **Block cipher padding modifiers** | |
| PAD_PKCS5 | CONSTANT INTEGER := 4096; |
| PAD_NONE | CONSTANT INTEGER := 8192; |
| **Block cipher suites** | |
| DES_CBC_PKCS5 | CONSTANT INTEGER := ENCRYPT_DES + CHAIN_CBC + PAD_PKCS5; |
| DES3_CBC_PKCS5 | CONSTANT INTEGER := ENCRYPT_3DES + CHAIN_CBC + PAD_PKCS5; |

| | |
|---|---|
| AES_CBC_PKCS5 | CONSTANT INTEGER := ENCRYPT_AES + CHAIN_CBC + PAD_PKCS5; |

key

Specifies the encryption Key.

iv

Optional. Specifies the initialization vector. The default value is NULL.

**Example**

The following example uses the DBMS_CRYPTO.DES_CBC_PKCS5 block cipher suite (a set of predefined algorithms and modifiers) to encrypt the value that is retrieved from the passwords table:

```
CREATE TABLE passwords
(
  principal  VARCHAR2(90) PRIMARY KEY,  -- username
  ciphertext RAW(9)                  -- encrypted password
);
CREATE PROCEDURE set_password(username VARCHAR2, cleartext RAW) AS
 typ       INTEGER := DBMS_CRYPTO.DES_CBC_PKCS5;
 key       RAW(128) := 'my secret key';
 iv        RAW(100) := 'my initialization vector';
 encrypted  RAW(2048);
BEGIN
  encrypted := dbms_crypto.encrypt(cleartext, typ, key, iv);
  UPDATE passwords SET ciphertext = encrypted WHERE principal = username;
END;
```

During password encryption, ENCRYPT uses "my secret key" as the key and "my initialization vector" as the initialization vector. You must use the same key and initialization vector when decrypting the password.

### HASH

You can use the HASH function to return the hash values of RAW or CLOB data. The hash algorithm is user-specified. The HASH function supports the following syntax:

```
HASH
  (src IN RAW, typ IN INTEGER) RETURN RAW
HASH
  (src IN CLOB, typ IN INTEGER) RETURN RAW
```

**Parameters**

src

Specifies the data for which the hash value will be calculated. The RAW, BLOB, or CLOB data types are supported.

typ

Specifies the hash algorithm. POLARDB compatible with Oracle supports the following hash algorithms.

| Hash algorithms | |
|---|---|
| HASH_MD4 | CONSTANT INTEGER := 1; |
| HASH_MD5 | CONSTANT INTEGER := 2; |
| HASH_SH1 | CONSTANT INTEGER := 3; |

**Example**

The following example uses DBMS_CRYPTO.HASH to retrieve the MD5 hash value of the " cleartext source" string:

```
DECLARE
  typ       INTEGER := DBMS_CRYPTO.HASH_MD5;
  hash_value RAW(100);
BEGIN

  hash_value := DBMS_CRYPTO.HASH('cleartext source', typ);

END;
```

**MAC**

You can use a specified MAC function to return the hashed MAC value of RAW or CLOB data. The HASH function supports the following syntax:

```
MAC
  (src IN RAW, typ IN INTEGER, key IN RAW) RETURN RAW
MAC
  (src IN CLOB, typ IN INTEGER, key IN RAW) RETURN RAW
```

**Parameters**

src

Specifies the data for which the hash value will be calculated. The RAW, BLOB, or CLOB data types are supported.

typ

Specifies the MAC function type. POLARDB compatible with Oracle supports the following MAC function types.

| MAC functions | |
|---|---|
| HMAC MD5 | CONSTANT INTEGER := 1; |

| MAC functions | |
|---|---|
| HMAC SH1 | CONSTANT INTEGER := 2; |

key

Specifies the key that is used to calculate the hashed MAC value.

**Example**

The following example uses DBMS_CRYPTO.MAC to retrieve the hash value of the "cleartext source" string:

```
DECLARE
  typ       INTEGER := DBMS_CRYPTO.HMAC_MD5;
  key       RAW(100) := 'my secret key';
  mac_value RAW(100);
BEGIN

  mac_value := DBMS_CRYPTO.MAC('cleartext source', typ, key);

END;
```

During the calculation, DBMS_CRYPTO.MAC uses "my secret key" as the key.

### RANDOMBYTES

You can use the RANDOMBYTES function to return a RAW value that contains cryptographically random bytes. You can specify the length for the RAW value. The following code describes the syntax of the RANDOMBYTES function:

```
RANDOMBYTES
  (number_bytes IN INTEGER) RETURNS RAW
```

**Parameters**

number_bytes

Specifies the number of random bytes that are returned by the function.

**Example**

The following example uses RANDOMBYTES to return a value that is 1,024 bytes in length:

```
DECLARE
  result RAW(1024);
BEGIN
  result := DBMS_CRYPTO.RANDOMBYTES(1024);
```

```
END;
```

### RANDOMINTEGER

You can use the RANDOMINTEGER function to return a random integer between 0 and 268, 435, or 455. The following code describes the syntax of the RANDOMINTEGER function:

```
RANDOMINTEGER() RETURNS INTEGER
```

**Example**

The following example uses the RANDOMINTEGER function to return a cryptographically secure random integer:

```
DECLARE
  result INTEGER;
BEGIN
  result := DBMS_CRYPTO.RANDOMINTEGER();
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

### RANDOMNUMBER

You can use the RANDOMNUMBER function to return a random number between 0 and 268, 435, or 455. The following code describes the syntax of the RANDOMNUMBER function:

```
RANDOMNUMBER() RETURNS NUMBER
```

**Example**

The following example uses the RANDOMINTEGER function to return a cryptographically secure random number:

```
DECLARE
  result NUMBER;
BEGIN
  result := DBMS_CRYPTO.RANDOMNUMBER();
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

## 17.6 DBMS_LOB

The DBMS_LOB package is used to perform operations on large objects.

**Table 17-4: DBMS_LOB functions and stored procedures**

| Function/stored procedure | Function or stored procedure | Return type | Description |
|---|---|---|---|
| APPEND(dest_lob IN OUT,src_lob) | Stored procedure | N/A | Appends a large object to another. |
| COMPARE(lob_1, lob_2 [, amount[, offset_1 [, offset_2 ]]]) | Function | INTEGER | Compares two large objects. |
| CONVERTOBLOB( dest_lob IN OUT, src_clob, amount , dest_offsetIN OUT, src_offset IN OUT,blob_csid, lang_context IN OUT, warning OUT) | Stored procedure | N/A | Converts characters to binary data. |
| CONVERTTOCLOB (dest_lob IN OUT, src_blob, amount , dest_offsetIN OUT, src_offset IN OUT,blob_csid, lang_context IN OUT, warning OUT) | Stored procedure | N/A | Converts binary data to characters. |
| COPY(dest_lob IN OUT, src_lob,amount [, dest_offset [, src_offset ]]) | Stored procedure | N/A | Copies a large object to another one. |
| ERASE(lob_loc IN OUT, amount IN OUT [, offset ]) | Stored procedure | N/A | Erases a large object. |
| GET_STORAGE_LIMIT( lob_loc) | Function | INTEGER | Retrieves the storage limit for large objects . |
| GETLENGTH(lob_loc) | Function | INTEGER | Retrieves the length of the large object. |

| Function/stored procedure | Function or stored procedure | Return type | Description |
|---|---|---|---|
| INSTR(lob_loc, pattern [,offset [, nth ]]) | Function | INTEGER | Retrieves the position of a pattern in the large object that starts at the specified offset. |
| READ(lob_loc, amount IN OUT,offset , buffer OUT) | Stored procedure | N/A | Reads a large object. |
| SUBSTR(lob_loc [, amount [,offset ]]) | Function | RAW or VARCHAR2 | Retrieves a portion of a large object. |
| TRIM(lob_loc IN OUT, newlen) | Stored procedure | N/A | Trims a large object to the specified length. |
| WRITE(lob_loc IN OUT, amount,offset, buffer) | Stored procedure | N/A | Writes data to a large object. |
| WRITEAPPEND( lob_loc IN OUT, amount, buffer) | Stored procedure | N/A | Writes data from the buffer to the end of a large object. |

The DBMS_SQL package in POLARDB compatible with Oracle is only partially implemented when compared to Oracle's version. POLARDB compatible with Oracle only supports the functions and stored procedures that are listed in the preceding table.

The following table lists the public variables that can be used in the package.

**Table 17-5: DBMS_LOB public variables**

| Public variable | Data type | Value |
|---|---|---|
| compress off | INTEGER | 0 |
| compress_on | INTEGER | 1 |
| deduplicate_off | INTEGER | 0 |
| deduplicate_on | INTEGER | 4 |
| default_csid | INTEGER | 0 |
| default_lang_ctx | INTEGER | 0 |
| encrypt_off | INTEGER | 0 |

| Public variable | Data type | Value |
|---|---|---|
| encrypt_on | INTEGER | 1 |
| file_readonly | INTEGER | 0 |
| lobmaxsize | INTEGER | 1073741823 |
| lob_readonly | INTEGER | 0 |
| lob_readwrite | INTEGER | 1 |
| no_warning | INTEGER | 0 |
| opt_compress | INTEGER | 1 |
| opt_deduplicate | INTEGER | 4 |
| opt_encrypt | INTEGER | 2 |
| warn_inconvertible_char | INTEGER | 1 |

In the following sections, if the data type of a large object is BLOB, the length and offset of the object are measured in bytes. If the data type of a large object is CLOB, the length and offset are measured in characters.

### APPEND

The APPEND stored procedure is used to append a large object to another. The data types of the two large objects must be the same.

```
APPEND(dest_lob IN OUT { BLOB | CLOB }, src_lob { BLOB | CLOB })
```

**Parameters**

| Parameter | Description |
|---|---|
| dest_lob | Specifies the location of the target large object. The data type of the dest_lob parameter must be the same as that of the src_lob parameter. |
| src_lob | Specifies the location of the source large object. The data type of the src_lob parameter must be the same as that of the dest_lob parameter. |

### COMPARE

The COMPARE stored procedure compares two large objects by byte at the specified offsets within the specified length. The data types of the two large objects that are compared must be the same.

```
status INTEGER COMPARE(lob_1 { BLOB | CLOB },
  lob_2 { BLOB | CLOB }
  [, amount INTEGER [, offset_1 INTEGER [, offset_2 INTEGER ]]])
```

**Parameters**

| Parameter | Description |
|---|---|
| lob_1 | Specifies the location of the first large object. The data type of the lob_1 parameter must be the same as that of the lob_2 parameter. |
| lob_2 | Specifies the location of the second large object. The data type of the lob_2 parameter must be the same as that of the lob_1 parameter. |
| amount | If the data types of large objects are BLOB , the objects are compared within the specified amount of bytes. If the data types of large objects are CLOB, the objects are compared within the specified amount of characters. The default value of the amount parameter is the maximum size of a large object. |
| offset 1 | Specifies the position in the first large object to start the comparison. The position of the first byte or character is labeled as offset 1. The default value is 1. |
| offset_2 | Specifies the position in the second large object to start the comparison. The position of the first byte or character is labeled as offset 1. The default value is 1. |

| Parameter | Description |
|---|---|
| status | Specifies the comparison result. If the two large objects are the same at the specified offsets within the specified length, 0 (zero) is returned. If the objects are not the same, a non-zero value is returned. If the value of the amount, offset_1, or offset_2 parameter is smaller than 0, NULL is returned. |

## CONVERTTOBLOB

The CONVERTTOBLOB stored procedure is used to convert a large object of the CLOB data type into a large object of the BLOB data type.

```
CONVERTTOBLOB(dest_lob IN OUT BLOB, src_clob CLOB,
  amount INTEGER, dest_offset IN OUT INTEGER,
  src_offset IN OUT INTEGER, blob_csid NUMBER,
  lang_context IN OUT INTEGER, warning OUT INTEGER)
```

**Parameters**

| Parameter | Description |
|---|---|
| dest_lob | Specifies a target large object of the BLOB data type. You can use the CONVERTTOBLOB stored procedure to convert a large object of the CLOB data type into a large object of the BLOB data type. |
| src_clob | Specifies a source large object of the CLOB data type. You can use the CONVERTTOBLOB stored procedure to convert a large object of the BLOB data type into a large object of the CLOB data type. |
| amount | Specifies the number of characters to be converted in the large object specified by the src_clob parameter. |
| dest_offset IN | Specifies the location of the byte in the target large object where writing of the source large object starts. The first byte is labeled as offset 1. |
| dest_offset OUT | Specifies the location of the byte in the target large object after the write operation is complete. The first byte is labeled as offset 1. |

| Parameter | Description |
|---|---|
| src_offset IN | Specifies the location of the character in the source large object where the conversion starts. The first character is labeled as offset 1. |
| src_offset OUT | Specifies the location of the character in the source large object after the conversion is complete. The first character is labeled as offset 1. |
| blob_csid | Specifies the character set ID of the target large object. |
| langcontext IN | Specifies the language environment for the conversion. The default value of 0 is typically used for the setting. |
| langcontext OUT | Specifies the language environment after the conversion is complete. |
| warning | If the conversion is successful, 0 is returned. If the conversion fails, 1 is returned. |

### CONVERTTOCLOB

The CONVERTTOCLOB stored procedure is used to convert a large object of the BLOB data type into a large object of the CLOB data type.

```
CONVERTTOCLOB(dest_lob IN OUT CLOB, src_blob BLOB,
  amount INTEGER, dest_offset IN OUT INTEGER,
  src_offset IN OUT INTEGER, blob_csid NUMBER,
  lang_context IN OUT INTEGER, warning OUT INTEGER)
```

#### Parameters

| Parameter | Description |
|---|---|
| dest_lob | Specifies a target large object of the CLOB data type. You can use the CONVERTTOBLOB stored procedure to convert a large object of the BLOB data type into a large object of the CLOB data type. |
| src_blob | Specifies a source large object of the BLOB data type. You can use the CONVERTTOBLOB stored procedure to convert a large object of the CLOB data type into a large object of the BLOB data type. |

| Parameter | Description |
|---|---|
| amount | Specifies the number of bytes to be converted in the large object specified by the src_blob parameter. |
| dest_offset IN | Specifies the location of the character in the target large object where writing of the source large object starts. The first character is labeled as offset 1. |
| dest_offset OUT | Specifies the location of the character in the target large object of the CLOB data type after the write operation is complete. The first character is labeled as offset 1. |
| src_offset IN | Specifies the location of the byte in the source large object where the conversion starts. The first byte is labeled as offset 1. |
| src_offset OUT | Specifies the location of the byte in the source large object after the conversion is complete. The first byte is labeled as offset 1. |
| blob_csid | Specifies the character set ID of the target large object. |
| CLOB. langcontext IN | Specifies the language environment for the conversion. The default value of 0 is typically used for the setting. |
| langcontext OUT | Specifies the language environment after the conversion is complete. |
| warning | If the conversion is successful, 0 is returned. If the conversion fails, 1 is returned. |

### COPY

The COPY stored procedure is used to copy a large object to another. The data types of the source and target large objects must be the same.

```
COPY(dest_lob IN OUT { BLOB | CLOB }, src_lob
{ BLOB | CLOB },
  amount INTEGER
  [, dest_offset INTEGER [, src_offset INTEGER ]])
```

**Parameters**

| Parameter | Description |
|---|---|
| dest_lob | Specifies the location of the target large object to which a source large object is copied. The data type of the parameter must be the same as the large object specified by the src_lob parameter. |
| src_lob | Specifies the location of the target large object to be copied. The data type of the parameter must be the same as the large object specified by the dest_lob parameter. |
| amount | Specifies the number of bytes or characters to be copied in the large object specified by the src_lob parameter. |
| dest_offset | Specifies the location in the target large object where writing of the source large object starts. The first position is labeled as offset 1. The default value is 1. |
| src_offset | Specifies the location of the character in the source large object where the copy operation starts. The first location is labeled as offset 1. The default value is 1. |

### ERASE

The ERASE stored procedure is used to erase a portion of the data in a large object. For a large object of the BLOB data type, the specified portion is replaced with a 0-byte filter. For a large object of the CLOB data type, the specified portion is replaced with spaces. The operation does not change the size of the large object.

```
ERASE(lob_loc IN OUT { BLOB | CLOB }, amount IN OUT INTEGER
  [, offset INTEGER ])
```

**Parameters**

| Parameter | Description |
|---|---|
| lob_loc | Specifies the large object to be erased. |
| amount IN | Specifies the number of bytes or characters to be erased in the large object. |

| Parameter | Description |
|---|---|
| amount OUT | Specifies the number of bytes or characters that have been erased. If the end of the large object is reached before the specified number of bytes or characters has been erased, the output value is smaller than the input value. |
| offset | Specifies the location in the large object from which erasing starts. The first byte or character is labeled as offset 1. The default value is 1. |

### GET_STORAGE_LIMIT

The GET_STORAGE_LIMIT function is used to retrieve the maximum storage space that can be used by large objects.

```
size INTEGER GET_STORAGE_LIMIT(lob_loc BLOB)
size INTEGER GET_STORAGE_LIMIT(lob_loc CLOB)
```

**Parameters**

| Parameter | Description |
|---|---|
| size | Specifies the maximum storage space that can be used by a large object in the database. |
| lob_loc | The parameter is provided to ensure the compatibility with Oracle databases and can be ignored during runtime. |

### GETLENGTH

The GETLENGTH function is used to retrieve the length of a large object.

```
amount INTEGER GETLENGTH(lob_loc BLOB)
amount INTEGER GETLENGTH(lob_loc CLOB)
```

**Parameters**

| Parameter | Description |
|---|---|
| lob_loc | Specifies the location of the large object . You can use the GETLENGTH function to retrieve the length of the object. |
| amount | Specifies the length of the large object. For a large object of the BLOB data type, the length is measured in bytes. For a large object of the CLOB data type, the length is measured in characters. |

## INSTR

The INSTR function is used to retrieve the position where the specified pattern appears for the specified nth number of times in a large object.

```
position INTEGER INSTR(lob_loc { BLOB | CLOB },
  pattern { RAW | VARCHAR2 } [, offset INTEGER [, nth INTEGER ]])
```

### Parameters

| Parameter | Description |
|---|---|
| lob_loc | Specifies the location of the large object in which you can use the INSTR function to search for the specified pattern. |
| pattern | Specifies the pattern to match in the large object. The pattern is a combination of bytes or characters. If the data type of a large object is BLOB, the data type of the pattern must be RAW. If the data type of a large object is CLOB, the data type of the pattern must be VARCHAR2. |
| offset | Specifies the position to start searching for the pattern in the large object specified by the lob_loc parameter. The first byte or character is labeled as offset 1. The default value is 1. |
| nth | Specifies the nth number of times when the pattern appears starting from the specified offset. The default value is 1. |

| Parameter | Description |
|---|---|
| position | Specifies the position where the pattern appears for the specified nth time in the large object. The search starts from the specified offset. |

### READ

The READ stored procedure is used to read a portion of a large object into a buffer.

```
READ(lob_loc { BLOB | CLOB }, amount IN OUT BINARY_INTEGER,
  offset INTEGER, buffer OUT { RAW | VARCHAR2 })
```

**Parameters**

| Parameter | Description |
|---|---|
| lob_loc | Specifies the location of the large object to be read. |
| amount IN | Specifies the total number of bytes or characters to be read. |
| amount OUT | Specifies the total number of bytes or characters that are read. If no more data is available for reading, 0 is returned and the DATA_NOT_FOUND exception is thrown. |
| offset | Specifies the location where the read operation starts in the large object. The first byte or character is labeled as offset 1. |
| buffer | Specifies the variable that receives the portion of the large object. If the data type of the lob_loc parameter is BLOB, the data type of the buffer parameter must be RAW. If the data type of the lob_loc parameter is CLOB, the data type of the buffer parameter must be VARCHAR2. |

### SUBSTR

The SUBSTR function is used to retrieve a portion of a large object.

```
data { RAW | VARCHAR2 } SUBSTR(lob_loc { BLOB | CLOB }
  [, amount INTEGER [, offset INTEGER ]])
```

**Parameters**

| Parameter | Description |
|---|---|
| lob_loc | Specifies the location of the large object to be read. |
| amount | Specifies the number of bytes or characters to be returned. The default value is 32,767. |
| offset | Specifies the position in the large object to start reading. The first byte or character is labeled as offset 1. The default value is 1. |
| data | Specifies the retrieved portion of the large object. If the data type of the lob_loc parameter is BLOB, the data type of the buffer parameter must be RAW. If the data type of the lob_loc parameter is CLOB, the data type of the data parameter must be VARCHAR2. |

**TRIM**

The TRIM stored procedure is used to trim a large object to the specified length.

```
TRIM(lob_loc IN OUT { BLOB | CLOB }, newlen INTEGER)
```

**Parameters**

| Parameter | Description |
|---|---|
| lob_loc | Specifies the location of the large object to be trimmed. |
| newlen | Specifies the total number of bytes or characters in the large object to be trimmed. |

**WRITE**

The WRITE stored procedure is used to write data to a large object. Any data in the large object at the specified offset within the specified length is overwritten by data in the buffer.

```
WRITE(lob_loc IN OUT { BLOB | CLOB },
  amount BINARY_INTEGER,
  offset INTEGER, buffer { RAW | VARCHAR2 })
```

**Parameters**

| Parameter | Description |
|---|---|
| lob_loc | Specifies the location of a large object to be written. |
| amount | Specifies the number of bytes or characters in the buffer to be written to the large object . |
| offset | Specifies the position of the byte or character in the large object where the write operation starts. The value of the offset starts from 1. |
| buffer | Specifies the data in the buffer to be written to the large object. If the data type of the lob_loc parameter is BLOB, the data type of the buffer parameter must be RAW. If the data type of the lob_loc parameter is CLOB, the data type of the buffer parameter must be VARCHAR2. |

**WRITEAPPEND**

The WRITEAPPEND stored procedure is used to add data to the end of a large object.

```
WRITEAPPEND(lob_loc IN OUT { BLOB | CLOB },
  amount BINARY_INTEGER, buffer { RAW | VARCHAR2 })
```

**Parameters**

| Parameter | Description |
|---|---|
| lob_loc | Specifies the location of the large object to which data is added. |
| amount | Specifies the number of bytes or characters in the buffer to be added to the end of the large object. |
| buffer | Specifies the data to be added to the large object. If the data type of the lob_loc parameter is BLOB, the data type of the buffer parameter must be RAW. If the data type of the lob_loc parameter is CLOB, the data type of the buffer parameter must be VARCHAR2. |

## 17.7 DBMS_LOCK

POLARDB compatible with Oracle supports the DBMS_LOCK.SLEEP stored procedure.

**Table 17-6: DBMS_LOCK stored procedures**

| Function/stored procedure | Return type | Description |
|---|---|---|
| SLEEP(seconds) | N/A | Pauses a session for the specified number of seconds . |

The DBMS_LOCK package in POLARDB compatible with Oracle is only partially implemented when compared to Oracle's version. POLARDB compatible with Oracle only supports DBMS_LOCK.SLEEP.

**SLEEP**

The SLEEP stored procedure is used to pause the current session for the specified number of seconds.

```
SLEEP(seconds NUMBER)
```

**Parameters**

| Parameter | Description |
|---|---|
| seconds | Specifies the number of seconds for which the session is to be paused. You can specify a fractional value. For example, you can specify 1.75 to indicate one and three-fourths of a second. |

## 17.8 DBMS_MVIEW

You can use the stored procedures in the DBMS_MVIEW package to manage and update materialized views and their dependencies. POLARDB compatible with Oracle supports the following DBMS_MVIEW stored procedures:

**Table 17-7: DBMS_MVIEW stored procedures**

| Stored procedure | Return type | Description |
|---|---|---|
| GET_MV_DEPENDENCIES (list VARCHAR2, deplist VARCHAR2); | N/A | The GET_MV_DEPENDENCIES stored procedure can be used to retrieve a list of dependencies for a specified view. |
| REFRESH(list VARCHAR2, method VARCHAR2, rollback seg VARCHAR2 , push deferred rpc BOOLEAN, refresh after errors BOOLEAN , purge option NUMBER, parallelism NUMBER, heap size NUMBER , atomic refresh BOOLEAN , nested BOOLEAN ); | N/A | The variation of the REFRESH stored procedure can be used to update a list of views separated by commas (,). |
| REFRESH(tab dbms_utility.uncl_array, method VARCHAR2, rollback_seg VARCHAR2, push_deferred_rpc BOOLEAN, refresh_after_errors BOOLEAN, purge_option NUMBER , parallelism NUMBER , heap_size NUMBER, atomic_refresh BOOLEAN, nested BOOLEAN); | N/A | The variation of the REFRESH stored procedure can be used to update all views named in a table of dbms_utility.uncl_array values. |
| REFRESH_ALL_MVIEWS (number_of_failures BINARY_INTEGER, method VARCHAR2, rollback_seg VARCHAR2, refresh_after_errors BOOLEAN, atomic_refresh BOOLEAN); | N/A | The REFRESH_ALL_MVIEWS stored procedure can be used to update all materialized views. |

| Stored procedure | Return type | Description |
|---|---|---|
| REFRESH_DEPENDENT (number_of_failures BINARY_INTEGER, list VARCHAR2, method VARCHAR2, rollback_seg VARCHAR2, refresh_after_errors BOOLEAN, atomic_refresh BOOLEAN, nested BOOLEAN); | N/A | The variation of the REFRESH_DEPENDENT stored procedure can be used to update all views that are dependent on the views listed in a comma-separated list. |
| REFRESH_DEPENDENT (number_of_failures BINARY_INTEGER, tab dbms_utility.uncl_array, method VARCHAR2, rollback_seg VARCHAR2, refresh_after_errors BOOLEAN, atomic_refresh BOOLEAN, nested BOOLEAN); | N/A | The variation of the REFRESH_DEPENDENT stored procedure can be used to update all views that are dependent on the views listed in a table of dbms_utility.uncl_array values. |

The DBMS_MVIEW package in POLARDB compatible with Oracle is only partially implemented when compared to Oracle's version. POLARDB compatible with Oracle only supports the stored procedures that are listed in the preceding table.

### GET_MV_DEPENDENCIES

After a materialized view is named, you can use the GET_MV_DEPENDENCIES stored procedure to retrieve a list of items that are dependent on the specified view. The following code describes the syntax of the GET_MV_DEPENDENCIES stored procedure:

```
GET_MV_DEPENDENCIES(
  list IN VARCHAR2,
  deplist OUT VARCHAR2);
```

**Parameters**

| Parameter | Description |
|---|---|
| list | Specifies the name of a materialized view, or a list of materialized view names separated by commas (,). |

| Parameter | Description |
|---|---|
| deplist | Specifies a list of schema-qualified dependencies separated by commas (,).<br>**Note:**<br>The data type of the deplist parameter is VARCHAR2. |

**Examples**

```
DECLARE
  deplist VARCHAR2(1000);
BEGIN
  DBMS_MVIEW.GET_MV_DEPENDENCIES('public.emp_view', deplist);
  DBMS_OUTPUT.PUT_LINE('deplist: ' || deplist);
END;
```

In this example, a list of dependencies on the public.emp_view materialized view is retrieved.

**REFRESH**

You can use the REFRESH stored procedure to update a list of views separated by commas (,), or all views specified in a table of DBMS_UTILITY.UNCL_ARRAY values. The REFRESH stored procedure has two forms of syntax. When you specify a list of views separated by commas (,), you can use the first form of syntax:

```
REFRESH(
  list IN VARCHAR2,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL,
  push_deferred_rpc IN BOOLEAN DEFAULT TRUE,
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  purge_option IN NUMBER DEFAULT 1,
  parallelism IN NUMBER DEFAULT 0,
  heap_size IN NUMBER DEFAULT 0,
  atomic_refresh IN BOOLEAN DEFAULT TRUE,
  nested IN BOOLEAN DEFAULT FALSE);
```

The second form of syntax is used to specify views in a table of DBMS_UTILITY.UNCL_ARRAY values.

```
REFRESH(
  tab IN OUT DBMS_UTILITY.UNCL_ARRAY,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL,
  push_deferred_rpc IN BOOLEAN DEFAULT TRUE,
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  purge_option IN NUMBER DEFAULT 1,
  parallelism IN NUMBER DEFAULT 0,
  heap_size IN NUMBER DEFAULT 0,
  atomic_refresh IN BOOLEAN DEFAULT TRUE,
  nested IN BOOLEAN DEFAULT FALSE);
```

**Parameters**

| Parameter | Description |
|---|---|
| list | The data type of the list parameter is VARCHAR2. The parameter specifies the name of a materialized view, or a list of materialized view names separated by commas (,). The names must be schema-qualified. |
| tab | The parameter specifies the names of materialized views in a table of DBMS_UTILITY.UNCL_ARRAY values. |
| method | The data type of the method parameter is VARCHAR2. The parameter specifies the update method to be applied to the specified view. C is the only supported method, which is used to perform a complete update of the view. |
| rollback_seg | rollback_seg is used for compatibility and can be ignored. The default value is NULL. |
| push_deferred rpc | push_deferred_rpc is used for compatibility and can be ignored. The default value is TRUE. |
| refresh_after_errors | refresh_after_errors is used for compatibility and can be ignored. The default value is FALSE. |
| purge_option | purge_option is used for compatibility and can be ignored. The default value is 1. |
| parallelism | parallelism is used for compatibility and can be ignored. The default value is 0. |
| heap_size IN NUMBER DEFAULT 0, | heap_size is used for compatibility and can be ignored. The default value is 0. |
| atomic refresh | atomic_refresh is used for compatibility and can be ignored. The default value is TRUE. |
| nested | nested is used for compatibility and can be ignored. The default value is FALSE. |

**Examples**

The following example uses DBMS_MVIEW.REFRESH to update the materialized view named public.emp_view:

```
EXEC DBMS_MVIEW.REFRESH(list => 'public.emp_view', method => 'C');
```

### REFRESH_ALL_M VIEWS

You can use the REFRESH_ALL_MVIEWS stored procedure to update materialized views that are not updated after the table or view on which the views depend is updated. The following code describes the syntax of the REFRESH_ALL_MVIEWS stored procedure:

```
REFRESH_ALL_MVIEWS(
  number_of_failures OUT BINARY_INTEGER,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL,
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  atomic_refresh IN BOOLEAN DEFAULT TRUE);
```

**Parameters**

| Parameter | Description |
|---|---|
| number_of_failures | The data type of the number_of_failures parameter is BINARY_INTEGER. The parameter specifies the number of failures that occur during the update operation. |
| method | The data type of the method parameter is VARCHAR2. The parameter specifies the update method to be applied to the specified view. C is the only supported method, which is used to perform a complete update of the view. |
| rollback_seg | rollback_seg is used for compatibility and can be ignored. The default value is NULL. |
| refresh_after_errors | refresh_after_errors is used for compatibil ity and can be ignored. The default value is FALSE. |
| atomic refresh | atomic_refresh is used for compatibility and can be ignored. The default value is TRUE. |

**Examples**

```
DECLARE
  errors INTEGER;
BEGIN
  DBMS_MVIEW.REFRESH_ALL_MVIEWS(errors, method => 'C');
```

```
END;
```

After the update is complete, the errors variable contains the number of failures.

### REFRESH_DEPENDENT

You can use the REFRESH_DEPENDENT stored procedure to update all materialized views that are dependent on the views specified in the call to the stored procedure. You can specify a list of views separated by commas (,) or specify views in a table of DBMS_UTILITY.UNCL_ARRAY values.

The following syntax of the stored procedure is used to update all materialized views that are dependent on the views specified in a comma-separated list:

```
REFRESH_DEPENDENT(
  number_of_failures OUT BINARY_INTEGER,
  list IN VARCHAR2,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  atomic_refresh IN BOOLEAN DEFAULT TRUE,
  nested IN BOOLEAN DEFAULT FALSE);
```

The following syntax of the stored procedure is used to update all materialized views that are dependent on the views specified in a table of DBMS_UTILITY.UNCL_ARRAY values.

```
REFRESH_DEPENDENT(
  number_of_failures OUT BINARY_INTEGER,
  tab IN DBMS_UTILITY.UNCL_ARRAY,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL,
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  atomic_refresh IN BOOLEAN DEFAULT TRUE,
  nested IN BOOLEAN DEFAULT FALSE);
```

**Parameters**

| Parameter | Description |
|---|---|
| number_of_failures | The data type of the number_of_failures parameter is BINARY_INTEGER. The parameter specifies the number of failures that occur during the update operation. |
| list | The data type of the list parameter is VARCHAR2. The parameter specifies the name of materialized view, or a list of materialized view names separated by commas (,). The names must be schema-qualified. |

| Parameter | Description |
|---|---|
| tab | Specifies the names of materialized views in a table of DBMS_UTILITY.UNCL_ARRAY values. |
| method | The data type of the method parameter is VARCHAR2. The parameter specifies the update method to be applied to the specified view. C is the only supported method, which is used to perform a complete update of the view. |
| rollback_seg | rollback_seg is used for compatibility and can be ignored. The default value is NULL. |
| refresh_after_errors | refresh_after_errors is used for compatibil ity and can be ignored. The default value is FALSE. |
| atomic refresh | atomic_refresh is used for compatibility and can be ignored. The default value is TRUE. |
| nested | nested is used for compatibility and can be ignored. The default value is FALSE. |

**Examples**

The following example describes a complete update on all materialized views that depend on a materialized view named emp_view. emp_view resides in the public schema.

```
DECLARE
  errors INTEGER;
BEGIN
  DBMS_MVIEW.REFRESH_DEPENDENT (errors, list => 'public. emp_view ', method => 'C ');
END;
```

After the update is complete, the errors variable contains the number of failures.

## 17.9 DBMS_OUTPUT

The DBMS_OUTPUT package provides the capability to send messages (lines of text) to a message buffer, or to retrieve messages from the message buffer. A message buffer is local to a single session. You can use the DBMS_PIPE package to send messages between sessions.

The following table lists the functions and stored procedures that are available in the DBMS_OUTPUT package.

Table 17-8: DBMS_OUTPUT functions and stored procedures

| Function/stored procedure | Return type | Description |
|---|---|---|
| DISABLE | N/A | Disables the capability to send and receive messages. |
| ENABLE(buffer_size) | N/A | Enables the capability to send and receive messages. |
| GET LINE(line OUT, status OUT) | N/A | Retrieves a line from the message buffer. |
| GET LINES(lines OUT, numlines IN OUT) | N/A | Retrieves multiple lines from the message buffer. |
| NEW LINE | N/A | Puts an end-of-line character sequence. |
| PUT(item) | N/A | Puts a partial line without an end-of-line character sequence. |
| PUT LINE(item) | N/A | Puts a complete line with an end-of-line character sequence. |
| SERVEROUTPUT(stdout) | N/A | Directs messages from PUT , PUT LINE, or NEW_LINE to either standard output or the message buffer. |

The following table lists the public variable that is available in the DBMS_SQL package.

Table 17-9: DBMS_OUTPUT public variables

| Public variable | Data type | Value | Description |
|---|---|---|---|
| chararr | TABLE | | For message lines. |

### CHARARR

The CHARARR variable is used to store multiple message lines.

```
TYPE chararr IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```

### DISABLE

The DISABLE stored procedure clears out the message buffer. Any messages in the buffer at the time when the DISABLE stored procedure is called will no longer be accessible. Any messages subsequently sent with the PUT, PUT_LINE, or NEW_LINE stored procedure are discarded. When the PUT, PUT_LINE, or NEW_LINE stored procedure is called, no error is returned to the sender and the sending and receiving of messages are disabled.

You can use the ENABLE or SERVEROUTPUT(TRUE) stored procedure to re-enable the sending and receiving of messages.

```
DISABLE
```

**Examples**

The following anonymous block disables the sending and receiving of messages in the current session.

```
BEGIN
   DBMS_OUTPUT.DISABLE;
END;
```

**ENABLE**

The ENABLE stored procedure enables the capability to send messages to the message buffer or receive messages from the message buffer. Setting SERVEROUTPUT(TRUE) also performs an implicit call of the ENABLE stored procedure.

The status of the SERVEROUTPUT stored stored procedure depends the destination of a message sent with the PUT, PUT_LINE, or NEW_LINE procedure.

- If the last status of the SERVEROUTPUT stored procedure is TRUE, the message is sent to the standard output of the command line.
- If the last status of the SERVEROUTPUT stored procedure is FALSE, the message is sent to the message buffer.

```
ENABLE [ (buffer_size INTEGER) ]
```

**Parameters**

| Parameter | Description |
|---|---|
| buffer_size | The maximum length of the message buffer . Unit: byte. If the specified value of the buffer_size parameter is less than 2000, the buffer size is set to 2000. |

**Examples**

The following anonymous block enables the sending and receiving of messages. The SERVEROUTPUT(TRUE) stored procedure is configured to force messages to standard output .

```
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Messages enabled');
END;

Messages enabled
```

To achieve the same effect, you can also use only the SERVEROUTPUT(TRUE) stored procedure.

```
BEGIN
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Messages enabled');
END;

Messages enabled
```

The following anonymous block enables the sending and receiving of messages. However , the SERVEROUTPUT (FALSE) stored procedure is called to send messages to the message buffer.

```
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.SERVEROUTPUT(FALSE);
    DBMS_OUTPUT.PUT_LINE('Message sent to buffer');
END;
```

### GET_LINE

The GET_LINE stored procedure provides the capability to retrieve a line of text from the message buffer. Only text that has been terminated by an end-of-line character sequence is retrieved. The text is a complete line that is generated by using the PUT_LINE stored procedure, or by a series of PUT calls followed by a NEW_LINE call.

```
GET_LINE(line OUT VARCHAR2, status OUT INTEGER)
```

**Parameters**

| Parameter | Description |
|---|---|
| line | The variable used to receive the line of text from the message buffer. |

| Parameter | Description |
|---|---|
| status | If a line of text was returned from the message buffer, the value is 0. If no text was returned, the value is 1. |

**Examples**

The following anonymous block writes the emp table to the message buffer as a comma-delimited string for each row.

```
EXEC DBMS_OUTPUT.SERVEROUTPUT(FALSE);

DECLARE
    v_emprec        VARCHAR2(120);
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    DBMS_OUTPUT.ENABLE;
    FOR i IN emp_cur LOOP
        v_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||
            NVL(LTRIM(TO_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||
            ',' || i.sal || ',' ||
            NVL(LTRIM(TO_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;
        DBMS_OUTPUT.PUT_LINE(v_emprec);
    END LOOP;
END;
```

The following anonymous block reads the message buffer and inserts the messages written by the preceding example into a table named messages. The rows in the message table are then displayed.

```
CREATE TABLE messages (
    status          INTEGER,
    msg             VARCHAR2(100)
);

DECLARE
    v_line          VARCHAR2(100);
    v_status          INTEGER := 0;
BEGIN
    DBMS_OUTPUT.GET_LINE(v_line,v_status);
    WHILE v_status = 0 LOOP
        INSERT INTO messages VALUES(v_status, v_line);
        DBMS_OUTPUT.GET_LINE(v_line,v_status);
    END LOOP;
END;

SELECT msg FROM messages;

                              msg
-----------------------------------------------------------------
 7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
 7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30
 7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30
 7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20
 7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
 7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
 7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10
 7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20
 7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10
 7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30
 7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
 7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
 7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
 7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
(14 rows)
```

### GET_LINES

The GET_LINES stored procedure provides the capability to retrieve one or more lines of text from the message buffer into a collection. Only text that has been terminated by an end-of-line character sequence is retrieved. The text is a complete line that is generated by using the PUT_LINE stored procedure, or by a series of PUT calls followed by a NEW_LINE call.

```
GET_LINES(lines OUT CHARARR, numlines IN OUT INTEGER)
```

**Parameters**

| Parameter | Description |
|---|---|
| lines | The table that receives the lines of text from the message buffer. For more information about the lines parameter, see CHARARR. |
| numlines IN | The number of lines to be retrieved from the message buffer. |
| numlines OUT | The number of lines retrieved from the message buffer. If the output value of the numlines parameter is less than the input value, then the message buffer contains no more lines. |

**Examples**

The following example uses the GET_LINES stored procedure to store all rows from the emp table that were placed on the message buffer, into an array.

```
EXEC DBMS_OUTPUT.SERVEROUTPUT(FALSE);

DECLARE
    v_emprec        VARCHAR2(120);
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    DBMS_OUTPUT.ENABLE;
    FOR i IN emp_cur LOOP
        v_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||
            NVL(LTRIM(TO_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||
            ',' || i.sal || ',' ||
            NVL(LTRIM(TO_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;
        DBMS_OUTPUT.PUT_LINE(v_emprec);
    END LOOP;
END;

DECLARE
    v_lines         DBMS_OUTPUT.CHARARR;
    v_numlines      INTEGER := 14;
    v_status        INTEGER := 0;
BEGIN
    DBMS_OUTPUT.GET_LINES(v_lines,v_numlines);
    FOR i IN 1..v_numlines LOOP
        INSERT INTO messages VALUES(v_numlines, v_lines(i));
    END LOOP;
END;

SELECT msg FROM messages;

                              msg
-----------------------------------------------------------------
 7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
 7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30
 7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30
 7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20
 7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
 7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
 7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10
 7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20
 7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10
 7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30
 7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
 7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
 7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
 7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
(14 rows)
```

### NEW LINE

The NEW_LINE stored procedure writes an end-of-line character sequence in the message buffer.

```
NEW_LINE
```

**Parameters**

The NEW_LINE stored procedure requires no parameters.

### PUT

The PUT stored procedure writes a string to the message buffer. No end-of-line character sequence is written at the end of the string. You can use the NEW_LINE stored procedure to add an end-of-line character sequence.

```
PUT(item VARCHAR2)
```

**Parameters**

| Parameter | Description |
|---|---|
| item | The text written to the message buffer. |

**Examples**

The following example uses the PUT stored procedure to display a comma-delimited list of employees from the emp table.

```
DECLARE
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    FOR i IN emp_cur LOOP
        DBMS_OUTPUT.PUT(i.empno);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.ename);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.job);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.mgr);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.hiredate);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.sal);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.comm);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.deptno);
        DBMS_OUTPUT.NEW_LINE;
    END LOOP;
END;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30
7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30
7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10
7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20
7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10
7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
```

## PUT_LINE

The PUT_LINE stored procedure writes a single line to the message buffer including an end-of-line character sequence.

```
PUT_LINE(item VARCHAR2)
```

**Parameters**

| Parameter | Description |
|---|---|
| item | The text to be written to the message buffer. |

**Examples**

The following example uses the PUT_LINE stored procedure to display a comma-delimited list of employees from the emp table.

```
DECLARE
    v_emprec        VARCHAR2(120);
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    FOR i IN emp_cur LOOP
        v_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||
            NVL(LTRIM(TO_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||
            ',' || i.sal || ',' ||
            NVL(LTRIM(TO_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;
        DBMS_OUTPUT.PUT_LINE(v_emprec);
    END LOOP;
END;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30
7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30
7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10
7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20
7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10
7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
```

### SERVEROUTPUT

The SERVEROUTPUT stored procedure provides the capability to direct messages to the standard output of the command line or to the message buffer. Setting SERVEROUTPUT( TRUE) also performs an implicit call of the ENABLE stored procedure.

The default setting of the SERVEROUTPUT stored procedure is implementation dependent . For example, in Oracle SQL*Plus, the default setting is SERVEROUTPUT(FALSE). In psql, the default setting is SERVEROUTPUT (TRUE). Note that in Oracle SQL*Plus, this setting is controlled by using the SQL*Plus SET command, not by a stored procedure as implemented in POLARDB compatible with Oracle.

```
SERVEROUTPUT(stdout BOOLEAN)
```

**Parameters**

| Parameter | Description |
|---|---|
| stdout | To ensure that subsequent PUT, PUT_LINE, or NEW_LINE commands send text to the standard output of the command line, you need to set this parameter to TRUE. To send text to the message buffer, you need to set this parameter to FALSE. |

**Examples**

The following anonymous block sends the first message to the command line and the second message to the message buffer.

```
BEGIN
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('This message goes to the command line');
    DBMS_OUTPUT.SERVEROUTPUT(FALSE);
    DBMS_OUTPUT.PUT_LINE('This message goes to the message buffer');
END;

This message goes to the command line
```

If the following anonymous block is executed within the same session, the message stored in the message buffer from the preceding example is flushed. This message is displayed on the command line as a new message.

```
BEGIN
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Flush messages from the buffer');
END;

This message goes to the message buffer
Flush messages from the buffer
```

## 17.10 DBMS_PIPE

The DBMS_PIPE package provides the capability to send messages through a pipe within or between sessions connected to the same database cluster.

The following table lists the functions and stored procedures that are available in the DBMS_PIPE package.

**Table 17-10: DBMS_PIPE functions and stored procedures**

| Function/stored procedure | Return type | Description |
|---|---|---|
| CREATE PIPE(pipename [, maxpipesize ] [, private ]) | INTEGER | Explicitly creates a private pipe if private is "true" (the default) or a public pipe if private is "false". |
| NEXT ITEM TYPE | INTEGER | Determines the data type of the next item in a received message. |
| PACK MESSAGE(item) | N/A | Places item in the local message buffer of the current session. |
| PURGE(pipename) | N/A | Removes unreceived messages from the specified pipe. |
| RECEIVE MESSAGE(pipename [, timeout ]) | INTEGER | Retrieves a message from a specified pipe. |
| REMOVE PIPE(pipename) | INTEGER | Deletes an explicitly created pipe. |
| RESET BUFFER | N/A | Resets the local message buffer. |
| SEND MESSAGE(pipename [, timeout ] [, maxpipesize ]) | INTEGER | Sends a message on a pipe. |
| UNIQUE SESSION NAME | VARCHAR2 | Obtains a unique session name. |
| UNPACK MESSAGE(item OUT) | N/A | Retrieves the next data item from a message into a type-compatible variable, item. |

Pipes are categorized as implicit or explicit. An implicit pipe is created if a reference is made to a pipe name that was not previously created by the CREATE_PIPE function. For example, if the SEND_MESSAGE function is executed using a non-existent pipe name, a new implicit pipe is created with that name. An explicit pipe is created using the CREATE_PIPE function with the first parameter specified. The first parameter specifies the pipe name for the new pipe.

Pipes are also categorized as private or public. A private pipe can only be accessed by the user who created the pipe. Even a superuser cannot access a private pipe that was created by another user. A public pipe can be accessed by any user who has access to the DBMS_PIPE package.

A public pipe can only be created by using the CREATE_PIPE function with the third parameter set to FALSE. The CREATE_PIPE function can be used to create a private pipe by setting the third parameter to TRUE or by omitting the third parameter. All implicit pipes are private.

The individual data items or message lines are first built in a local message buffer, unique to the current session. The PACK_MESSAGE stored procedure builds the message in the local message buffer of the current session. The SEND_MESSAGE function is then used to send the message through the pipe.

The receiving of a message involves the reverse operation. The RECEIVE_MESSAGE function is used to retrieve a message from the specified pipe. The message is written to the local message buffer of the current session. The UNPACK_MESSAGE stored procedure is then used to transfer the message data items from the message buffer to program variables. If a pipe contains multiple messages, RECEIVE_MESSAGE retrieves the messages in first-in-first-out ( FIFO) order.

Each session maintains separate message buffers for messages created with the PACK_MESSAGE stored procedure and messages retrieved by the RECEIVE_MESSAGE function. The messages can be both built and received in the same session. However, if consecutive RECEIVE_MESSAGE calls are made, only the message from the last RECEIVE_ME SSAGE call will be preserved in the local message buffer.

## CREATE_PIPE

The CREATE_PIPE function creates an explicit public pipe or an explicit private pipe with a specified name.

```
status INTEGER CREATE_PIPE(pipename VARCHAR2
  [, maxpipesize INTEGER ] [, private BOOLEAN ])
```

### Parameters

| Parameter | Description |
|---|---|
| pipename | The name of the pipe. |
| maxpipesize | The maximum capacity of the pipe. Unit: byte. Default value: 8192. |

| Parameter | Description |
|---|---|
| private | To create a public pipe, you need to set this parameter to FALSE. To create a private pipe , you need to set this parameter to TRUE. Default value: TRUE. |
| status | The status code returned by the operation. 0 indicates successful creation. |

### Examples

The following example creates a private pipe named messages:

```
DECLARE
    v_status       INTEGER;
BEGIN
    v_status := DBMS_PIPE.CREATE_PIPE('messages');
    DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END;
CREATE_PIPE status: 0
```

The following example creates a public pipeline named mailbox:

```
DECLARE
    v_status       INTEGER;
BEGIN
    v_status := DBMS_PIPE.CREATE_PIPE('mailbox',8192,FALSE);
    DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END;
CREATE_PIPE status: 0
```

## NEXT_ITEM_TYPE

The NEXT_ITEM_TYPE function returns an integer code identifying the data type of the next data item in a message that has been retrieved into the local message buffer of the current session. If an item is removed from the local message buffer by the UNPACK_MESSAGE stored procedure, the NEXT_ITEM_TYPE function returns the data type code for the next available item. If no more items exist in the message, the code 0 is returned.

```
typecode INTEGER NEXT_ITEM_TYPE
```

### Parameters

| Parameter | Description |
|---|---|
| typecode | A code that identifies the data type of the next data item. Table 17-11: Data type codes of NEXT_ITEM_TYPE lists the code of each data type. |

**Table 17-11: Data type codes of NEXT_ITEM_TYPE**

| Type code | Data type |
|---|---|
| 0 | No more data items |
| 9 | NUMBER |
| 11 | VARCHAR2 |
| 13 | DATE |
| 23 | RAW |

**Note:**

The type codes listed in the table are not compatible with Oracle databases. Oracle assigns a different numbering sequence to the data types.

The following example shows a pipe packed with a NUMBER item, a VARCHAR2 item, a DATE item, and a RAW item. A second anonymous block then uses the NEXT_ITEM_TYPE function to display the type code of each item.

```
DECLARE
    v_number        NUMBER := 123;
    v_varchar       VARCHAR2(20) := 'Character data';
    v_date          DATE := SYSDATE;
    v_raw           RAW(4) := '21222324';
    v_status        INTEGER;
BEGIN
    DBMS_PIPE.PACK_MESSAGE(v_number);
    DBMS_PIPE.PACK_MESSAGE(v_varchar);
    DBMS_PIPE.PACK_MESSAGE(v_date);
    DBMS_PIPE.PACK_MESSAGE(v_raw);
    v_status := DBMS_PIPE.SEND_MESSAGE('datatypes');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' ‖ v_status);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' ‖ SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' ‖ SQLCODE);
END;

SEND_MESSAGE status: 0

DECLARE
    v_number        NUMBER;
    v_varchar       VARCHAR2(20);
    v_date          DATE;
    v_timestamp     TIMESTAMP;
    v_raw           RAW(4);
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('datatypes');
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' ‖ v_status);
    DBMS_OUTPUT.PUT_LINE('----------------------------------');

    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
    DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_number);
    DBMS_OUTPUT.PUT_LINE('NUMBER Item   : ' || v_number);
    DBMS_OUTPUT.PUT_LINE('---------------------------------');
    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
    DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_varchar);
    DBMS_OUTPUT.PUT_LINE('VARCHAR2 Item : ' || v_varchar);
    DBMS_OUTPUT.PUT_LINE('---------------------------------');

    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
    DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_date);
    DBMS_OUTPUT.PUT_LINE('DATE Item     : ' || v_date);
    DBMS_OUTPUT.PUT_LINE('---------------------------------');

    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
    DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_raw);
    DBMS_OUTPUT.PUT_LINE('RAW Item      : ' || v_raw);
    DBMS_OUTPUT.PUT_LINE('---------------------------------');

    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
    DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
    DBMS_OUTPUT.PUT_LINE('---------------------------------');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

RECEIVE_MESSAGE status: 0
---------------------------------
NEXT_ITEM_TYPE: 9
NUMBER Item   : 123
---------------------------------
NEXT_ITEM_TYPE: 11
VARCHAR2 Item : Character data
---------------------------------
NEXT_ITEM_TYPE: 13
DATE Item     : 02-OCT-07 11:11:43
---------------------------------
NEXT_ITEM_TYPE: 23
RAW Item      : 21222324
---------------------------------
NEXT_ITEM_TYPE: 0
```

### PACK_MESSAGE

The PACK_MESSAGE stored procedure places an item of data in the local message buffer of the current session. You must call the PACK_MESSAGE stored procedure at least once before issuing a SEND_MESSAGE call.

```
PACK_MESSAGE(item { DATE | NUMBER | VARCHAR2 | RAW })
```

After you retrieve the message by issuing a RECEIVE_MESSAGE call, you can use the UNPACK_MESSAGE stored procedure to obtain data items.

#### Parameters

| Parameter | Description |
|---|---|
| item | An expression that is used to calculate the acceptable parameter data types. The calculated value is added to the local message buffer of the session. |

### PURGE

The PURGE stored procedure removes unreceived messages from a specified implicit pipe.

```
PURGE(pipename VARCHAR2)
```

You can use the REMOVE_PIPE function to delete an explicit pipe.

#### Parameters

| Parameter | Description |
|---|---|
| pipename | The name of the pipe. |

#### Examples

Two messages are sent on a pipe:

```
DECLARE
    v_status       INTEGER;
BEGIN
    DBMS_PIPE.PACK_MESSAGE('Message #1');
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);

    DBMS_PIPE.PACK_MESSAGE('Message #2');
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END;

SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

Receive the first message and unpack it:

```
DECLARE
    v_item         VARCHAR2(80);
    v_status       INTEGER;
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_item);
    DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END;

RECEIVE_MESSAGE status: 0
```

```
Item: Message #1
```

Purge the pipe:

```
EXEC DBMS_PIPE.PURGE('pipe');
```

The following code example shows an attempt to retrieve the next message. The RECEIVE_MESSAGE call returns status code 1, which indicates that a timeout occurs because no message is available.

```
DECLARE
    v_item          VARCHAR2(80);
    v_status         INTEGER;
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
END;

RECEIVE_MESSAGE status: 1
```

**RECEIVE_MESSAGE**

The RECEIVE_MESSAGE function retrieves a message from a specified pipe.

```
status INTEGER RECEIVE_MESSAGE(pipename VARCHAR2
  [, timeout INTEGER ])
```

**Parameters**

pipename

The name of the pipe.

timeout

The timeout period. Unit: second. Default value: 86400000 (1000 days).

Status

The status code returned by the operation.

The following table lists possible status codes.

| Status code | Description |
|---|---|
| 0 | The operation is successful. |
| 1 | A timeout occurs. |
| 2 | The message is too large for the buffer. |

### REMOVE_PIPE

The REMOVE_PIPE function deletes an explicit private pipe or explicit public pipe.

```
status INTEGER REMOVE_PIPE(pipename VARCHAR2)
```

You can use the REMOVE_PIPE function to delete an explicit pipe, such as a pipe created by using the CREATE_PIPE function.

**Parameters**

| Parameter | Description |
|---|---|
| pipename | The name of the pipe. |
| status | The status code returned by the operation . A status code of 0 is returned even if the specified pipe does not exist. |

**Examples**

Two messages are sent on a pipe:

```
DECLARE
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.CREATE_PIPE('pipe');
    DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status : ' ‖ v_status);

    DBMS_PIPE.PACK_MESSAGE('Message #1');
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' ‖ v_status);

    DBMS_PIPE.PACK_MESSAGE('Message #2');
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' ‖ v_status);
END;

CREATE_PIPE status : 0
SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

Receive the first message and unpack it:

```
DECLARE
    v_item          VARCHAR2(80);
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' ‖ v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_item);
    DBMS_OUTPUT.PUT_LINE('Item: ' ‖ v_item);
END;

RECEIVE_MESSAGE status: 0
```

```
Item: Message #1
```

Remove the pipe:

```
SELECT DBMS_PIPE.REMOVE_PIPE('pipe') FROM DUAL;

remove_pipe
-------------
          0
(1 row)
```

The following code example shows an attempt to retrieve the next message. The RECEIVE_MESSAGE call returns status code 1, which indicates that a timeout occurs because the pipe has been deleted.

```
DECLARE
    v_item          VARCHAR2(80);
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
END;

RECEIVE_MESSAGE status: 1
```

**RESET_BUFFER**

The RESET_BUFFER stored procedure resets a pointer to the local message buffer back to the beginning of the buffer. This causes subsequent PACK_MESSAGE calls to overwrite any data items that existed in the message buffer prior to the RESET_BUFFER call.

```
RESET_BUFFER
```

**Examples**

A message to John is written to the local message buffer. You can call the RESET_BUFFER stored procedure to replace this message with a message to Bob. The message to Bob is sent on the pipe.

```
DECLARE
    v_status        INTEGER;
BEGIN
    DBMS_PIPE.PACK_MESSAGE('Hi, John');
    DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 3:00, today?') ;
    DBMS_PIPE.PACK_MESSAGE('If not, is tomorrow at 8:30 ok with you?') ;
    DBMS_PIPE.RESET_BUFFER;
    DBMS_PIPE.PACK_MESSAGE('Hi, Bob');
    DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 9:30, tomorrow?') ;
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END;
```

```
SEND_MESSAGE status: 0
```

The message to Bob is displayed in the received message.

```
DECLARE
    v_item          VARCHAR2(80);
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
    DBMS_PIPE.UNPACK_MESSAGE(v_item);
    DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
    DBMS_PIPE.UNPACK_MESSAGE(v_item);
    DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END;

RECEIVE_MESSAGE status: 0
Item: Hi, Bob
Item: Can you attend a meeting at 9:30, tomorrow?
```

### SEND_MESSAGE

The SEND_MESSAGE function sends a message from the local message buffer to the specified pipe.

```
status SEND_MESSAGE(pipename VARCHAR2 [, timeout INTEGER ]
  [, maxpipesize INTEGER ])
```

**Parameters**

| Parameter | Description |
|---|---|
| pipename | The name of the pipe. |
| timeout | The timeout period. Unit: second. Default value: 86400000 (1000 days). |
| maxpipesize | The maximum capacity of the pipe. Unit: byte. Default value: 8192 bytes. |
| Status | The status code returned by the operation. |

The following table lists possible status codes.

**Table 17-12: Status codes of SEND_MESSAGE**

| Status code | Description |
|---|---|
| 0 | The operation is successful. |
| 1 | A timeout occurs. |
| 3 | The function is interrupted. |

### UNIQUE_SESSION_NAME

The UNIQUE_SESSION_NAME function returns a name that is unique to the current session.

```
name VARCHAR2 UNIQUE_SESSION_NAME
```

#### Parameters

| Parameter | Description |
|---|---|
| name | A unique session name. |

#### Examples

The following anonymous block retrieves and displays a unique session name.

```
DECLARE
    v_session       VARCHAR2(30);
BEGIN
    v_session := DBMS_PIPE.UNIQUE_SESSION_NAME;
    DBMS_OUTPUT.PUT_LINE('Session Name: ' || v_session);
END;

Session Name: PG$PIPE$5$2752
```

### UNPACK_MESSAGE

The UNPACK_MESSAGE stored procedure copies the data items of a message from the local message buffer to a specified program variable. Before you use the UNPACK_MESSAGE stored procedure, you must place the message in the local message buffer by using the RECEIVE_MESSAGE function.

```
UNPACK_MESSAGE(item OUT { DATE | NUMBER | VARCHAR2 | RAW })
```

#### Parameters

| Parameter | Description |
|---|---|
| item | A variable that receives a data item from the local message buffer. This variable must be compatible with the type of the data item. |

### Comprehensive example

The following example uses a pipe as a "mailbox". A series of stored procedures are used to create the mailbox, to add a multi-item message to the mailbox (up to three items), and to display the full contents of the mailbox. These stored procedures are enclosed in a package named mailbox.

```
CREATE OR REPLACE PACKAGE mailbox
IS
    PROCEDURE create_mailbox;
    PROCEDURE add_message (
        p_mailbox   VARCHAR2,
        p_item_1    VARCHAR2,
        p_item_2    VARCHAR2 DEFAULT 'END',
        p_item_3    VARCHAR2 DEFAULT 'END'
    );
    PROCEDURE empty_mailbox (
        p_mailbox   VARCHAR2,
        p_waittime  INTEGER DEFAULT 10
    );
END mailbox;

CREATE OR REPLACE PACKAGE BODY mailbox
IS
    PROCEDURE create_mailbox
    IS
        v_mailbox   VARCHAR2(30);
        v_status    INTEGER;
    BEGIN
        v_mailbox := DBMS_PIPE.UNIQUE_SESSION_NAME;
        v_status := DBMS_PIPE.CREATE_PIPE(v_mailbox,1000,FALSE);
        IF v_status = 0 THEN
            DBMS_OUTPUT.PUT_LINE('Created mailbox: ' || v_mailbox);
        ELSE
            DBMS_OUTPUT.PUT_LINE('CREATE_PIPE failed - status: ' ||
                v_status);
        END IF;
    END create_mailbox;

    PROCEDURE add_message (
        p_mailbox   VARCHAR2,
        p_item_1    VARCHAR2,
        p_item_2    VARCHAR2 DEFAULT 'END',
        p_item_3    VARCHAR2 DEFAULT 'END'
    )
    IS
        v_item_cnt  INTEGER := 0;
        v_status    INTEGER;
    BEGIN
        DBMS_PIPE.PACK_MESSAGE(p_item_1);
        v_item_cnt := 1;
        IF p_item_2 ! = 'END' THEN
            DBMS_PIPE.PACK_MESSAGE(p_item_2);
            v_item_cnt := v_item_cnt + 1;
        END IF;
        IF p_item_3 ! = 'END' THEN
            DBMS_PIPE.PACK_MESSAGE(p_item_3);
            v_item_cnt := v_item_cnt + 1;
        END IF;
        v_status := DBMS_PIPE.SEND_MESSAGE(p_mailbox);
        IF v_status = 0 THEN
            DBMS_OUTPUT.PUT_LINE('Added message with ' || v_item_cnt ||
                ' item(s) to mailbox ' || p_mailbox);
        ELSE
            DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE in add_message failed - ' ||
                'status: ' || v_status);
```

```
      END IF;
   END add_message;

   PROCEDURE empty_mailbox (
      p_mailbox   VARCHAR2,
      p_waittime  INTEGER DEFAULT 10
   )
   IS
      v_msgno     INTEGER DEFAULT 0;
      v_itemno    INTEGER DEFAULT 0;
      v_item      VARCHAR2(100);
      v_status    INTEGER;
   BEGIN
      v_status := DBMS_PIPE.RECEIVE_MESSAGE(p_mailbox,p_waittime);
      WHILE v_status = 0 LOOP
         v_msgno := v_msgno + 1;
         DBMS_OUTPUT.PUT_LINE('****** Start message #' || v_msgno ||
            ' ******');
         BEGIN
            LOOP
               v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
               EXIT WHEN v_status = 0;
               DBMS_PIPE.UNPACK_MESSAGE(v_item);
               v_itemno := v_itemno + 1;
               DBMS_OUTPUT.PUT_LINE('Item #' || v_itemno || ': ' ||
                  v_item);
            END LOOP;
            DBMS_OUTPUT.PUT_LINE('******* End message #' || v_msgno ||
               ' *******');
            DBMS_OUTPUT.PUT_LINE('*');
            v_itemno := 0;
            v_status := DBMS_PIPE.RECEIVE_MESSAGE(p_mailbox,1);
         END;
      END LOOP;
      DBMS_OUTPUT.PUT_LINE('Number of messages received: ' || v_msgno);
      v_status := DBMS_PIPE.REMOVE_PIPE(p_mailbox);
      IF v_status = 0 THEN
         DBMS_OUTPUT.PUT_LINE('Deleted mailbox ' || p_mailbox);
      ELSE
         DBMS_OUTPUT.PUT_LINE('Could not delete mailbox - status: '
            || v_status);
      END IF;
   END empty_mailbox;
END mailbox;
```

The following example demonstrates the execution of the stored procedures in mailbox. The first stored procedure creates a public pipe with a name generated by the UNIQUE_SESSION_NAME function.

```
EXEC mailbox.create_mailbox;

Created mailbox: PG$PIPE$13$3940
```

By using the mailbox name, a user in the same database with access to the mailbox and DBMS_PIPE packages can add messages.

```
EXEC mailbox.add_message('PG$PIPE$13$3940','Hi, John','Can you attend a meeting at 3:
00, today?','-- Mary');
```

```
Added message with 3 item(s) to mailbox PG$PIPE$13$3940

EXEC mailbox.add_message('PG$PIPE$13$3940','Don''t forget to submit your report','
Thanks,','-- Joe');

Added message with 3 item(s) to mailbox PG$PIPE$13$3940
```

The contents of the mailbox can be emptied.

```
EXEC mailbox.empty_mailbox('PG$PIPE$13$3940');

****** Start message #1 ******
Item #1: Hi, John
Item #2: Can you attend a meeting at 3:00, today?
Item #3: -- Mary
******* End message #1 *******
*
****** Start message #2 ******
Item #1: Don't forget to submit your report
Item #2: Thanks,
Item #3: Joe
******* End message #2 *******
*
Number of messages received: 2
Deleted mailbox PG$PIPE$13$3940
```

## 17.11 DBMS_PROFILER

The DBMS_PROFILER package collects and stores the performance information about PL /pgSQL and SPL statements that are executed during a performance profiling session. The following table lists functions and stored procedures that can be used to control the profiling tool.

**Table 17-13: DBMS_PROFILER functions and stored procedures**

| Function/stored procedure | Function or stored procedure | Return type | Description |
|---|---|---|---|
| FLUSH DATA | Both | Status code or exception | Flushes performance data collected in the current session without terminating the session (profiling continues). |
| GET VERSION(major OUT, minor OUT) | Procedure | N/A | Returns the version number of this package. |

| Function/stored procedure | Function or stored procedure | Return type | Description |
|---|---|---|---|
| INTERNAL VERSION CHECK | Function | Status code | Confirms that the current version of the profiler will work with the current database. |
| PAUSE PROFILER | Both | Status code or exception | Pauses data collection. |
| PAUSE_PROFILER | Both | Status code or exception | Resumes data collection. |
| START PROFILER (run_comment, run_comment1 [, run_number OUT ]) | Both | Status code or exception | Starts data collection . |
| STOP PROFILER | Both | Status code or exception | Stops data collection and flush performance data to the PLSQL PROFILER RAWDATA table. |

The functions within the DBMS_PROFILER package return a status code to indicate success or failure. The stored procedures within the DBMS_PROFILER package raise an exception only if they encounter a failure. The following table lists the status codes and messages returned by the functions, and the exceptions raised by the stored procedures.

**Table 17-14: DBMS_PROFILER status codes and exceptions**

| Status code | Message | Exception | Description |
|---|---|---|---|
| -1 | error version | version_mismatch | The profiler version and the database are incompatible. |
| 0 | success | N/A | The operation is successful. |
| 1 | error_param | profiler_error | The operation received an incorrect parameter. |
| 2 | error_io | profiler_error | The data flush operation has failed. |

### FLUSH_DATA

The FLUSH_DATA function or stored procedure flushes the data collected in the current session without terminating the profiler session. The data is flushed to the tables described in the POLARDB compatible with Oracle Performance Features Guide. The syntax for FLUSH_DATA functions and stored procedures is as follows:

```
status INTEGER FLUSH_DATA
FLUSH_DATA
```

**Parameters**

**Table 17-15:**

| Parameter | Description |
|---|---|
| status | The status code returned by the operation. |

### GET_VERSION

The GET_VERSION stored procedure returns the version of the DBMS_PROFILER package. Syntax:

```
GET_VERSION(major OUT INTEGER, minor OUT INTEGER)
```

**Parameters**

**Table 17-16:**

| Parameter | Description |
|---|---|
| major | The major version number of the DBMS_PROFILER package. |
| minor | The minor version number of the DBMS_PROFILER package. |

### INTERNAL_VERSION_CHECK

The INTERNAL_VERSION_CHECK function confirms that the current version of the DBMS_PROFILER package will work with the current database. The syntax of the INTERNAL_VERSION_CHECK function is as follows:

```
status INTEGER INTERNAL_VERSION_CHECK
```

**Parameters**

**Table 17-17:**

| Parameter | Description |
|---|---|
| status | The status code returned by the operation. |

### PAUSE_PROFILER

The PAUSE_PROFILER function or stored procedure pauses a profiling session. Syntax:

```
status INTEGER PAUSE_PROFILER

PAUSE_PROFILER
```

**Parameters**

**Table 17-18:**

| Parameter | Description |
|---|---|
| status | The status code returned by the operation. |

### RESUME_PROFILER

The RESUME_PROFILER function or stored procedure resumes a profiling session. The syntax of the RESUME_PROFILER function or stored procedure is as follows:

```
status INTEGER RESUME_PROFILER

RESUME_PROFILER
```

**Parameters**

**Table 17-19:**

| Parameter | Description |
|---|---|
| status | The status code returned by the operation. |

### START_PROFILER

The START_PROFILER function or stored procedure starts a data collection session. Syntax:

```
status INTEGER START_PROFILER(run_comment TEXT := SYSDATE,
  run_comment1 TEXT := '' [, run_number OUT INTEGER ])

START_PROFILER(run_comment TEXT := SYSDATE,
  run_comment1 TEXT := '' [, run_number OUT INTEGER ])
```

**Parameters**

**Table 17-20:**

| Parameter | Description |
|---|---|
| run_comment | A user-defined comment for the profiler session. The default value is SYSDATE. |
| run_comment1 | An additional user-defined comment for the profiler session. The default value is ''. |
| run_number | The session number of the profiler session. |
| status | The status code returned by the operation. |

### STOP_PROFILER

The STOP_PROFILER function or stored procedure stops a profiling session and flushes the performance information to the DBMS_PROFILER tables and views. Syntax:

```
status INTEGER STOP_PROFILER

STOP_PROFILER
```

**Parameters**

**Table 17-21:**

| Parameter | Description |
|---|---|
| status | The status code returned by the operation. |

## 17.12 DBMS_RANDOM

The DBMS_RANDOM package provides a number of methods to generate random values. The following table lists the functions and stored procedures that are available in the DBMS_RANDOM package.

**Table 17-22: DBMS_RANDOM functions and stored procedures**

| Function/stored procedure | Return type | Description |
|---|---|---|
| INITIALIZE(val) | N/A | Initializes the DBMS_RANDOM package with the specified seed value. Deprecated, but supported for backward compatibility. |
| NORMAL() | NUMBER | Returns a random NUMBER. |

| Function/stored procedure | Return type | Description |
|---|---|---|
| RANDOM | INTEGER | Returns a random INTEGER , which is greater than or equal to -2A31 and less than 2A31. Deprecated, but supported for backward compatibility. |
| SEED(val) | N/A | Resets the seed with the specified value. |
| SEED(val) | N/A | Resets the seed with the specified value. |
| STRING(opt, len) | VARCHAR2 | Returns a random string. |
| TERMINATE | N/A | Has no effect. Deprecated, but supported for backward compatibility. |
| VALUE | NUMBER | Returns a random number with a value greater than or equal to 0 and less than 1, with 38 digit precision. |
| VALUE(low, high) | NUMBER | Returns a random number with a value greater than or equal to low and less than high. |

### INITIALIZE

The INITIALIZE stored procedure uses a seed value to initialize the DBMS_RANDOM package.

Syntax:

```
INITIALIZE(val IN INTEGER)
```

The INITIALIZE stored procedure can be considered deprecated because it is only included for backward compatibility.

#### Parameters

| Parameter | Description |
|---|---|
| val | The seed value used by the DBMS_RANDOM package algorithm. |

#### Examples

The following code snippet demonstrates a call to the INITIALIZE stored procedure that initializes the DBMS_RANDOM package with the seed value, 6475.

```
DBMS_RANDOM.INITIALIZE(6475);
```

### NORMAL

The NORMAL function returns a random number of type NUMBER. Syntax:

```
result NUMBER NORMAL()
```

**Parameters**

**Table 17-23:**

| Parameter | Description |
|---|---|
| result | A random value of type NUMBER. |

**Examples**

The following code snippet demonstrates a call to the NORMAL function:

```
x:= DBMS_RANDOM.NORMAL();
```

### RANDOM

The RANDOM function returns a random INTEGER value that is greater than or equal to -2 ^ 31 and less than 2 ^31. Syntax:

```
result INTEGER RANDOM()
```

The RANDOM function can be considered deprecated because it is only included for backward compatibility.

**Parameters**

**Table 17-24:**

| Parameter | Description |
|---|---|
| result | A random value of type INTEGER. |

**Examples**

The following code snippet demonstrates a call to the RANDOM function. The call returns a random number:

```
x := DBMS_RANDOM.RANDOM();
```

## SEED

The SEED stored procedure resets the seed value for the DBMS_RANDOM package by using a string value. Syntax:

```
SEED(val IN VARCHAR2)
```

### Parameters

| Parameter | Description |
|---|---|
| val | The val parameter is the seed value used by the DBMS_RANDOM package algorithm. |

### Examples

The following code snippet demonstrates a call to the SEED stored procedure. The call sets the seed value to abc123.

```
DBMS_RANDOM.SEED('abc123');
```

## STRING

The STRING function returns a random VARCHAR2 string in a user-specified format. Syntax:

```
result VARCHAR2 STRING(opt IN CHAR, len IN NUMBER)
```

### Parameters

opt: The formatting option for the returned string. The following table lists possible values of the option parameter.

| Option | Description |
|---|---|
| u or U | Uppercase alpha string |
| l or L | Lowercase alpha string |
| a or A | Mixed case string |
| x or X | Uppercase alpha-numeric string |
| p or P | Printable characters |

len: The length of the returned string.

result: The result parameter is a random value of type VARCHAR2.

**Examples**

The following code snippet demonstrates a call to the STRING function. The call returns a random alpha-numeric character string that is 10 characters in length.

```
x := DBMS_RANDOM.STRING('X', 10);
```

### TERMINATE

The TERMINATE stored procedure has no effect. Syntax:

```
TERMINATE
```

We do not recommend that you use the TERMINATE stored procedure because it is only supported for compatibility.

### VALUE

The VALUE function returns a random NUMBER that is greater than or equal to 0, and less than 1, with 38 digit precision. The VALUE function has two forms. The syntax of the first form is:

```
result NUMBER VALUE()
```

**Parameters**

| Parameter | Description |
|---|---|
| result | A random value of type NUMBER. |

**Examples**

The following code snippet demonstrates a call to the VALUE function. The call returns a random NUMBER:

```
x := DBMS_RANDOM.VALUE();
```

### VALUE

The VALUE function returns a random NUMBER with a value that is between user-specified boundaries. The VALUE function has two forms. The syntax of the second form is:

```
result NUMBER VALUE(low IN NUMBER, high IN NUMBER)
```

**Parameters**

| Parameter | Description |
|---|---|
| low | The lower boundary for the random value. The random value may be equal to low. |
| high | The upper boundary for the random value. The random value will be less than high. |
| result | A random value of type NUMBER. |

**Examples**

The following code snippet demonstrates a call to the VALUE function. The call returns a random NUMBER with a value that is greater than or equal to 1 and less than 100.

```
x := DBMS_RANDOM.VALUE(1, 100);
```

## 17.13 DBMS_RLS

The DBMS_RLS package enables the implementation of Virtual Private Database on certain POLARDB compatible with Oracle database objects.

**Table 17-25: DBMS_RLS functions and stored procedures**

| Function/stored procedure | Function or stored procedure | Return type | Description |
|---|---|---|---|
| ADD_POLICY( object_schema , object_name , policy_name, function_schema, policy_function [, statement_types [, update_check [, enable [, static_policy [, policy_type [, long_predicate [, sec_relevant_cols [, sec_relevant_cols_opt ]]]]]]]]) | Stored procedure | N/A | Adds a security policy to a database object. |
| DROP_POLICY( object_schema , object_name, policy_name) | Stored procedure | N/A | Removes a security policy from a database object. |

| Function/stored procedure | Function or stored procedure | Return type | Description |
|---|---|---|---|
| ENABLE_POLICY (object_schema , object_name, policy_name, enable ) | Stored procedure | N/A | Enables or disables a security policy. |

The DBMS_RLS package in POLARDB compatible with Oracle is partially implemented when compared to Oracle's version. POLARDB compatible with Oracle only supports the functions and stored procedures that are listed in the preceding table.

Virtual Private Database adopts fine-grained access control that uses security policies. Fine-grained access control in Virtual Private Database means that access to data can be controlled down to specific rows as defined by security policies.

The rules that encode a security policy are defined in a policy function. This policy function is an SPL function with specific input parameters and return values. The security policy is the association of the policy function to a particular database object, typically a table.

**Note:**

- In POLARDB compatible with Oracle, the policy function can be written in any language supported by POLARDB compatible with Oracle such as SQL, PL/pgSQL, and SPL. For example, apart from Oracle-compatible SPL languages, we can also use SQL and PL/pgSQL languages.
- Virtual Private Database of POLARDB compatible with Oracle only supports one type of database objects: tables. Policies cannot be applied to views or synonyms.

The benefits of using Virtual Private Database are described as follows:

- Virtual Private Database provides a fine-grained level of security. Database object level privileges given by the GRANT command determine access privileges to the entire instance of a database object. In contrast, Virtual Private Database provides access control for the individual rows of a database object instance.
- A different security policy can be applied depending upon the type of SQL command ( INSERT, UPDATE, DELETE, or SELECT).

- The security policy is dynamic and can vary for each applicable SQL command affecting the database object. The security policy is determined by multiples factors, such as the session user of the application accessing the database object.
- Invocation of the security policy is transparent to all applications that access the database object. Therefore, you do not need to modify individual applications to apply the security policy.
- After a security policy is enabled, no applications (including new applications) can circumvent the security policy except by the system privilege that is mentioned in the following note.
- Even superusers cannot circumvent the security policy except by the system privilege that is mentioned in the following note.

**Note:**

The only way security policies can be circumvented is that the user has the EXEMPT ACCESS POLICY system privilege. The EXEMPT ACCESS POLICY privilege must be granted with extreme care because a user with this privilege is exempted from all policies in the database.

The DBMS_RLS package provides stored procedures to create, remove, enable, and disable policies.

The process for implementing Virtual Private Database is described as follows:

- Create a policy function. The function must have two input parameters of type VARCHAR2 . The first input parameter is used for the schema that contains the database object to which the policy is to be applied. The second input parameter is used for the name of the database object. The function must have a VARCHAR2 return type. The function must return a string in the form of a WHERE clause predicate. This predicate is dynamically appended as an AND condition to the SQL command that acts upon the database object . The rows that do not satisfy the policy function predicate are filtered out from the SQL command result set.
- Use the ADD_POLICY stored procedure to define a new policy, which associates a policy function with a database object. You can use the ADD_POLICY stored procedure to specify the types of SQL commands (INSERT, UPDATE, DELETE, or SELECT) to which the policy is to apply. You can specify whether to enable the policy at the time of its creation . You can also specify whether the policy can apply to newly inserted rows and the modified image of updated rows.

- Use the ENABLE_POLICY stored procedure to disable or enable an existing policy.
- Use the DROP_POLICY stored procedure to delete an existing policy. The DROP_POLICY stored procedure does not delete the policy function or the associated database object.

After policies are created, they can be viewed in the catalog views that are compatible with Oracle databases.

The SYS_CONTEXT function is often used with the DBMS_RLS package. Syntax:

```
SYS_CONTEXT(namespace, attribute)
```

- namespace is of the VARCHAR2 data type. The only valid value is USERENV. If another value is specified for this parameter, the function returns NULL.
- attribute is of the VARCHAR2 data type. The following table lists available values of the attribute parameter.

| Value of attribute | Equivalent value |
|---|---|
| SESSION_USER | pg_catalog.session_user |
| CURRENT_USER | pg_catalog.current_user |
| CURRENT_SCHEMA | pg_catalog.current_schema |
| HOST | pg_catalog.inet_host |
| IP_ADDRESS | pg_catalog.inet_client_addr |
| SERVER_HOST | pg_catalog.inet_server_addr |

**Note:**

The examples of the DBMS_RLS package use a modified copy of the sample emp table provided with POLARDB compatible with Oracle. A role named salesmgr is granted all privileges on the table. You can create the modified copy of the emp table named vpemp and the salesmgr role as follows:

```
CREATE TABLE public.vpemp AS SELECT empno, ename, job, sal, comm, deptno FROM
emp;
ALTER TABLE vpemp ADD authid VARCHAR2(12);
UPDATE vpemp SET authid = 'researchmgr' WHERE deptno = 20;
UPDATE vpemp SET authid = 'salesmgr' WHERE deptno = 30;
SELECT * FROM vpemp;

 empno | ename  |    job    |   sal   |  comm   | deptno |   authid
-------+--------+-----------+---------+---------+--------+-------------
  7782 | CLARK  | MANAGER   | 2450.00 |         |     10 |
  7839 | KING   | PRESIDENT | 5000.00 |         |     10 |
  7934 | MILLER | CLERK     | 1300.00 |         |     10 |
  7369 | SMITH  | CLERK     |  800.00 |         |     20 | researchmgr
  7566 | JONES  | MANAGER   | 2975.00 |         |     20 | researchmgr
  7788 | SCOTT  | ANALYST   | 3000.00 |         |     20 | researchmgr
  7876 | ADAMS  | CLERK     | 1100.00 |         |     20 | researchmgr
  7902 | FORD   | ANALYST   | 3000.00 |         |     20 | researchmgr
  7499 | ALLEN  | SALESMAN  | 1600.00 |  300.00 |     30 | salesmgr
  7521 | WARD   | SALESMAN  | 1250.00 |  500.00 |     30 | salesmgr
  7654 | MARTIN | SALESMAN  | 1250.00 | 1400.00 |     30 | salesmgr
  7698 | BLAKE  | MANAGER   | 2850.00 |         |     30 | salesmgr
  7844 | TURNER | SALESMAN  | 1500.00 |    0.00 |     30 | salesmgr
  7900 | JAMES  | CLERK     |  950.00 |         |     30 | salesmgr
(14 rows)

CREATE ROLE salesmgr WITH LOGIN PASSWORD 'password';
GRANT ALL ON vpemp TO salesmgr;
```

## ADD_POLICY

The ADD_POLICY stored procedure creates a new policy by associating a policy function with a database object.

You must be a superuser to call the ADD_POLICY stored procedure.

```
ADD_POLICY(object_schema VARCHAR2, object_name VARCHAR2,
  policy_name VARCHAR2, function_schema VARCHAR2,
  policy_function VARCHAR2
  [, statement_types VARCHAR2
  [, update_check BOOLEAN
  [, enable BOOLEAN
  [, static_policy BOOLEAN
  [, policy_type INTEGER
  [, long_predicate BOOLEAN
  [, sec_relevant_cols VARCHAR2
  [, sec_relevant_cols_opt INTEGER ]]]]]]]])
```

### Parameters

| Parameter | Description |
|---|---|
| object_schema | The name of the schema that contains the database object to which the policy is to be applied. |
| object_name | The name of the database object to which the policy is to be applied. A database object can have more than one policy applied to it. |
| policy_name | policy_name is the name assigned to the policy. The combination of the database object (identified by object_schema and object_name) and policy name must be unique within the database. |

| Parameter | Description |
|---|---|
| function_schema | The name of the schema that contains the policy function.<br>**Note:**<br>The policy function may belong to a package. In this case, function_schema must contain the name of the schema in which the package is defined. |
| policy_function | policy_function is the name of the SPL function that defines the rules of the security policy. The same function may be specified in more than one policy.<br>**Note:**<br>The policy function may belong to a package. In this case, policy_function must also contain the package name in dot notation (package_name.function_name). |
| statement_types | statement_types is a comma-separated list of SQL commands to which the policy applies. Valid SQL commands are INSERT, UPDATE, DELETE, and SELECT. Default value: INSERT, UPDATE, DELETE, SELECT.<br>**Note:**<br>POLARDB compatible with Oracle accepts INDEX as a statement type, but it is ignored. Policies are not applied to INDEX operations in POLARDB compatible with Oracle. |

| Parameter | Description |
|---|---|
| update_check | update_check applies to INSERT and UPDATE SQL commands only.<br>• If update_check is set to TRUE, the policy is applied to newly inserted rows and to the modified image of updated rows. If a new or modified row does not qualify according to the policy function predicate, the INSERT or UPDATE command throws an exception and no rows are inserted or modified.<br>• If update_check is set to FALSE, the policy is not applied to newly inserted rows or the modified image of updated rows. Therefore, a newly inserted row may not appear in the result set of a subsequent SQL command that invokes the same policy. Similarly, rows which qualified according to the policy prior to an UPDATE command may not appear in the result set of a subsequent SQL command that invokes the same policy. |
| enable | • If enable is set to TRUE, the policy is enabled and applied to the SQL commands specified by the statement_types parameter.<br>• If enable is set to FALSE, the policy is disabled and not applied to SQL commands. You can enable the policy by using the ENABLE_POLICY stored procedure. The default value is TRUE. |

| Parameter | Description |
|---|---|
| static_policy | - In Oracle, if static_policy is set to TRUE, the policy is static. The policy function is evaluated once per database object the first time it is invoked by a policy on the database object. The resulting predicate string of the policy function is saved in memory. In this case, when the database server instance is running, the predicate string can be reused for all invocations of that policy on that database object.<br>- In Oracle, if static_policy is set to FALSE, the policy is dynamic. The policy function is re-evaluated and the predicate string of the policy function is re-generated for all invocations of the policy.<br>- The default value is FALSE.<br>**Note:**<br>- In Oracle 10g, the policy_type parameter was introduced, which is intended to replace the static_policy parameter. In Oracle, if the policy_type parameter is not set to its default value (NULL), the policy_type parameter setting overrides the static_policy setting.<br>- POLARDB compatible with Oracle ignores the setting of the static_policy parameter. POLARDB compatible with Oracle implements only the dynamic policy, regardless of the setting of the static_policy parameter. |

| Parameter | Description |
|---|---|
| policy_type | In Oracle, policy_type determines when the policy function is re-evaluated. Therefore, it also determines whether and when the predicate string returned by the policy function changes. The default value is NULL.<br>**Note:**<br>POLARDB compatible with Oracle ignores the setting of the policy_type parameter. POLARDB compatible with Oracle always assumes a dynamic policy. |
| long_predicate | In Oracle, if long_predicate is set to TRUE, predicates can be up to 32 KB in length. Otherwise, predicates are limited to 4 KB in length. The default value is FALSE.<br>**Note:**<br>POLARDB compatible with Oracle ignores the setting of the long_predicate parameter. A POLARDB compatible with Oracle policy function can return a predicate of unlimited length for all practical purposes. |
| sec_relevant_cols | sec_relevant_cols is a comma-separated list of columns of object_name. This parameter provides column-level Virtual Private Database for the listed columns. The policy is enforced if a listed column is referenced in an SQL command of a type specified in statement_types. The policy is not enforced if no such columns are referenced.<br>The default value is NULL. The same effect is achieved if all columns of the database object are included in sec_relevant_cols. |

| Parameter | Description |
|---|---|
| sec_relevant_cols_opt | In Oracle, if sec_relevant_cols_opt is set to DBMS_RLS.ALL_ROWS (INTEGER constant of value 1), the columns listed in sec_relevant_cols return NULL on all rows where the applied policy predicate is false. If sec_relevant_cols_opt is not set to DBMS_RLS.ALL_ROWS, these rows will not be returned in the result set. The default value is NULL.<br>**Note:**<br>POLARDB compatible with Oracle does not support the DBMS_RLS.ALL_ROWS function. If sec_relevant_cols_opt is set to DBMS_RLS.ALL_ROWS (INTEGER value of 1), POLARDB compatible with Oracle will throw an error. |

**Examples**

This example uses the following policy function:

```
CREATE OR REPLACE FUNCTION verify_session_user (
    p_schema       VARCHAR2,
    p_object       VARCHAR2
)
RETURN VARCHAR2
IS
BEGIN
    RETURN 'authid = SYS_CONTEXT(''USERENV'', ''SESSION_USER'')';
END;
```

This function generates the predicate authid = SYS_CONTEXT('USERENV', 'SESSION_USER'), which is added to the WHERE clause of each SQL command of the type specified in the ADD_POLICY stored procedure.

This limits the effect of the SQL command to rows where the content of the authid column is the same as the session user.

**Note:**

This example uses the SYS_CONTEXT function to return the login user name. In Oracle, the SYS_CONTEXT function returns attributes of an application context. The first parameter of the SYS_CONTEXT function is the name of an application context. The second parameter is the name of an attribute set within the application context. USERENV is a special built-in namespace that describes the current session. POLARDB compatible with Oracle does not support application contexts, but supports this specific usage of the SYS_CONTEXT function.

The following anonymous block calls the ADD_POLICY stored procedure. This is to create a policy named secure_update. Then, the policy will be applied to the vpemp table by using the verify_session_user function regardless of whether an INSERT, UPDATE, or DELETE SQL command is provided when the vpemp table is referenced.

```
DECLARE
    v_object_schema         VARCHAR2(30) := 'public';
    v_object_name           VARCHAR2(30) := 'vpemp';
    v_policy_name           VARCHAR2(30) := 'secure_update';
    v_function_schema       VARCHAR2(30) := 'enterprisedb';
    v_policy_function       VARCHAR2(30) := 'verify_session_user';
    v_statement_types       VARCHAR2(30) := 'INSERT,UPDATE,DELETE';
    v_update_check          BOOLEAN      := TRUE;
    v_enable                BOOLEAN      := TRUE;
BEGIN
    DBMS_RLS.ADD_POLICY(
        v_object_schema,
        v_object_name,
        v_policy_name,
        v_function_schema,
        v_policy_function,
        v_statement_types,
        v_update_check,
        v_enable
    );
END;
```

After the policy is created, a terminal session is started by the salesmgr user. The following query shows the content of the vpemp table.

```
edb=# \c edb salesmgr
Password for user salesmgr:
You are now connected to database "edb" as user "salesmgr".
edb=> SELECT * FROM vpemp;
 empno | ename  |    job    |   sal   |  comm   | deptno |   authid
-------+--------+-----------+---------+---------+--------+-------------
  7782 | CLARK  | MANAGER   | 2450.00 |         |     10 |
  7839 | KING   | PRESIDENT | 5000.00 |         |     10 |
  7934 | MILLER | CLERK     | 1300.00 |         |     10 |
  7369 | SMITH  | CLERK     |  800.00 |         |     20 | researchmgr
  7566 | JONES  | MANAGER   | 2975.00 |         |     20 | researchmgr
  7788 | SCOTT  | ANALYST   | 3000.00 |         |     20 | researchmgr
  7876 | ADAMS  | CLERK     | 1100.00 |         |     20 | researchmgr
  7902 | FORD   | ANALYST   | 3000.00 |         |     20 | researchmgr
  7499 | ALLEN  | SALESMAN  | 1600.00 |  300.00 |     30 | salesmgr
  7521 | WARD   | SALESMAN  | 1250.00 |  500.00 |     30 | salesmgr
  7654 | MARTIN | SALESMAN  | 1250.00 | 1400.00 |     30 | salesmgr
  7698 | BLAKE  | MANAGER   | 2850.00 |         |     30 | salesmgr
  7844 | TURNER | SALESMAN  | 1500.00 |    0.00 |     30 | salesmgr
  7900 | JAMES  | CLERK     |  950.00 |         |     30 | salesmgr
```

```
(14 rows)
```

An unqualified UPDATE command (without a WHERE clause) is issued by the salesmgr user:

```
edb=> UPDATE vpemp SET comm = sal * .75;
UPDATE 6
```

Instead of updating all rows in the table, the policy restricts the effect of the update to rows where the authid column contains the salesmgr value. The salesmgr value is specified by the policy function predicate: authid = SYS_CONTEXT('USERENV', 'SESSION_USER').

The following query shows that the comm column has been changed for rows where authid contains salesmgr. All other rows are unchanged.

```
edb=> SELECT * FROM vpemp;
 empno | ename  |    job    |   sal   |  comm   | deptno |   authid
-------+--------+-----------+---------+---------+--------+-------------
  7782 | CLARK  | MANAGER   | 2450.00 |         |     10 |
  7839 | KING   | PRESIDENT | 5000.00 |         |     10 |
  7934 | MILLER | CLERK     | 1300.00 |         |     10 |
  7369 | SMITH  | CLERK     |  800.00 |         |     20 | researchmgr
  7566 | JONES  | MANAGER   | 2975.00 |         |     20 | researchmgr
  7788 | SCOTT  | ANALYST   | 3000.00 |         |     20 | researchmgr
  7876 | ADAMS  | CLERK     | 1100.00 |         |     20 | researchmgr
  7902 | FORD   | ANALYST   | 3000.00 |         |     20 | researchmgr
  7499 | ALLEN  | SALESMAN  | 1600.00 | 1200.00 |     30 | salesmgr
  7521 | WARD   | SALESMAN  | 1250.00 |  937.50 |     30 | salesmgr
  7654 | MARTIN | SALESMAN  | 1250.00 |  937.50 |     30 | salesmgr
  7698 | BLAKE  | MANAGER   | 2850.00 | 2137.50 |     30 | salesmgr
  7844 | TURNER | SALESMAN  | 1500.00 | 1125.00 |     30 | salesmgr
  7900 | JAMES  | CLERK     |  950.00 |  712.50 |     30 | salesmgr
(14 rows)
```

The following INSERT command throws an exception because the update_check parameter was set to TRUE in the ADD_POLICY stored procedure. The policy is invalid because the researchmgr value specified for the authid column does not match the salesmgr session user.

```
edb=> INSERT INTO vpemp VALUES (9001,'SMITH','ANALYST',3200.00,NULL,20, 'researchmg
r');
ERROR:  policy with check option violation
DETAIL:  Policy predicate was evaluated to FALSE with the updated values
```

If update_check was set to FALSE, the preceding INSERT command would have succeeded.

The following example illustrates the use of the sec_relevant_cols parameter to apply a policy only when certain columns are referenced in the SQL command. The following policy function is used in this example, which selects rows where the employee salary is less than USD 2,000 per month.

```
CREATE OR REPLACE FUNCTION sal_lt_2000 (
    p_schema        VARCHAR2,
    p_object        VARCHAR2
)
RETURN VARCHAR2
IS
BEGIN
    RETURN 'sal < 2000';
END;
```

The policy is created so that it is enforced only if a SELECT command includes the sal or comm column.

```
DECLARE
    v_object_schema         VARCHAR2(30) := 'public';
    v_object_name           VARCHAR2(30) := 'vpemp';
    v_policy_name           VARCHAR2(30) := 'secure_salary';
    v_function_schema       VARCHAR2(30) := 'enterprisedb';
    v_policy_function       VARCHAR2(30) := 'sal_lt_2000';
    v_statement_types       VARCHAR2(30) := 'SELECT';
    v_sec_relevant_cols     VARCHAR2(30) := 'sal,comm';
BEGIN
    DBMS_RLS.ADD_POLICY(
        v_object_schema,
        v_object_name,
        v_policy_name,
        v_function_schema,
        v_policy_function,
        v_statement_types,
        sec_relevant_cols => v_sec_relevant_cols
    );
END;
```

If a query does not reference the sal or comm column, the policy is not applied. The following query returns all 14 rows of the vpemp table:

```
edb=# SELECT empno, ename, job, deptno, authid FROM vpemp;
 empno | ename  |    job    | deptno |   authid
-------+--------+-----------+--------+-------------
  7782 | CLARK  | MANAGER   |     10 |
  7839 | KING   | PRESIDENT |     10 |
  7934 | MILLER | CLERK     |     10 |
  7369 | SMITH  | CLERK     |     20 | researchmgr
  7566 | JONES  | MANAGER   |     20 | researchmgr
  7788 | SCOTT  | ANALYST   |     20 | researchmgr
  7876 | ADAMS  | CLERK     |     20 | researchmgr
  7902 | FORD   | ANALYST   |     20 | researchmgr
  7499 | ALLEN  | SALESMAN  |     30 | salesmgr
  7521 | WARD   | SALESMAN  |     30 | salesmgr
  7654 | MARTIN | SALESMAN  |     30 | salesmgr
  7698 | BLAKE  | MANAGER   |     30 | salesmgr
  7844 | TURNER | SALESMAN  |     30 | salesmgr
  7900 | JAMES  | CLERK     |     30 | salesmgr
(14 rows)
```

If the query references the sal or comm column, the policy is applied to the query. This query deletes rows where sal is greater than or equal to 2000, as shown in the following example:

```
edb=# SELECT empno, ename, job, sal, comm, deptno, authid FROM vpemp;
```

```
 empno | ename  |   job    |   sal   |  comm   | deptno |   authid
-------+--------+----------+---------+---------+--------+-------------
  7934 | MILLER | CLERK    | 1300.00 |         |     10 |
  7369 | SMITH  | CLERK    |  800.00 |         |     20 | researchmgr
  7876 | ADAMS  | CLERK    | 1100.00 |         |     20 | researchmgr
  7499 | ALLEN  | SALESMAN | 1600.00 | 1200.00 |     30 | salesmgr
  7521 | WARD   | SALESMAN | 1250.00 |  937.50 |     30 | salesmgr
  7654 | MARTIN | SALESMAN | 1250.00 |  937.50 |     30 | salesmgr
  7844 | TURNER | SALESMAN | 1500.00 | 1125.00 |     30 | salesmgr
  7900 | JAMES  | CLERK    |  950.00 |  712.50 |     30 | salesmgr
(8 rows)
```

## DROP_POLICY

The DROP_POLICY stored procedure deletes an existing policy. However, the DROP_POLICY stored procedure cannot delete the policy function and database object associated with the policy.

You must be a superuser to execute the DROP_POLICY stored procedure.

```
DROP_POLICY(object_schema VARCHAR2, object_name VARCHAR2,

  policy_name VARCHAR2)
```

### Parameters

| Parameter | Description |
|---|---|
| object_schema | The name of the schema that contains the database object to which the policy applies. |
| object_name | The name of the database object to which the policy applies. |
| policy_name | The name of the policy to be deleted. |

### Examples

The following example deletes the secure_update policy on the public.vpemp table:

```
DECLARE
    v_object_schema         VARCHAR2(30) := 'public';
    v_object_name           VARCHAR2(30) := 'vpemp';
    v_policy_name           VARCHAR2(30) := 'secure_update';
BEGIN
    DBMS_RLS.DROP_POLICY(
        v_object_schema,
        v_object_name,
        v_policy_name
    );
```

```
END;
```

### ENABLE_POLICY

The ENABLE_POLICY stored procedure enables or disables an existing policy on the specified database object.

You must be a superuser to execute the ENABLE_POLICY stored procedure.

```
ENABLE_POLICY(object_schema VARCHAR2, object_name VARCHAR2,
  policy_name VARCHAR2, enable BOOLEAN)
```

**Parameters**

| Parameter | Description |
|---|---|
| object_schema | The name of the schema that contains the database object to which the policy applies. |
| object_name | The name of the database object to which the policy applies. |
| policy_name | The name of the policy to be enabled or disabled. |
| enable | If the enable parameter is set to TRUE, the policy is enabled. If the enable parameter is set to FALSE, the policy is disabled. |

**Examples**

The following example disables the secure_update policy on the public.vpemp table:

```
DECLARE
    v_object_schema         VARCHAR2(30) := 'public';
    v_object_name           VARCHAR2(30) := 'vpemp';
    v_policy_name           VARCHAR2(30) := 'secure_update';
    v_enable                BOOLEAN := FALSE;
BEGIN
    DBMS_RLS.ENABLE_POLICY(
        v_object_schema,
        v_object_name,
        v_policy_name,
        v_enable
    );
```

```
END;
```

## 17.14 DBMS_SESSION

PolarDB databases compatible with Oracle support the following DBMS_SESSION.SET_ROLE procedure:

| Function/Procedure | Return type | Description |
|---|---|---|
| SET_ROLE(role_cmd) | N/A | Executes the SET ROLE statement followed by the string value specified in role_cmd. |

The implementation of DBMS_AQ in PolarDB databases compatible with Oracle is a partial implementation when compared with native Oracle. Only DBMS_SESSION.SET_ROLE is supported.

### SET_ROLE

The SET_ROLE procedure sets the current session user to the role specified in role_cmd. After the current session invokes the SET_ROLE procedure, the session uses the permissions assigned to the specified role. The procedure has the following signature:

```
SET_ROLE(role_cmd)
```

The SET_ROLE procedure appends the value specified for role_cmd to the SET ROLE statement, and then invokes the statement.

**Parameters**

| Parameter | Description |
|---|---|
| role_cmd | Specifies a role name in the form of a string value. |

**Examples**

You can run the SET ROLE command to call the SET_ROLE procedure and set the identity of the current session user to manager. The following example shows this call:

```
exec DBMS_SESSION.SET_ROLE('manager');
```

## 17.15 DBMS_SQL

The DBMS_SQL package provides an application interface compatible with Oracle databases to the POLARDB compatible with Oracle dynamic SQL functionality. By using the DBMS_SQL package, you can construct queries and other commands at run time, rather than when you write the application. POLARDB compatible with Oracle offers native support for dynamic SQL. The DBMS_SQL package provides a method of using dynamic SQL that is compatible with Oracle databases without modifying your application.

The DBMS_SQL package assumes that the current user has the required permissions when running dynamic SQL statements.

**Table 17-26: DBMS_SQL functions and stored procedures**

| Function/stored procedure | Function or stored procedure | Return type | Description |
|---|---|---|---|
| BIND_VARIABLE( c, name, value [, out_value_size ]) | Stored procedure | N/A | Binds a value to a variable. |
| BIND_VARIABLE_CHAR(c, name , value [, out_value_ size ]) | Stored procedure | N/A | Binds a CHAR value to a variable. |
| BIND_VARIABLE_RAW (c, name, value [, out_value_size ]) | Stored procedure | N/A | Binds a RAW value to a variable. |
| CLOSE_CURSOR(c IN OUT) | Stored procedure | N/A | Closes a cursor. |
| COLUMN_VALUE(c, position, value OUT [, column_error OUT [, actual_length OUT ]]) | Stored procedure | N/A | Returns a column value into a variable. |

| Function/stored procedure | Function or stored procedure | Return type | Description |
|---|---|---|---|
| COLUMN_VALUE_CHAR(c, position , value OUT [, column_error OUT [, actual_length OUT ]]) | Stored procedure | N/A | Returns a CHAR column value into a variable. |
| COLUMN_VALUE_RAW(c, position , value OUT [, column_error OUT [, actual_length OUT ]]) | Stored procedure | N/A | Returns a RAW column value into a variable. |
| DEFINE_COLUMN(c, position, column [, column_size ]) | Stored procedure | N/A | Defines a column in the SELECT list. |
| DEFINE_COLUMN_CHAR(c, position, column, column_size) | Stored procedure | N/A | Defines a CHAR column in the SELECT list. |
| DEFINE_COLUMN_RAW(c, position , column, column_size) | Stored procedure | N/A | Defines a RAW column in the SELECT list. |
| DESCRIBE_COLUMNS | Stored procedure | N/A | Defines columns to hold a cursor result set. |
| EXECUTE(c) | Function | INTEGER | Executes a cursor. |
| EXECUTE_AND_FETCH(c [, exact ]) | Function | INTEGER | Executes a cursor and fetches a single row. |
| FETCH_ROWS(c) | Function | INTEGER | Fetches rows from the cursor. |
| IS_OPEN(c) | Function | BOOLEAN | Check whether a cursor is open. |
| LAST_ROW_COUNT | Function | INTEGER | Returns the cumulative number of rows fetched. |
| OPEN_CURSOR | Function | INTEGER | Opens a cursor. |

| Function/stored procedure | Function or stored procedure | Return type | Description |
|---|---|---|---|
| PARSE(c, statement, language_flag) | Stored procedure | N/A | Parses a statement. |

The DBMS_SQL package in POLARDB compatible with Oracle is partially implemented when compared to Oracle's version. POLARDB compatible with Oracle only supports the functions and stored procedures that are listed in the preceding table.

The following table lists the public variables that are available in the DBMS_SQL package.

**Table 17-27: DBMS_SQL public variables**

| Public variable | Data type | Value | Description |
|---|---|---|---|
| native | INTEGER | 1 | Provided for compatibility with Oracle syntax. For more information, see DBMS_SQL.PARSE . |
| V6 | INTEGER | 2 | Provided for compatibility with Oracle syntax. For more information, see DBMS_SQL.PARSE . |
| V7 | INTEGER | 3 | Provided for compatibility with Oracle syntax. For more information, see DBMS_SQL.PARSE . |

### BIND_VARIABLE

The BIND_VARIABLE stored procedure provides the capability to associate a value with an IN or IN OUT bind variable in an SQL command.

```
BIND_VARIABLE(c INTEGER, name VARCHAR2,
  value { BLOB | CLOB | DATE | FLOAT | INTEGER | NUMBER |
        TIMESTAMP | VARCHAR2 }
  [, out_value_size INTEGER ])
```

**Parameters**

| Parameter | Description |
|---|---|
| c | The ID of the cursor for the SQL command with bind variables. |
| name | The name of the bind variable in the SQL command. |
| value | The value to be assigned. |
| out_value_size | If name is an IN OUT variable, this parameter defines the maximum length of the output value. If this parameter is not specified, the length of the current value is the maximum length by default. |

**Examples**

The following anonymous block uses bind variables to insert a row into the emp table.

```
DECLARE
    curid          INTEGER;
    v_sql          VARCHAR2(150) := 'INSERT INTO emp VALUES ' ||
                   '(:p_empno, :p_ename, :p_job, :p_mgr, ' ||
                   ':p_hiredate, :p_sal, :p_comm, :p_deptno)';
    v_empno        emp.empno%TYPE;
    v_ename        emp.ename%TYPE;
    v_job          emp.job%TYPE;
    v_mgr          emp.mgr%TYPE;
    v_hiredate     emp.hiredate%TYPE;
    v_sal          emp.sal%TYPE;
    v_comm         emp.comm%TYPE;
    v_deptno       emp.deptno%TYPE;
    v_status       INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    v_empno    := 9001;
    v_ename    := 'JONES';
    v_job      := 'SALESMAN';
    v_mgr      := 7369;
    v_hiredate := TO_DATE('13-DEC-07','DD-MON-YY');
    v_sal      := 8500.00;
    v_comm     := 1500.00;
    v_deptno   := 40;
    DBMS_SQL.BIND_VARIABLE(curid,':p_empno',v_empno);
    DBMS_SQL.BIND_VARIABLE(curid,':p_ename',v_ename);
    DBMS_SQL.BIND_VARIABLE(curid,':p_job',v_job);
    DBMS_SQL.BIND_VARIABLE(curid,':p_mgr',v_mgr);
    DBMS_SQL.BIND_VARIABLE(curid,':p_hiredate',v_hiredate);
    DBMS_SQL.BIND_VARIABLE(curid,':p_sal',v_sal);
    DBMS_SQL.BIND_VARIABLE(curid,':p_comm',v_comm);
    DBMS_SQL.BIND_VARIABLE(curid,':p_deptno',v_deptno);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;

Number of rows processed: 1
```

### BIND_VARIABLE_CHAR

The BIND_VARIABLE_CHAR stored procedure provides the capability to associate a CHAR value with an IN or IN OUT bind variable in an SQL command.

```
BIND_VARIABLE_CHAR(c INTEGER, name VARCHAR2, value CHAR
  [, out_value_size INTEGER ])
```

**Parameters**

| Parameter | Description |
|---|---|
| c | The ID of the cursor for the SQL command with bind variables. |
| name | The name of the bind variable in the SQL command. |
| value | The value of type CHAR to be assigned. |
| out_value_size | If name is an IN OUT variable, this parameter defines the maximum length of the output value. If this parameter is not specified, the length of the current value is the maximum length by default. |

### BIND_VARIABLE_RAW

The BIND_VARIABLE_RAW stored procedure provides the capability to associate a RAW value with an IN or IN OUT bind variable in an SQL command.

```
BIND_VARIABLE_RAW(c INTEGER, name VARCHAR2, value RAW
  [, out_value_size INTEGER ])
```

**Parameters**

| Parameter | Description |
|---|---|
| c | The ID of the cursor for the SQL command with bind variables. |
| name | The name of the bind variable in the SQL command. |
| value | The value of type RAW to be assigned. |

| Parameter | Description |
|---|---|
| out_value_size | If name is an IN OUT variable, this parameter defines the maximum length of the output value. If this parameter is not specified, the length of the current value is the maximum length by default. |

### CLOSE_CURSOR

The CLOSE_CURSOR stored procedure closes a cursor. When the cursor is closed, resources allocated to the cursor are released and the cursor can no longer be used.

```
CLOSE_CURSOR(c IN OUT INTEGER)
```

#### Parameters

| Parameter | Description |
|---|---|
| c | The ID of the cursor to be closed. |

#### Examples

The following example shows how to close an open cursor.

```
DECLARE
    curid         INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
            .
            .
            .
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

### COLUMN_VALUE

The COLUMN_VALUE stored procedure defines a variable to receive a value from a cursor.

```
COLUMN_VALUE(c INTEGER, position INTEGER, value OUT { BLOB |
  CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 }
  [, column_error OUT NUMBER [, actual_length OUT INTEGER ]])
```

#### Parameters

| Parameter | Description |
|---|---|
| c | The ID of the cursor that returns data to the variable being defined. |

| Parameter | Description |
|---|---|
| position | The position of the returned data within the cursor. The first value in the cursor is position 1. |
| value | The variable that receives the data returned in the cursor by a prior fetch call. |
| column_error | If an error occurs, this parameter indicates the error code associated with the column. |
| actual_length | The actual length of the data before truncation. |

**Examples**

The following example shows the portion of an anonymous block that receives the values from a cursor by using the COLUMN_VALUE stored procedure.

```
DECLARE
    curid           INTEGER;
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_sql           VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                                'comm FROM emp';
    v_status        INTEGER;
BEGIN
            .
            .
            .
    LOOP
        v_status := DBMS_SQL.FETCH_ROWS(curid);
        EXIT WHEN v_status = 0;
        DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
        DBMS_SQL.COLUMN_VALUE(curid,2,v_ename);
        DBMS_SQL.COLUMN_VALUE(curid,3,v_hiredate);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,5,v_comm);
        DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,10) || '  ' ||
            TO_CHAR(v_hiredate,'yyyy-mm-dd') || ' ' ||
            TO_CHAR(v_sal,'9,999.99') || ' ' ||
            TO_CHAR(NVL(v_comm,0),'9,999.99'));
    END LOOP;
    DBMS_SQL.CLOSE_CURSOR(curid);
```

```
END;
```

### COLUMN_VALUE_CHAR

The COLUMN_VALUE_CHAR stored procedure defines a variable to receive a CHAR value from a cursor.

```
COLUMN_VALUE_CHAR(c INTEGER, position INTEGER, value OUT CHAR
  [, column_error OUT NUMBER [, actual_length OUT INTEGER ]])
```

**Parameters**

| Parameter | Description |
|---|---|
| c | The ID of the cursor that returns data to the variable being defined. |
| position | The position of the returned data within the cursor. The first value in the cursor is position 1. |
| value | The variable of data type CHAR that receives the data returned in the cursor by a prior fetch call. |
| column_error | If an error occurs, this parameter indicates the error code associated with the column. |
| actual_length | The actual length of the data before truncation. |

### COLUMN_VALUE_RAW

The COLUMN_VALUE_RAW stored procedure defines a variable to receive a RAW value from a cursor.

```
COLUMN_VALUE_RAW(c INTEGER, position INTEGER, value OUT RAW
  [, column_error OUT NUMBER [, actual_length OUT INTEGER ]])
```

**Parameters**

| Parameter | Description |
|---|---|
| c | The ID of the cursor that returns data to the variable being defined. |
| position | The position of the returned data within the cursor. The first value in the cursor is position 1. |

| Parameter | Description |
|---|---|
| value | The variable of data type RAW that receives the data returned in the cursor by a prior fetch call. |
| column_error | If an error occurs, this parameter indicates the error code associated with the column. |
| actual_length | The actual length of the data before truncation. |

### DEFINE_COLUMN

The DEFINE_COLUMN stored procedure defines a column or expression in the SELECT list that is to be returned and retrieved in a cursor.

```
DEFINE_COLUMN(c INTEGER, position INTEGER, column { BLOB |
  CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 }
  [, column_size INTEGER ])
```

**Parameters**

| Parameter | Description |
|---|---|
| c | The ID of the cursor associated with the SELECT command. |
| position | The position of the column or expression in the SELECT list that is being defined. |
| column | A variable that matches the data type of the column or expression in the specified position of the SELECT result set. |
| column_size | The maximum length of the returned data . The column_size parameter must be specified if the data type of the column is VARCHAR2. Returned data exceeding column_size is truncated to the maximum length specified by the column_size parameter. |

**Examples**

The following example shows how to use the DEFINE_COLUMN stored procedure to define the empno, ename, hiredate, sal, and comm columns of the emp table.

```
DECLARE
    curid         INTEGER;
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_sql           VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                                    'comm FROM emp';
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    DBMS_SQL.DEFINE_COLUMN(curid,1,v_empno);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_ename,10);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_hiredate);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_sal);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_comm);
            .
            .
            .
END;
```

The following example shows an alternative to the preceding example that produces the exact same results. Note that the lengths of the data types are irrelevant. The empno, sal, and comm columns will still return data equivalent to NUMBER(4) and NUMBER(7,2), respectively, even though v_num is defined as NUMBER(1). In the preceding example, each of the declarations in the COLUMN_VALUE stored procedure are configured with appropriate maximum sizes. The ename column will return data up to ten characters in length as defined by the length parameter in the DEFINE_COLUMN call. The length that is indicated by the data type VARCHAR2(1) declared for v_varchar is ignored. The actual size of the returned data is determined by the COLUMN_VALUE stored procedure.

```
DECLARE
    curid           INTEGER;
    v_num           NUMBER(1);
    v_varchar       VARCHAR2(1);
    v_date          DATE;
    v_sql           VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                                    'comm FROM emp';
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    DBMS_SQL.DEFINE_COLUMN(curid,1,v_num);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_varchar,10);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_date);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_num);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_num);
            .
            .
            .
```

```
END;
```

### DEFINE_COLUMN_CHAR

The DEFINE_COLUMN_CHAR stored procedure defines a CHAR column or expression in the SELECT list that is to be returned and retrieved in a cursor.

```
DEFINE_COLUMN_CHAR(c INTEGER, position INTEGER, column CHAR, column_size INTEGER
)
```

**Parameters**

| Parameter | Description |
|---|---|
| c | The ID of the cursor associated with the SELECT command. |
| position | The position of the column or expression in the SELECT list that is being defined. |
| column | A CHAR variable. |
| column_size | The maximum length of the returned data . Returned data exceeding column_size is truncated to column_size characters. |

### DEFINE_COLUMN_RAW

The DEFINE_COLUMN_RAW stored procedure defines a RAW column or expression in the SELECT list that is to be returned and retrieved in a cursor.

```
DEFINE_COLUMN_RAW(c INTEGER, position INTEGER, column RAW,
  column_size INTEGER)
```

**Parameters**

| Parameter | Description |
|---|---|
| c | The ID of the cursor associated with the SELECT command. |
| position | The position of the column or expression in the SELECT list that is being defined. |
| column | A RAW variable. |
| column_size | The maximum length of the returned data . Returned data exceeding column_size is truncated to column_size characters. |

### DESCRIBE_COLUMNS

The DESCRIBE_COLUMNS stored procedure describes the columns returned by a cursor.

```
DESCRIBE_COLUMNS(c INTEGER, col_cnt OUT INTEGER, desc_t OUT
  DESC_TAB);
```

**Parameters**

| Parameter | Description |
|---|---|
| c | The ID of the cursor. |
| col_cnt | The number of columns in the cursor result set. |
| desc_tab | The table that contains a description of each column returned by the cursor. The descriptions are of type DESC_REC, and contain the following values: |

| Column name | Type |
|---|---|
| col_type | INTEGER |
| col_max_len | INTEGER |
| col_name | VARCHAR2(128) |
| col_name_len | INTEGER |
| col_schema_name | VARCHAR2(128) |
| col_schema_name_len | INTEGER |
| col_precision | INTEGER |
| col_scale | INTEGER |
| col_charsetid | INTEGER |
| col_charsetform | INTEGER |
| col_null_ok | BOOLEAN |

### EXECUTE

The EXECUTE function runs a parsed SQL command or SPL block.

```
status INTEGER EXECUTE(c INTEGER)
```

**Parameters**

| Parameter | Description |
|---|---|
| c | The cursor ID of the parsed SQL statement or SPL block to be run. |
| status | If the SQL command is DELETE, INSERT , or UPDATE, this parameter indicates the number of records processed. This parameter is meaningless for other commands. |

**Examples**

The following anonymous block inserts a row into the dept table.

```
DECLARE
    curid          INTEGER;
    v_sql          VARCHAR2(50);
    v_status       INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    v_sql := 'INSERT INTO dept VALUES (50, ''HR'', ''LOS ANGELES'')';
    DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

### EXECUTE_AND_FETCH

The EXECUTE_AND_FETCH function runs a parsed SELECT command and fetches one row.

```
status INTEGER EXECUTE_AND_FETCH(c INTEGER
  [, exact BOOLEAN ])
```

**Parameters**

| Parameter | Description |
|---|---|
| c | The ID of the cursor for the SELECT command to be run. |

| Parameter | Description |
|---|---|
| exact | If this parameter is set to TRUE, an exception occurs if the number of rows in the result set is not equal to 1. If this parameter is set to FALSE, no exception occurs. The default value of this parameter is FALSE. If this parameter is set to TRUE and the result set contains no records, a NO_DATE_FOUND exception will occur. If this parameter is set to TRUE and the result set contains multiple records, a TOO_MANY_ROWS exception will occur. |
| status | If a row is fetched, 1 is returned for this parameter. If no rows are fetched, 0 is returned for this parameter. If an exception occurs, no value is returned. |

**Examples**

The following stored procedure uses the EXECUTE_AND_FETCH function to retrieve one employee by using the employee’s name. If the employee is not found, or more than one employees with the same name are found, an exception will occur.

```
CREATE OR REPLACE PROCEDURE select_by_name(
    p_ename         emp.ename%TYPE
)
IS
    curid           INTEGER;
    v_empno         emp.empno%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_dname         dept.dname%TYPE;
    v_disp_date     VARCHAR2(10);
    v_sql           VARCHAR2(120) := 'SELECT empno, hiredate, sal, ' ||
                                     'NVL(comm, 0), dname ' ||
                                     'FROM emp e, dept d ' ||
                                     'WHERE ename = :p_ename ' ||
                                     'AND e.deptno = d.deptno';
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    DBMS_SQL.BIND_VARIABLE(curid,':p_ename',UPPER(p_ename));
    DBMS_SQL.DEFINE_COLUMN(curid,1,v_empno);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_hiredate);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_sal);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_comm);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_dname,14);
    v_status := DBMS_SQL.EXECUTE_AND_FETCH(curid,TRUE);
    DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
    DBMS_SQL.COLUMN_VALUE(curid,2,v_hiredate);
    DBMS_SQL.COLUMN_VALUE(curid,3,v_sal);
    DBMS_SQL.COLUMN_VALUE(curid,4,v_comm);
    DBMS_SQL.COLUMN_VALUE(curid,5,v_dname);
    v_disp_date := TO_CHAR(v_hiredate, 'MM/DD/YYYY');
    DBMS_OUTPUT.PUT_LINE('Number    : ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name      : ' || UPPER(p_ename));
    DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
    DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
    DBMS_OUTPUT.PUT_LINE('Department: ' || v_dname);
    DBMS_SQL.CLOSE_CURSOR(curid);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_ename || ' not found');
        DBMS_SQL.CLOSE_CURSOR(curid);
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('Too many employees named, ' ||
            p_ename || ', found');
        DBMS_SQL.CLOSE_CURSOR(curid);
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
        DBMS_SQL.CLOSE_CURSOR(curid);
END;

EXEC select_by_name('MARTIN')

Number    : 7654
Name      : MARTIN
Hire Date : 09/28/1981
Salary    : 1250
Commission: 1400
Department: SALES
```

### FETCH_ROWS

The FETCH_ROWS function retrieves a row from a cursor.

```
status INTEGER FETCH_ROWS(c INTEGER)
```

#### Parameters

| Parameter | Description |
|---|---|
| c | The ID of the cursor used to fetch a row. |
| status | If a row is fetched, 1 is returned for this parameter. If no rows are fetched, 0 is returned for this parameter. |

#### Examples

The following example fetches the rows from the emp table and displays the results.

```
DECLARE
```

```
    curid           INTEGER;
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_sql           VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                                    'comm FROM emp';
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    DBMS_SQL.DEFINE_COLUMN(curid,1,v_empno);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_ename,10);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_hiredate);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_sal);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_comm);

    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      HIREDATE    SAL       COMM');
    DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------  ' ||
        '--------');
    LOOP
        v_status := DBMS_SQL.FETCH_ROWS(curid);
        EXIT WHEN v_status = 0;
        DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
        DBMS_SQL.COLUMN_VALUE(curid,2,v_ename);
        DBMS_SQL.COLUMN_VALUE(curid,3,v_hiredate);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,5,v_comm);
        DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,10) || '  ' ||
            TO_CHAR(v_hiredate,'yyyy-mm-dd') || ' ' ||
            TO_CHAR(v_sal,'9,999.99') || ' ' ||
            TO_CHAR(NVL(v_comm,0),'9,999.99'));
    END LOOP;
    DBMS_SQL.CLOSE_CURSOR(curid);
END;

EMPNO  ENAME       HIREDATE     SAL       COMM
-----  ----------  ----------  --------  --------
7369   SMITH       1980-12-17    800.00       .00
7499   ALLEN       1981-02-20  1,600.00    300.00
7521   WARD        1981-02-22  1,250.00    500.00
7566   JONES       1981-04-02  2,975.00       .00
7654   MARTIN      1981-09-28  1,250.00  1,400.00
7698   BLAKE       1981-05-01  2,850.00       .00
7782   CLARK       1981-06-09  2,450.00       .00
7788   SCOTT       1987-04-19  3,000.00       .00
7839   KING        1981-11-17  5,000.00       .00
7844   TURNER      1981-09-08  1,500.00       .00
7876   ADAMS       1987-05-23  1,100.00       .00
7900   JAMES       1981-12-03    950.00       .00
7902   FORD        1981-12-03  3,000.00       .00
7934   MILLER      1982-01-23  1,300.00       .00
```

### IS_OPEN

The IS_OPEN function provides the capability to checks whether the specified cursor is open .

```
status BOOLEAN IS_OPEN(c INTEGER)
```

### Parameters

| Parameter | Description |
|---|---|
| c | The ID of the cursor to be checked. |
| status | If the cursor is open, this parameter is set to TRUE. If the cursor is not open, this parameter is set to FALSE. |

## LAST_ROW_COUNT

The LAST_ROW_COUNT function returns the total number of rows that are fetched.

```
rowcnt INTEGER LAST_ROW_COUNT
```

### Parameters

| Parameter | Description |
|---|---|
| rowcnt | The total number of fetched rows. |

### Examples

The following example uses the LAST_ROW_COUNT function to display the total number of rows fetched in the query.

```
DECLARE
    curid           INTEGER;
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_sql           VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                                    'comm FROM emp';
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    DBMS_SQL.DEFINE_COLUMN(curid,1,v_empno);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_ename,10);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_hiredate);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_sal);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_comm);

    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      HIREDATE    SAL       COMM');
    DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------  ' ||
        '--------');
    LOOP
        v_status := DBMS_SQL.FETCH_ROWS(curid);
        EXIT WHEN v_status = 0;
        DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
        DBMS_SQL.COLUMN_VALUE(curid,2,v_ename);
        DBMS_SQL.COLUMN_VALUE(curid,3,v_hiredate);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,5,v_comm);
        DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,10) || '  ' ||
            TO_CHAR(v_hiredate,'yyyy-mm-dd') || ' ' ||
            TO_CHAR(v_sal,'9,999.99') || ' ' ||
            TO_CHAR(NVL(v_comm,0),'9,999.99'));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Number of rows: ' || DBMS_SQL.LAST_ROW_COUNT);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;

EMPNO  ENAME       HIREDATE    SAL       COMM
-----  ----------  ----------  --------  --------
7369   SMITH       1980-12-17    800.00       .00
7499   ALLEN       1981-02-20  1,600.00    300.00
7521   WARD        1981-02-22  1,250.00    500.00
7566   JONES       1981-04-02  2,975.00       .00
7654   MARTIN      1981-09-28  1,250.00  1,400.00
7698   BLAKE       1981-05-01  2,850.00       .00
7782   CLARK       1981-06-09  2,450.00       .00
7788   SCOTT       1987-04-19  3,000.00       .00
7839   KING        1981-11-17  5,000.00       .00
7844   TURNER      1981-09-08  1,500.00       .00
7876   ADAMS       1987-05-23  1,100.00       .00
7900   JAMES       1981-12-03    950.00       .00
7902   FORD        1981-12-03  3,000.00       .00
7934   MILLER      1982-01-23  1,300.00       .00
Number of rows: 14
```

## OPEN_CURSOR

The OPEN_CURSOR function creates a new cursor. A cursor must be used to parse and execute a dynamic SQL statements. After being opened, a curser can be re-used with the same or different SQL statements without the need for you to close and re-open the cursor.

```
c INTEGER OPEN_CURSOR
```

### Parameters

| Parameter | Description |
|---|---|
| c | The ID of the newly created cursor. |

### Examples

The following example shows how to create a new cursor.

```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
            .
            .
    .
END;
```

## PARSE

The PARSE stored procedure parses an SQL command or SPL block. If the SQL command is a DDL command, it is immediately run and does not require calling the EXECUTE function.

```
PARSE(c INTEGER, statement VARCHAR2, language_flag INTEGER)
```

### Parameters

| Parameter | Description |
|---|---|
| c | The ID of an open cursor. |
| statement | The SQL command or SPL block to be parsed. An SQL command cannot end with a semicolon (;). An SPL block must end with a semicolon (;). |
| language_flag | The language flag provided for compatibility with Oracle syntax. Use DBMS_SQL.V6, DBMS_SQL.V7 or DBMS_SQL.native. This flag is ignored, and all syntax is assumed to be in POLARDB compatible with Oracle form. |

### Examples

The following anonymous block creates a table named job. Note that DDL statements are immediately run by the PARSE stored procedure and do not require calling the EXECUTE function.

```
DECLARE
    curid         INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid, 'CREATE TABLE job (jobno NUMBER(3), ' ||
        'jname VARCHAR2(9))',DBMS_SQL.native);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

The following code snippet inserts two rows into the job table.

```
DECLARE
    curid         INTEGER;
    v_sql         VARCHAR2(50);
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    v_sql := 'INSERT INTO job VALUES (100, ''ANALYST'')';
    DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
    v_sql := 'INSERT INTO job VALUES (200, ''CLERK'')';
    DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;

Number of rows processed: 1
Number of rows processed: 1
```

The following anonymous block uses the DBMS_SQL package to execute a block that contains two INSERT statements. Note that the end of the block contains a terminating semicolon (;), while in the preceding example, each individual INSERT statement does not have a terminating semicolon (;).

```
DECLARE
    curid          INTEGER;
    v_sql          VARCHAR2(100);
    v_status       INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    v_sql := 'BEGIN ' ||
               'INSERT INTO job VALUES (300, ''MANAGER''); '  ||
               'INSERT INTO job VALUES (400, ''SALESMAN''); ' ||
             'END;';
    DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

## 17.16 DBMS_UTILITY

The DBMS_UTILITY package supports the following utility programs:

| Function/Procedure | Category | Return type | Description |
|---|---|---|---|
| ANALYZE_DATABASE(method [, estimate_rows [, estimate_percent [, method_opt ]]]) | Procedure | N/A | Analyzes database tables. |
| ANALYZE_PART_OBJECT(schema, object_name [, object_type [, command_type [, command_opt [, sample_clause ]]]]) | Procedure | N/A | Analyzes a partitioned table. |

| Function/Procedure | Category | Return type | Description |
|---|---|---|---|
| ANALYZE_SCHEMA (schema, method [, estimate_rows [, estimate_percent [, method_opt ]]]) | Procedure | N/A | Analyzes schema tables. |
| CANONICALIZE(name , canon_name OUT, canon_len) | Procedure | N/A | Canonicalizes a string by using a method, for example , by removing space characters. |
| COMMA_TO_TABLE( list, tablen OUT, tab OUT) | Procedure | N/A | Converts a comma -delimited list of names to a table of names. |
| DB_VERSION(version OUT, compatibility OUT) | Procedure | N/A | Retrieves a database version. |
| EXEC_DDL_STATEMENT(parse_string) | Procedure | N/A | Executes a data description language (DDL) statement. |
| FORMAT_CALL_STACK | Function | TEXT | Formats the current call stack. |
| GET_CPU_TIME | Function | NUMBER | Retrieves the current CPU time. |
| GET_DEPENDENCY( type, schema, name) | Procedure | N/A | Retrieve objects that are dependent upon the specified object. |
| GET_HASH_VALUE (name, base, hash_size) | Function | NUMBER | Computes a hash value. |
| GET_PARAMETER_VALUE(parnam , intval OUT, strval OUT) | Procedure | BINARY_INTEGER | Retrieves database initialization parameter settings. |
| GET_TIME | Function | NUMBER | Retrieves the current time. |

| Function/Procedure | Category | Return type | Description |
|---|---|---|---|
| NAME_TOKENIZE( name, a OUT, b OUT , c OUT, dblink OUT, nextpos OUT) | Procedure | N/A | Parses the specified name into its component parts. |
| TABLE_TO_COMMA( tab, tablen OUT, list OUT) | Procedure | N/A | Converts a table of names to a comma-delimited list. |

The implementation of DBMS_UTILITY in PolarDB databases compatible with Oracle is a partial implementation when compared with native Oracle. Only those functions and procedures listed in the preceding table are supported.

The following table lists the public variables available in the DBMS_UTILITY package.

| Public variable | Data type | Value | Description |
|---|---|---|---|
| inv_error_on_restrictions | PLS_INTEGER | 1 | Used by the INVALIDATE procedure. |
| lname_array | TABLE | - | Lists long names. |
| uncl_array | TABLE | - | Lists users and names. |

### LNAME_ARRAY

The LNAME_ARRAY variable is used to store lists of long names including fully-qualified names.

```
TYPE lname_array IS TABLE OF VARCHAR2(4000) INDEX BY BINARY_INTEGER;
```

### UNCL_ARRAY

The UNCL_ARRAY variable is used to store lists of users and names.

```
TYPE uncl_array IS TABLE OF VARCHAR2(227) INDEX BY BINARY_INTEGER;
```

### ANALYZE_DATABASE, ANALYZE SCHEMA, and ANALYZE PART_OBJECT

You can use the ANALYZE_DATABASE(), ANALYZE_SCHEMA() and ANALYZE_PART_OBJECT() procedures to gather statistics on tables in a database. When you execute the ANALYZE statement, Postgres samples the data in a table and records distribution statistics in the pg_statistics system table.

ANALYZE_DATABASE, ANALYZE_SCHEMA, and ANALYZE_PART_OBJECT differ in the number of tables that are processed:

- ANALYZE_DATABASE analyzes all tables in all schemas within the current database.
- ANALYZE_SCHEMA analyzes all tables in a specified schema within the current database.
- ANALYZE_PART_OBJECT analyzes a single table.

The ANALYZE command has the following syntax:

```
ANALYZE_DATABASE(method VARCHAR2 [, estimate_rows NUMBER
  [, estimate_percent NUMBER [, method_opt VARCHAR2 ]]])

ANALYZE_SCHEMA(schema VARCHAR2, method VARCHAR2
  [, estimate_rows NUMBER [, estimate_percent NUMBER
  [, method_opt VARCHAR2 ]]])

ANALYZE_PART_OBJECT(schema VARCHAR2, object_name VARCHAR2
  [, object_type CHAR [, command_type CHAR
  [, command_opt VARCHAR2 [, sample_clause ]]]])
```

**Parameters**

- ANALYZE_DATABASE and ANALYZE_SCHEMA

| Parameter | Description |
|---|---|
| method | The method parameter specifies whether the ANALYZE procedure populates the pg_statistics table or removes entries from the pg_statistics table. If you specify a method of DELETE, the ANALYZE procedure removes the relevant rows from pg_statistics. If you specify a method of COMPUTE or ESTIMATE, the ANALYZE procedure analyzes one or more multiple tables and records the distribution information in pg_statistics. The COMPUTE and ESTIMATE methods have no difference. Both methods execute the Postgres ANALYZE statement. All other parameters are validated and then ignored. |

| Parameter | Description |
|---|---|
| estimate_rows | The number of rows on which the estimated statistics is based. One of estimate_rows or estimate_percent must be specified if the ESTIMATE method is specified.<br>This parameter is ignored, but is included for compatibility. |
| estimate_percent | The percentage of rows on which the estimated statistics is based. One of estimate_rows or estimate_percent must be specified if the ESTIMATE method is specified.<br>This parameter is ignored, but is included for compatibility. |
| method_opt | The object types to be analyzed. The following combinations are supported:<br>[ FOR TABLE ]<br>[ FOR ALL [ INDEXED ] COLUMNS ] [ SIZE n ]<br>[ FOR ALL INDEXES ]<br>This parameter is ignored, but is included for compatibility. |

- ANALYZE_PART_OBJECT

| Parameter | Description |
|---|---|
| schema | The name of the schema whose objects are analyzed. |
| object_name | The name of the partitioned object to be analyzed. |
| object_type | The type of object to be analyzed. Valid values: T: table, I: index.<br>This parameter is ignored, but is included for compatibility. |

| Parameter | Description |
|---|---|
| command_type | The type of the analysis function to be run. Valid values:<br>- E: gathers estimated statistics based on a specified number of rows or a percentage of rows in the sample_clause clause.<br>- C: computes exact statistics.<br>- V: validates the structure and integrity of the partitions.<br>This parameter is ignored, but is included for compatibility. |
| command_opt | If command_type is set to C or E, the following combinations are supported:<br>`[ FOR TABLE ]`<br>`[ FOR ALL COLUMNS ]`<br>`[ FOR ALL LOCAL INDEXES ]`<br>If command_type is set to V and object_type is set to T, CASCADE is supported.<br>This parameter is ignored, but is included for compatibility. |
| sample_clause | If command_type is set to E, the following clause is included to specify the number of rows or percentage of rows on which the estimated statistics is based:<br>`SAMPLE n { ROWS \| PERCENT }`<br>This parameter is ignored, but is included for compatibility. |

### CANONICALIZE

The CANONICALIZE procedure supports the following features to manage an input string:

- If the string is not enclosed in double quotation marks, checks whether the string uses the characters of a valid identifier. If not, an error message is returned. If the string is enclosed in double quotation marks, all characters are allowed.

- If the string is not enclosed in double quotation marks and does not contain periods, capitalizes all alphabetic characters and eliminates leading and trailing spaces.
- If the string is enclosed in double quotation marks and does not contain periods, removes the double quotation marks.
- If the string contains periods and no portion of the string is enclosed in double quotation marks, capitalizes each portion of the string and encloses each portion in double quotation marks.
- If the string contains periods and portions of the string are double-quoted, returns the double-quoted portions unchanged including the double quotation marks and returns the non-double-quoted portions capitalized and enclosed in double quotation marks.

```
CANONICALIZE(name VARCHAR2, canon_name OUT VARCHAR2,
  canon_len BINARY_INTEGER)
```

**Parameters**

| Parameter | Description |
|---|---|
| name | The string to be canonicalized. |
| canon_name | The canonicalized string. |
| canon_len | The number of bytes in a name to be canonicalized starting from the first character. |

**Examples**

The following procedure applies the CANONICALIZE procedure on its input parameter and displays the results.

```
CREATE OR REPLACE PROCEDURE canonicalize (
    p_name      VARCHAR2,
    p_length    BINARY_INTEGER DEFAULT 30
)
IS
    v_canon     VARCHAR2(100);
BEGIN
    DBMS_UTILITY.CANONICALIZE(p_name,v_canon,p_length);
    DBMS_OUTPUT.PUT_LINE('Canonicalized name ==>' || v_canon || '<==');
    DBMS_OUTPUT.PUT_LINE('Length: ' || LENGTH(v_canon));
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

EXEC canonicalize('Identifier')
Canonicalized name ==>IDENTIFIER<==
Length: 10
```

```
EXEC canonicalize('"Identifier"')
Canonicalized name ==>Identifier<==
Length: 10

EXEC canonicalize('"_+142%"')
Canonicalized name ==>_+142%<==
Length: 6

EXEC canonicalize('abc.def.ghi')
Canonicalized name ==>"ABC"." DEF"." GHI"<==
Length: 17

EXEC canonicalize('"abc.def.ghi"')
Canonicalized name ==>abc.def.ghi<==
Length: 11

EXEC canonicalize('"abc".def."ghi"')
Canonicalized name ==>"abc"." DEF"."ghi"<==
Length: 17

EXEC canonicalize('"abc.def".ghi')
Canonicalized name ==>"abc.def"." GHI"<==
Length: 15
```

### COMMA_TO_TABLE

You can use the COMMA_TO_TABLE procedure to convert a comma-delimited list of names into a table of names. Each entry in the list is changed into a table entry. The names must be formatted as valid identifiers.

```
COMMA_TO_TABLE(list VARCHAR2, tablen OUT BINARY_INTEGER,
  tab OUT { LNAME_ARRAY | UNCL_ARRAY })
```

#### Parameters

| Parameter | Description |
|---|---|
| list | The comma-delimited list of names from the tab parameter. |
| tablen | The number of entries in a list. |
| tab | The table that contains the listed names. |
| LNAME_ARRAY | DBMS_UTILITY LNAME_ARRAY. For more information, see LNAME_ARRAY. |
| UNCL_ARRAY | DBMS_UTILITY UNCL_ARRAY. For more information, see UNCL_ARRAY. |

#### Examples

The following example shows how the COMMA_TO_TABLE procedure converts a list of names to a table and displays the table entries.

```
CREATE OR REPLACE PROCEDURE comma_to_table (
    p_list      VARCHAR2
)
IS
    r_lname     DBMS_UTILITY.LNAME_ARRAY;
    v_length    BINARY_INTEGER;
BEGIN
    DBMS_UTILITY.COMMA_TO_TABLE(p_list,v_length,r_lname);
    FOR i IN 1..v_length LOOP
        DBMS_OUTPUT.PUT_LINE(r_lname(i));
    END LOOP;
END;

EXEC comma_to_table('edb.dept, edb.emp, edb.jobhist')

edb.dept
edb.emp
edb.jobhist
```

## DB_VERSION

You can use the DB_VERSION procedure to return the version number of the database.

```
DB_VERSION(version OUT VARCHAR2, compatibility OUT VARCHAR2)
```

**Parameters**

| Parameter | Description |
|---|---|
| version | The version of the database. |
| compatibility | The compatibility of the database. The meaning is defined by implementation. |

**Examples**

The following anonymous block displays the database version information.

```
DECLARE
    v_version       VARCHAR2(150);
    v_compat        VARCHAR2(150);
BEGIN
    DBMS_UTILITY.DB_VERSION(v_version,v_compat);
    DBMS_OUTPUT.PUT_LINE('Version: '       || v_version);
    DBMS_OUTPUT.PUT_LINE('Compatibility: ' || v_compat);
END;

Version: EnterpriseDB 10.0.0 on i686-pc-linux-gnu, compiled by GCC gcc (GCC) 4.1.2
20080704 (Red Hat 4.1.2-48), 32-bit
```

```
Compatibility: EnterpriseDB 10.0.0 on i686-pc-linux-gnu, compiled by GCC gcc (GCC) 4.1.
220080704 (Red Hat 4.1.2-48), 32-bit
```

### EXEC_DDL_STATEMENT

You can use the EXEC_DDL_STATEMENT procedure to run a DDL command.

```
EXEC_DDL_STATEMENT(parse_string VARCHAR2)
```

**Parameters**

| Parameter | Description |
|---|---|
| parse_string | The DDL command to be run. |

**Examples**

The following anonymous block creates the job table.

```
BEGIN
    DBMS_UTILITY.EXEC_DDL_STATEMENT(
        'CREATE TABLE job (' ‖
          'jobno NUMBER(3),' ‖
          'jname VARCHAR2(9))'
    );
END;
```

If the parse_string does not include a valid DDL statement, the following error message is returned:

```
#  exec dbms_utility.exec_ddl_statement('select rownum from dual');
ERROR:  EDB-20001: 'parse_string' must be a valid DDL statement
```

In this case, the behavior of PolarDB databases compatible with Oracle differs from that of Oracle. Oracle supports the invalid parse_string and no error message is returned.

### FORMAT_CALL_STACK

You can use the FORMAT_CALL_STACK function to return the formatted contents of the current call stack.

```
DBMS_UTILITY.FORMAT_CALL_STACKreturn VARCHAR2
```

This function can be used in a stored procedure, function, or package to return the current call stack in a readable format. This function is helpful in debugging.

### GET_CPU_TIME

You can use the GET_CPU_TIME function to return the CPU time in hundredths of a second from some arbitrary point in time.

### Parameters

| Parameter | Description |
|---|---|
| cputime | The number of hundredths of a second of CPU time. |

### Examples

The following SELECT command retrieves the current CPU time, which is 603 hundredths of a second or 0.0603 seconds.

```
SELECT DBMS_UTILITY.GET_CPU_TIME FROM DUAL;

get_cpu_time
--------------
        603
```

## GET_DEPENDENCY

You can use the GET_DEPENDENCY procedure to list the objects that are dependent on the specified object. The procedure does not show dependencies for functions or procedures.

```
GET_DEPENDENCY(type VARCHAR2, schema VARCHAR2,
  name VARCHAR2)
```

### Parameters

| Parameter | Description |
|---|---|
| type | The type of the name object. Valid values: INDEX, PACKAGE, PACKAGE BODY, SEQUENCE , TABLE, TRIGGER, TYPE, and VIEW. |
| schema | The name of the schema in which the name object exists. |
| name | The name of the object for which dependencies are to be retrieved. |

### Examples

The following anonymous block retrieves dependencies on the EMP table.

```
BEGIN
    DBMS_UTILITY.GET_DEPENDENCY('TABLE','public','EMP');
END;

DEPENDENCIES ON public.EMP
------------------------------------------------------------------
*TABLE public.EMP()
*   CONSTRAINT c public.emp()
*   CONSTRAINT f public.emp()
*   CONSTRAINT p public.emp()
*   TYPE public.emp()
*   CONSTRAINT c public.emp()
*   CONSTRAINT f public.jobhist()
*   VIEW .empname_view()
```

## GET_HASH_VALUE

You can use the GET_HASH_VALUE function to compute a hash value for a specified string.

```
hash NUMBER GET_HASH_VALUE(name VARCHAR2, base NUMBER,
  hash_size NUMBER)
```

### Parameters

| Parameter | Description |
|---|---|
| name | The string for which a hash value is computed. |
| base | The value starting from which hash values are generated. |
| hash_size | The number of hash values for the expected hash table. |
| hash | The hash value that is generated. |

### Examples

The following anonymous block creates a table of hash values by using the ename column of the emp table and then displays the key along with the hash value. The hash values start from 100 and include a maximum of 1,024 distinct values.

```
DECLARE
    v_hash          NUMBER;
    TYPE hash_tab IS TABLE OF NUMBER INDEX BY VARCHAR2(10);
    r_hash          HASH_TAB;
    CURSOR emp_cur IS SELECT ename FROM emp;
BEGIN
    FOR r_emp IN emp_cur LOOP
        r_hash(r_emp.ename) :=
            DBMS_UTILITY.GET_HASH_VALUE(r_emp.ename,100,1024);
    END LOOP;
    FOR r_emp IN emp_cur LOOP
        DBMS_OUTPUT.PUT_LINE(RPAD(r_emp.ename,10) || ' ' ||
            r_hash(r_emp.ename));
    END LOOP;
END;

SMITH      377
ALLEN      740
WARD       718
JONES      131
MARTIN     176
BLAKE      568
```

```
CLARK     621
SCOTT     1097
KING      235
TURNER    850
ADAMS     156
JAMES     942
FORD      775
MILLER    148
```

### GET_PARAMETER_VALUE

You can use the GET_PARAMETER_VALUE procedure to retrieve database initialization parameter settings.

```
status BINARY_INTEGER GET_PARAMETER_VALUE(parnam VARCHAR2,
  intval OUT INTEGER, strval OUT VARCHAR2)
```

**Parameters**

| Parameter | Description |
|---|---|
| parnam | The name of the parameter whose value is returned. The parameters are listed in the pg_settings system view. |
| intval | The value of an integer parameter or the length of the strval parameter. |
| strval | The value of a string parameter. |
| status | Returns 0 if the parameter value is INTEGER or BOOLEAN. Returns 1 if the parameter value is a string. |

**Examples**

The following anonymous block shows the values of two initialization parameters.

```
DECLARE
    v_intval       INTEGER;
    v_strval       VARCHAR2(80);
BEGIN
    DBMS_UTILITY.GET_PARAMETER_VALUE('max_fsm_pages', v_intval, v_strval);
    DBMS_OUTPUT.PUT_LINE('max_fsm_pages' ‖ ': ' ‖ v_intval);
    DBMS_UTILITY.GET_PARAMETER_VALUE('client_encoding', v_intval, v_strval);
    DBMS_OUTPUT.PUT_LINE('client_encoding' ‖ ': ' ‖ v_strval);
END;

max_fsm_pages: 72625
client_encoding: SQL_ASCII
```

### GET_TIME

You can use the GET_TIME function to return the current time in hundredths of a second.

#### Parameters

| Parameter | Description |
|---|---|
| time | The number of hundredths of a second elapsed since the program is started. |

#### Examples

The following example shows the calls to the GET_TIME function.

```
SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
----------
  1555860

SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
----------
  1556037
```

### NAME_TOKENIZE

You can use the NAME_TOKENIZE procedure to parse a name into its component parts . Names that are not enclosed in double quotation marks are capitalized. The double quotation marks are removed from names with double quotation marks.

```
NAME_TOKENIZE(name VARCHAR2, a OUT VARCHAR2,   b OUT VARCHAR2,c OUT VARCHAR2
, dblink OUT VARCHAR2,   nextpos OUT BINARY_INTEGER)
```

#### Parameters

| Parameter | Description |
|---|---|
| name | The string that contains a name in the following format:<br>`a[.b[.c]][@dblink ]` |
| a | Returns the leftmost component. |
| b | Returns the second component if the component exists. |
| c | Returns the third component if the component exists. |
| dblink | Returns the database link name. |
| nextpos | Position of the last character parsed in the name. |

**Examples**

The following stored procedure is used to display the returned parameter values of the NAME_TOKENIZE procedure for various names.

```
CREATE OR REPLACE PROCEDURE name_tokenize (
    p_name          VARCHAR2
)
IS
    v_a             VARCHAR2(30);
    v_b             VARCHAR2(30);
    v_c             VARCHAR2(30);
    v_dblink        VARCHAR2(30);
    v_nextpos       BINARY_INTEGER;
BEGIN
    DBMS_UTILITY.NAME_TOKENIZE(p_name,v_a,v_b,v_c,v_dblink,v_nextpos);
    DBMS_OUTPUT.PUT_LINE('name   : ' || p_name);
    DBMS_OUTPUT.PUT_LINE('a      : ' || v_a);
    DBMS_OUTPUT.PUT_LINE('b      : ' || v_b);
    DBMS_OUTPUT.PUT_LINE('c      : ' || v_c);
    DBMS_OUTPUT.PUT_LINE('dblink : ' || v_dblink);
    DBMS_OUTPUT.PUT_LINE('nextpos: ' || v_nextpos);
END;
```

Tokenize the name parameter set to emp:

```
BEGIN
    name_tokenize('emp');
END;

name   : emp
a      : EMP
b      :
c      :
dblink :
nextpos: 3
```

Tokenize the name parameter set to edb.list_emp:

```
BEGIN
    name_tokenize('edb.list_emp');
END;

name   : edb.list_emp
a      : EDB
b      : LIST_EMP
c      :
dblink :
nextpos: 12
```

Tokenize the name parameter set to "edb"."Emp_Admin".update_emp_sal:

```
BEGIN
    name_tokenize('"edb"." Emp_Admin".update_emp_sal');
END;

name   : "edb"." Emp_Admin".update_emp_sal
a      : edb
b      : Emp_Admin
```

```
c     : UPDATE_EMP_SAL
dblink :
nextpos: 32
```

Tokenize the name parameter set to edb.emp@edb_dblink:

```
BEGIN
    name_tokenize('edb.emp@edb_dblink');
END;

name   : edb.emp@edb_dblink
a      : EDB
b      : EMP
c      :
dblink : EDB_DBLINK
nextpos: 18
```

### TABLE_TO_COMMA

You can use the TABLE_TO_COMMA procedure to convert a table of names into a comma-delimited list of names. Each table entry is changed into a list entry. The names must be formatted as valid identifiers.

```
TABLE_TO_COMMA(tab { LNAME_ARRAY | UNCL_ARRAY },
  tablen OUT BINARY_INTEGER, list OUT VARCHAR2)
```

#### Parameters

| Parameter | Description |
|---|---|
| tab | The table that contains names. |
| LNAME_ARRAY | DBMS_UTILITY LNAME_ARRAY. For more information, see LNAME_ARRAY. |
| UNCL_ARRAY | DBMS_UTILITY UNCL_ARRAY. For more information, see UNCL_ARRAY. |
| tablen | The number of entries in the list. |
| list | The comma-delimited list of names specified by the tab parameter. |

#### Examples

The following example shows how the COMMA_TO_TABLE procedure converts a comma-delimited list to a table and how the TABLE_TO_COMMA procedure then converts the table back to a comma-delimited list and displays the list.

```
CREATE OR REPLACE PROCEDURE table_to_comma (
    p_list      VARCHAR2
)
IS
    r_lname     DBMS_UTILITY.LNAME_ARRAY;
    v_length    BINARY_INTEGER;
    v_listlen   BINARY_INTEGER;
    v_list      VARCHAR2(80);
BEGIN
    DBMS_UTILITY.COMMA_TO_TABLE(p_list,v_length,r_lname);
    DBMS_OUTPUT.PUT_LINE('Table Entries');
    DBMS_OUTPUT.PUT_LINE('-------------');
    FOR i IN 1..v_length LOOP
        DBMS_OUTPUT.PUT_LINE(r_lname(i));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('-------------');
    DBMS_UTILITY.TABLE_TO_COMMA(r_lname,v_listlen,v_list);
    DBMS_OUTPUT.PUT_LINE('Comma-Delimited List: ' || v_list);
END;

EXEC table_to_comma('edb.dept, edb.emp, edb.jobhist')

Table Entries
-------------
edb.dept
edb.emp
edb.jobhist
-------------
Comma-Delimited List: edb.dept, edb.emp, edb.jobhist
```

# 17.17 UTL_ENCODE

The UTL_ENCODE package provides the capability to encode and decode data.

**Table 17-28: UTL_ENCODE functions and stored procedures**

| Function/stored procedure | Return type | Description |
|---|---|---|
| BASE64_DECODE(r) | RAW | Translates a Base64 encoded string to the original RAW value. |
| BASE64_ENCODE(r) | RAW | Translates a RAW string to an encoded Base64 value. |
| BASE64_ENCODE(loid) | TEXT | Translates a TEXT string to an encoded Base64 value. |
| MIMEHEADER_DECODE(buf) | VARCHAR2 | Translates an encoded MIMEHEADER formatted string to its original value. |
| MIMEHEADER_ENCODE(buf, encode_charset, encoding) | VARCHAR2 | Converts and encodes a string in MIMEHEADER format. |
| QUOTED_PRINTABLE_DECODE (r) | RAW | Translates an encoded string to a RAW value. |

| Function/stored procedure | Return type | Description |
|---|---|---|
| QUOTED_PRINTABLE_ENCODE(r) | RAW | Translates an input string to a quoted-printable formatted RAW value. |
| TEXT_DECODE(buf, encode_charset, encoding) | VARCHAR2 | Decodes a string encoded by TEXT_ENCODE. |
| TEXT_ENCODE(buf, encode_charset, encoding) | VARCHAR2 | Translates a string to a user-specified character set, and then encode the string. |
| UUDECODE(r) | RAW | Translates a uuencode encoded string to a RAW value. |
| UUENCODE(r, type, filename , permission) | RAW | Translates a RAW string to an encoded uuencode value. |

### BASE64_DECODE

Converts a Base64 encoded string into the original value that is encoded by the BASE64_ENCODE function. Syntax:

```
BASE64_DECODE(r IN RAW)
```

This function returns a RAW value.

#### Parameters

| Parameter | Description |
|---|---|
| r | The r parameter is the string that contains the Base64 encoded data that will be converted into a RAW value. |

#### Examples

**Note:**

Before using this example, you must run the following command:

```
SET bytea_output = escape;
```

This command instructs the server to escape non-printable characters and display BYTEA or RAW values in readable form. For more information, see the Postgres Core Documentation available at: http://www.enterprisedb.com/docs/en/9.3/pg/datatype-binary.html

The following example uses the BASE64_ENCODE function to encode a string that contains the text abc and then uses the BASE64_DECODE function to decode the string:

```
edb=# SELECT UTL_ENCODE.BASE64_ENCODE(CAST ('abc' AS RAW));
 base64_encode
---------------
 YWJj
(1 row)

edb=# SELECT UTL_ENCODE.BASE64_DECODE(CAST ('YWJj' AS RAW));
 base64_decode
---------------
 abc
(1 row)
```

### BASE64_ENCODE

The BASE64_ENCODE function converts and encodes a string in Base64 format, as described in RFC 4648. This function is useful for composing MIME emails that you intend to send using the UTL_SMTP package. The BASE64_ENCODE function has two syntaxes:

```
BASE64_ENCODE(r IN RAW)
```

And

```
BASE64_ENCODE(loid IN OID)
```

This function returns a RAW value or an OID.

**Parameters**

| Parameter | Description |
|---|---|
| r | The r parameter specifies the RAW string that will be converted into Base64. |
| loid | The loid parameter specifies the ID of a large object that will be converted into Base64. |

**Examples**

**Note:**

Before using this example, you must run the following command:

```
SET bytea_output = escape;
```

This command instructs the server to escape non-printable characters and display BYTEA or RAW values in readable form. For more information, see the Postgres Core Documentation available at: http://www.enterprisedb.com/docs/en/9.3/pg/datatype-binary.html

The following example uses the BASE64_ENCODE function to encode a string that contains the text abc and then uses the BASE64_DECODE function to decode the string:

```
edb=# SELECT UTL_ENCODE.BASE64_ENCODE(CAST ('abc' AS RAW));
 base64_encode
---------------
 YWJj
(1 row)

edb=# SELECT UTL_ENCODE.BASE64_DECODE(CAST ('YWJj' AS RAW));
 base64_decode
---------------
 abc
(1 row)
```

**MIMEHEADER_DECODE**

The MIMEHEADER_DECODE function decodes values that are encoded by the MIMEHEADER_ENCODE function. Syntax:

```
MIMEHEADER_DECODE(buf IN VARCHAR2)
```

This function returns a VARCHAR2 value.

**Parameters**

| Parameter | Description |
|---|---|
| buf | The buf parameter contains the value ( encoded by the MIMEHEADER_ENCODE function) that will be decoded. |

**Examples**

The following example uses the MIMEHEADER_ENCODE function to encode a string and then uses the MIMEHEADER_DECODE function to decode the string:

```
edb=# SELECT UTL_ENCODE.MIMEHEADER_ENCODE('What is the date?') FROM DUAL;
     mimeheader_encode
------------------------------
 =? UTF8? Q? What is the date?? =
(1 row)
```

```
edb=# SELECT UTL_ENCODE.MIMEHEADER_DECODE('=? UTF8? Q? What is the date?? =')
FROM DUAL;
 mimeheader_decode
-------------------
 What is the date?
(1 row)
```

## MIMEHEADER_ENCODE

The MIMEHEADER_ENCODE function converts a string into mime header format, and then encodes the string. Syntax:

```
MIMEHEADER_ENCODE(buf IN VARCHAR2, encode_charset IN VARCHAR2 DEFAULT NULL,
encoding IN INTEGER DEFAULT NULL)
```

This function returns a VARCHAR2 value.

### Parameters

| Parameter | Description |
|---|---|
| buf | The buf parameter contains the string that will be formatted and encoded. The string is a VARCHAR2 value. |
| encode_charset | The encode_charset parameter specifies the character set into which the string will be converted before being formatted and encoded. Default value: NULL. |
| encoding | The encoding parameter specifies the encoding type used when encoding the string. You can specify one of the following two encoding types:<br>• Specify the Q encoding type to enable quoted-printable encoding. If you do not specify a value, the MIMEHEADER_ENCODE function will use quoted-printable encoding.<br>• Specify the B encoding type to enable base-64 encoding. |

### Examples

The following example uses the MIMEHEADER_ENCODE function to encode a string and then uses the MIMEHEADER_DECODE function to decode the string:

```
edb=# SELECT UTL_ENCODE.MIMEHEADER_ENCODE('What is the date?') FROM DUAL;
     mimeheader_encode
------------------------------
```

```
 =? UTF8? Q? What is the date?? =
(1 row)

edb=# SELECT UTL_ENCODE.MIMEHEADER_DECODE('=? UTF8? Q? What is the date?? =')
FROM DUAL;
 mimeheader_decode
-------------------
 What is the date?
(1 row)
```

### QUOTED_PRINTABLE_DECODE

The QUOTED_PRINTABLE_DECODE function converts an encoded quoted-printable string into a decoded RAW string. Syntax:

```
QUOTED_PRINTABLE_DECODE(r IN RAW)
```

This function returns a RAW value.

#### Parameters

| Parameter | Description |
|---|---|
| r | The r parameter contains the encoded string that will be decoded. The string is a RAW value that is encoded by the QUOTED_PRINTABLE_ENCODE function. |

#### Examples

**Note:**

Before using this example, you must run the following command:

```
SET bytea_output = escape;
```

This command instructs the server to escape non-printable characters and display BYTEA or RAW values in readable form. For more information, see the Postgres Core Documentation available at: http://www.enterprisedb.com/docs/en/9.3/pg/datatype-binary.html.

The following example first encodes and then decodes a string:

```
edb=# SELECT UTL_ENCODE.QUOTED_PRINTABLE_ENCODE('E=mc2') FROM DUAL;
quoted_printable_encode
-------------------------
 E=3Dmc2
(1 row)

edb=# SELECT UTL_ENCODE.QUOTED_PRINTABLE_DECODE('E=3Dmc2') FROM DUAL;
 quoted_printable_decode
-------------------------
 E=mc2
```

```
(1 row)
```

### QUOTED_PRINTABLE_ENCODE

The QUOTED_PRINTABLE_ENCODE function converts and encodes a string into quoted-printable format. Syntax:

```
QUOTED_PRINTABLE_ENCODE(r IN RAW)
```

This function returns a RAW value.

#### Parameters

| Parameter | Description |
|---|---|
| r | The r parameter contains the string (a RAW value) that will be encoded in a quoted-printable format. |

#### Examples

**Note:**

Before using this example, you must run the following command:

```
SET bytea_output = escape;
```

This command instructs the server to escape non-printable characters and display BYTEA or RAW values in readable form. For more information, see the Postgres Core Documentation available at: http://www.enterprisedb.com/docs/en/9.3/pg/datatype-binary.html.

The following example first encodes and then decodes a string:

```
edb=# SELECT UTL_ENCODE.QUOTED_PRINTABLE_ENCODE('E=mc2') FROM DUAL;
quoted_printable_encode
-------------------------
 E=3Dmc2
(1 row)

edb=# SELECT UTL_ENCODE.QUOTED_PRINTABLE_DECODE('E=3Dmc2') FROM DUAL;
 quoted_printable_decode
-------------------------
 E=mc2
```

```
(1 row)
```

### TEXT_DECODE

The TEXT_DECODE function converts and decodes an encoded string into the VARCHAR2 value that was originally encoded by the TEXT_ENCODE function. Syntax:

```
TEXT_DECODE(buf IN VARCHAR2, encode_charset IN VARCHAR2 DEFAULT NULL, encoding
IN PLS_INTEGER DEFAULT NULL)
```

This function returns a VARCHAR2 value.

**Parameters**

| Parameter | Description |
|---|---|
| buf | The buf parameter contains the encoded string that will be converted into the original value encoded by the TEXT_ENCODE function. |
| encode_charset | The encode_charset parameter specifies the character set into which the string will be converted before encoding. Default value: NULL. |
| encoding | The encoding parameter specifies the encoding type used by the TEXT_DECODE function. You can specify one of the following two encoding types:<br>• UTL_ENCODE.BASE64 specifies the Base64 encoding.<br>• UTL_ENCODE.QUOTED_PRINTABLE specifies the quoted printable encoding. This is the default encoding type. |

**Examples**

The following example uses the TEXT_ENCODE function to encode a string and then uses the TEXT_DECODE function to decode the string:

```
edb=# SELECT UTL_ENCODE.TEXT_ENCODE('What is the date?', 'BIG5', UTL_ENCODE.
BASE64) FROM DUAL;
      text_encode
--------------------------
 V2hhdCBpcyB0aGUgZGF0ZT8=
(1 row)

edb=# SELECT UTL_ENCODE.TEXT_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5',
UTL_ENCODE.BASE64) FROM DUAL;
```

```
   text_decode
-------------------
 What is the date?
(1 row)
```

### TEXT_ENCODE

The TEXT_ENCODE function converts a string into a specified character set, and then encodes the string. Syntax:

```
TEXT_DECODE(buf IN VARCHAR2, encode_charset IN VARCHAR2 DEFAULT NULL, encoding
IN PLS_INTEGER DEFAULT NULL)
```

This function returns a VARCHAR2 value.

**Parameters**

| Parameter | Description |
|---|---|
| buf | The buf parameter contains the encoded string that will be converted into the specified character set and encoded by the TEXT_ENCODE function. |
| encode_charset | The encode_charset parameter specifies the character set into which the value will be converted before encoding. Default value: NULL. |
| encoding | The encoding parameter specifies the encoding type used by the TEXT_ENCODE function. You can specify one of the following two encoding types:<br>• UTL_ENCODE.BASE64 specifies the Base64 encoding.<br>• UTL_ENCODE.QUOTED_PRINTABLE specifies the quoted printable encoding. This is the default encoding type. |

**Examples**

The following example uses the TEXT_ENCODE function to encode a string and then uses the TEXT_DECODE function to decode the string:

```
edb=# SELECT UTL_ENCODE.TEXT_ENCODE('What is the date?', 'BIG5', UTL_ENCODE.
BASE64) FROM DUAL;
      text_encode
--------------------------
 V2hhdCBpcyB0aGUgZGF0ZT8=
(1 row)
```

```
edb=# SELECT UTL_ENCODE.TEXT_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5',
UTL_ENCODE.BASE64) FROM DUAL;
   text_decode
-------------------
 What is the date?
(1 row)
```

## UUDECODE

The UUDECODE function converts and decodes a uuencode encoded string into the RAW value that was originally encoded by the UUENCODE function. Syntax:

```
UUDECODE(r IN RAW)
```

This function returns a RAW value.

### Parameter

| Parameter | Description |
|---|---|
| r | The r parameter contains the uuencoded string that will be converted into a RAW value. |

### Examples

**Note:**

Before using this example, you must run the following command:

```
SET bytea_output = escape;
```

This command instructs the server to escape non-printable characters and display BYTEA or RAW values in readable form. For more information, see the Postgres Core Documentation available at: http://www.enterprisedb.com/docs/en/9.3/pg/datatype-binary.html

The following example uses the UUENCODE function to encode a string and then uses the UUDECODE function to decode the string:

```
edb=# SET bytea_output = escape;
SET
edb=# SELECT UTL_ENCODE.UUENCODE('What is the date?') FROM DUAL;
                  uuencode
--------------------------------------------------------------------
 begin 0 uuencode.txt\01215VAA="! I<R! T:&4@9&%T93\\`\012`\012end\012
(1 row)

edb=# SELECT UTL_ENCODE.UUDECODE
edb-# ('begin 0 uuencode.txt\01215VAA="! I<R! T:&4@9&%T93\\`\012`\012end\012')
edb-# FROM DUAL;
    uudecode
-------------------
```

```
 What is the date?
(1 row)
```

### UUENCODE

The UUENCODE function converts RAW data into a uuencode formatted encoded string.

Syntax:

```
UUENCODE(r IN RAW, type IN INTEGER DEFAULT 1, filename IN VARCHAR2 DEFAULT NULL,
permission IN VARCHAR2 DEFAULT NULL)
```

This function returns a RAW value.

#### Parameters

| Parameter | Description |
|---|---|
| r | The r parameter contains the RAW string that will be converted into uuencode format . |
| type | The type parameter is an INTEGER value or constant. This constant specifies the type of uuencoded string that will be returned. Default value: 1. Table 17-29: The type parameter lists the valid values. |
| filename | The filename parameter is a VARCHAR2 value that specifies the file name that you want to embed in the encoded form . If you do not specify a file name, the UUENCODE function will include a filename of uuencode.txt in the encoded form. |
| permission | The permission parameter is a VARCHAR2 value that specifies the permission mode. Default value: NULL. |

Table 17-29: The type parameter

| Value | Constant |
|---|---|
| 1 | complete |
| 2 | header_piece |
| 3 | middle_piece |
| 4 | end_piece |

#### Examples

**Note:**

Before using this example, you must run the following command:

```
SET bytea_output = escape;
```

This command instructs the server to escape non-printable characters and display BYTEA or RAW values in readable form. For more information, see the Postgres Core Documentation available at: http://www.enterprisedb.com/docs/en/9.3/pg/datatype-binary.html

The following example uses the UUENCODE function to encode a string and then uses the UUDECODE function to decode the string:

```
edb=# SET bytea_output = escape;
SET
edb=# SELECT UTL_ENCODE.UUENCODE('What is the date?') FROM DUAL;
                          uuencode
--------------------------------------------------------------------
 begin 0 uuencode.txt\01215VAA="! I<R! T:&4@9&%T93\\`\012`\012end\012
(1 row)

edb=# SELECT UTL_ENCODE.UUDECODE
edb-# ('begin 0 uuencode.txt\01215VAA="! I<R! T:&4@9&%T93\\`\012`\012end\012')
edb-# FROM DUAL;
    uudecode
-------------------
 What is the date?
(1 row)
```

## 17.18 UTL_RAW

The UTL_RAW package allows you to manipulate or retrieve the length of raw data types.

**Note:**

An administrator must grant execute permissions to each user or group before they can use this package.

| Function/Procedure | Category | Return type | Description |
|---|---|---|---|
| CAST_TO_RAW(c IN VARCHAR2) | Function | RAW | Converts a VARCHAR2 string to a RAW value. |
| CAST_TO_VARCHAR2(r IN RAW) | Function | VARCHAR2 | Converts a RAW value to a VARCHAR2 string. |

| Function/Procedure | Category | Return type | Description |
|---|---|---|---|
| CONCAT(r1 IN RAW, r2 IN RAW, r3 IN RAW ,...) | Function | RAW | Concatenates multiple RAW values into a single RAW value. |
| CONVERT(r IN RAW, to_charset IN VARCHAR2, from_charset IN VARCHAR2 | Function | RAW | Converts encoded data from one encoding format to another encoding format, and returns the result as a RAW value. |
| LENGTH(r IN RAW) | Function | NUMBER | Returns the length of a RAW value. |
| SUBSTR(r IN RAW, pos IN INTEGER, len IN INTEGER) | Function | RAW | Returns a portion of a RAW value. |

The implementation of UTL_RAW in PolarDB databases compatible with Oracle is a partial implementation when compared with native Oracle. Only those functions and procedures listed in the preceding table are supported.

## CAST_TO_RAW

You can use the CAST_TO_RAW function to convert a VARCHAR2 string to a RAW value. The function has the following signature:

```
CAST_TO_RAW(c VARCHAR2)
```

The function returns a RAW value if you pass a non-NULL value. If you pass a NULL value, the function returns NULL.

### Parameters

| Parameter | Description |
|---|---|
| c | The VARCHAR2 value that is converted to RAW. |

### Examples

The following example shows how the CAST_TO_RAW function converts a VARCHAR2 string to a RAW value:

```
DECLARE
  v VARCHAR2;
  r RAW;
BEGIN
  v := 'Accounts';
  dbms_output.put_line(v);
  r := UTL_RAW.CAST_TO_RAW(v);
  dbms_output.put_line(r);
END;
```

The result set includes the content of the original string and the converted RAW value.

```
Accounts
\x4163636f756e7473
```

### CAST_TO_VARCHAR2

You can use the CAST_TO_VARCHAR2 function to convert RAW data to VARCHAR2 data. The function has the following signature:

```
CAST_TO_VARCHAR2(r RAW)
```

The function returns a VARCHAR2 value if you pass a non-NULL value. If you pass a NULL value, the function returns NULL.

#### Parameters

| Parameter | Description |
|---|---|
| r | The RAW value that is converted to a VARCHAR2 value. |

#### Examples

The following example shows how the CAST_TO_VARCHAR2 function converts a RAW value to a VARCHAR2 string:

```
DECLARE
  r RAW;
  v VARCHAR2;
BEGIN
  r := '\x4163636f756e7473'
  dbms_output.put_line(v);
  v := UTL_RAW.CAST_TO_VARCHAR2(r);
  dbms_output.put_line(r);
```

```
END;
```

The result set includes the content of the original string and the converted RAW value.

```
\x4163636f756e7473
Accounts
```

### CONCAT

You can use the CONCAT function to concatenate multiple RAW values into a single RAW value. The function has the following signature:

```
CONCAT(r1 RAW, r2 RAW, r3 RAW,...)
```

The function returns a RAW value. Different from the Oracle implementation, the implementation of PolarDB databases compatible with Oracles is a variadic function, and does not limit the number of values that can be concatenated.

**Parameters**

| Parameter | Description |
|---|---|
| r1, r2, r3,... | The RAW values that CONCAT concatenates. |

**Examples**

The following example shows how the CONCAT function concatenates multiple RAW values into a single RAW value:

```
SELECT UTL_RAW.CAST_TO_VARCHAR2(UTL_RAW.CONCAT('\x61', '\x62', '\x63')) FROM
DUAL;  concat
--------  abc(1 row)
```

The concatenated values as the result is then converted to the VARCHAR2 format by the CAST_TO_VARCHAR2 function.

### CONVERT

You can use the CONVERT function to convert a string from one encoding format to another encoding format and returns the result as a RAW value. The function has the following signature:

```
CONVERT(r RAW, to_charset VARCHAR2, from_charset VARCHAR2)
```

The function returns a RAW value.

**Parameters**

| Parameter | Description |
|---|---|
| r | The RAW value that is converted. |
| to_charset | The name of the encoding format to which r is converted. |
| from_charset | The name of the encoding format from which r is converted. |

**Examples**

The following example shows how the UTL_RAW.CAST_TO_RAW function converts the VARCHAR2 string Accounts to a raw value, converts the raw value from UTF8 to LATIN7, and then converts the value from LATIN7 to UTF8:

```
DECLARE   r RAW;   v VARCHAR2;BEGIN   v:= 'Accounts';   dbms_output.put_line(v);
  r:= UTL_RAW.CAST_TO_RAW(v);   dbms_output.put_line(r);   r:= UTL_RAW.CONVERT(r, '
UTF8', 'LATIN7');   dbms_output.put_line(r);   r:= UTL_RAW.CONVERT(r, 'LATIN7', 'UTF8');
dbms_output.put_line(r);
```

The example returns the VARCHAR2 value, the RAW value, and the converted values.

```
Accounts
\x4163636f756e7473
\x4163636f756e7473
\x4163636f756e7473
```

### LENGTH

You can use the LENGTH function to return the length of a RAW value. The function has the following signature:

```
LENGTH(r RAW)
```

The function returns a RAW value.

**Parameters**

| Parameter | Description |
|---|---|
| r | The RAW value that LENGTH evaluates. |

**Examples**

The following example shows how the LENGTH function returns the length of a RAW value:

```
SELECT UTL_RAW.LENGTH(UTL_RAW.CAST_TO_RAW('Accounts')) FROM DUAL;  length
```

```
--------8(1 row)
```

The following example uses the LENGTH function to return the length of a RAW value that includes multi-byte characters:

```
SELECT UTL_RAW.LENGTH(UTL_RAW.CAST_TO_RAW('独孤求败'));
 length
--------
     12
(1 row)
```

### SUBSTR

You can use the SUBSTR function to return a substring of a RAW value. The function has the following signature:

```
SUBSTR (r RAW, pos INTEGER, len INTEGER)
```

The function returns a RAW value.

**Parameters**

| Parameter | Description |
|---|---|
| r | The RAW value from which the substring is returned. |
| pos | The position within the RAW value where the first byte of the returned substring is located.<br>• If pos is set to 0 or 1, the substring begins at the first byte of the RAW value.<br>• If pos is greater than one, the substring begins at the first byte specified by pos. For example, if pos is set to 3, the substring begins at the third byte of the value.<br>• If pos is negative, the substring covers a length of pos bytes from the end of the source value. For example, if pos is set to -3, the substring begins at the third byte from the end of the value. |
| len | The maximum number of bytes that are returned. |

**Examples**

The following example shows how the SUBSTR function retrieves a 3-byte substring that starts from the beginning of a RAW value:

```
SELECT UTL_RAW.SUBSTR(UTL_RAW.CAST_TO_RAW('Accounts'), 3, 5) FROM DUAL;
 substr--------  count(1 row)
```

The following example shows how the SUBSTR function retrieves a 5-byte substring that starts from the end of a RAW value:

```
SELECT UTL_RAW.SUBSTR(UTL_RAW.CAST_TO_RAW('Accounts'), -5 , 3) FROM DUAL;
 substr
--------
 oun
(1 row)
```

## 17.19 UTL_URL

The UTL_URL package provides a method to escape invalid and reserved characters within a URL.

**Table 17-30: UTL_URL functions and procedures**

| Function/stored procedure | Return type | Description |
|---|---|---|
| ESCAPE(url, escape reserved chars, url_charset) | VARCHAR2 | Escapes any invalid and reserved characters in a URL. |
| UNESCAPE(url, url charset) | VARCHAR2 | Converts a URL to its original form. |

If the call to a function includes an invalid URL, the UTL_URL package will return the BAD_URL exception.

### ESCAPE

The ESCAPE function escapes invalid and reserved characters within a URL. Syntax:

```
ESCAPE(url VARCHAR2, escape_reserved_chars BOOLEAN, url_charset VARCHAR2)
```

Reserved characters are replaced with a percent sign (%), followed by the two-digit hexadecimal code of the ASCII value for the escaped character.

**Parameters**

| Parameter | Description |
|---|---|
| url | **url** specifies the Uniform Resource Locator (URL) that UTL_URL will escape. |

| Parameter | Description |
|---|---|
| escape_reserved_chars | **escape_reserved_chars** is a BOOLEAN value that instructs the ESCAPE function to escape reserved and invalid characters.<br>• If **escaped_reserved_chars** is set to FALSE, the ESCAPE function will only escape the invalid characters in the specified URL.<br>• If **escape_reserved_chars** is set to TRUE, the ESCAPE function will escape both the invalid characters and the reserved characters in the specified URL. By default, **escape_reserved_chars** is set to FALSE.<br>For more information about valid characters within a URL, see Table 17-31: Valid characters.<br>Some characters are valid in some parts of a URL, while invalid in others. For more information about rules related to invalid characters, see RFC 2396. For more information about examples of characters that are considered to be invalid in any part of a URL, see Table 17-32: Invalid characters.<br>For more information about characters that are considered to be reserved by the ESCAPE function, see Table 17-33: Reserved characters. If **escape_reserved_chars** is set to TRUE, the ESCAPE function will escape the reserved characters. |
| url_charset | **url_charset** specifies a character set to which a given character will be converted before it is escaped. If **url_charset** is NULL, the character will not be converted. The default value of **url_charset** is ISO-8859-1. |

**Table 17-31: Valid characters**

| | | |
|---|---|---|
| Uppercase letters A through Z | Lowercase letters a through z | Digits 0 through 9 |
| Asterisk (*) | Exclamation point (!) | Hyphen (-) |
| Opening parenthesis (() | Period (.) | Closing parenthesis ()) |
| Single-quote (') | Tilde (~) | Underscore (_) |

**Table 17-32: Invalid characters**

| Invalid character | Escape sequence |
|---|---|
| Space ( ) | %20 |
| Curly braces ({ or }) | %7b and %7d |
| Hash mark (#) | %23 |

**Table 17-33: Reserved characters**

| Reserved character | Escape sequence |
|---|---|
| Ampersand (&) | %5C |
| At sign (@) | %25 |
| Colon (:) | %3a |
| Comma (,) | %2c |
| Dollar sign ($) | %24 |
| Equal sign (=) | %3d |
| Plus sign (+) | %2b |
| Question mark (?) | %3f |
| Semicolon (;) | %3b |
| Slash (/) | %2f |

### Examples

The following anonymous block uses the ESCAPE function to escape the spaces in the URL:

```
DECLARE
  result varchar2(400);
BEGIN
 result := UTL_URL.ESCAPE('http://www.example.com/Using the ESCAPE function.html');
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

The escaped URL is:

```
http://www.example.com/Using%20the%20ESCAPE%20function.html
```

If you include a value of TRUE for the escape_reserved_chars parameter when calling the function:

```
DECLARE
  result varchar2(400);
BEGIN
```

```
 result := UTL_URL.ESCAPE('http://www.example.com/Using the ESCAPE function.html',
TRUE);
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

The ESCAPE function escapes the reserved characters and the invalid characters in the URL:

```
http%3A%2F%2Fwww.example.com%2FUsing%20the%20ESCAPE%20function.html
```

### UNESCAPE

The UNESCAPE function removes escape characters added to a URL by the ESCAPE function, converting the URL to its original form. Syntax:

```
UNESCAPE(url VARCHAR2, url_charset VARCHAR2)
```

#### Parameters

| Parameter | Description |
|---|---|
| url | url specifies the Uniform Resource Locator ( URL) that UTL_URL will unescape. |
| url_charset | After a character is unescaped, the character is assumed to be in url_charset encoding. Before the character is returned , the character will be converted from url_charset encoding to database encoding . If url_charset is NULL, the character will not be converted. The default value of url_charset is ISO-8859-1. |

#### Examples

The following anonymous block uses the ESCAPE function to escape the blank spaces in the URL:

```
DECLARE
  result varchar2(400);
BEGIN
 result := UTL_URL.UNESCAPE('http://www.example.com/Using%20the%20UNESCAPE%
20function.html');
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

The unescaped URL is:

```
http://www.example.com/Using the UNESCAPE function.html
```

# 18 PL/SQL functions and procedures

## 18.1 Overview

This chapter describes the Stored Procedure Language (SPL). SPL is a highly productive, procedural programming language for writing custom procedures, functions, triggers, and packages of POLARDB compatible with Oracle.

SPL provides the following features for developing applications:

- Full procedural programming functionality to complement the SQL language.
- A single, common language to create stored procedures, functions, triggers, and packages of POLARDB clusters compatible with Oracle.
- Integration with PgAdmin III, providing a seamless development and testing environment.
- The use of reusable code.
- Ease of use.

This chapter describes the basic elements of an SPL program, the organization of an SPL program, and how it is used to create a function or stored procedure.

In addition, this chapter delves into the details of the SPL language and provide examples of its application.

## 18.2 Basic SPL elements

## 18.2.1 Basic SPL elements

This topic describes the basic elements of an SPL program.

### Character set

You can use the following set of characters to write SPL programs:

- Uppercase letters A to Z and lowercase letters a to z
- Digits 0 to 9
- Symbols ( ) + - * / < > = ! ~ ^ ; : . ' @ % , " # $ & _ | { } ? [ ]
- White space characters tabs, spaces, and carriage returns

These characters can be used to write identifiers, expressions, statements, and control structures that comprise the SPL language.

**Note:**
The data that can be manipulated by an SPL program is determined by the character set supported by the database encoding.

**Case sensitivity**

Keywords and user-defined identifiers that are used in an SPL program are not case-sensitive.

For example, the statement

```
DBMS_OUTPUT.PUT_LINE('Hello World');
```

is equivalent to the following statements:

```
dbms_output.put_line('Hello World');
```

```
Dbms_Output.Put_Line('Hello World');
```

```
DBMS_output.Put_line('Hello World');
```

However, character constants, string constants, and other data obtained from the POLARDB compatible with Oracle database or external data sources, are case sensitive. The statement `DBMS_OUTPUT.PUT_LINE('Hello World!');` produces the following output:

```
Hello World!
```

However, the statement DBMS_OUTPUT.PUT_LINE('HELLO WORLD!'); produces the following output:

```
HELLO WORLD!
```

**Identifiers**

Identifiers are user-defined names that are used to identify various elements of an SPL program including variables, cursors, labels, programs, and parameters.

The syntax rules for valid identifiers in the SPL language are the same as for identifiers in the SQL language.

An identifier must be different from SPL or SQL keywords. The following are some examples of valid identifiers:

```
x
last    name
a_$_Sign
Many$$$$$$$$signs
THIS_IS_AN_EXTREMELY_LONG_NAME A1
```

**Qualifiers**

A qualifier is a name that specifies the owner or context of an entity that is the object of the qualification. A qualified object is specified as the qualifier name. The qualified object is followed by a period (.) and the name of the object being qualified. Note that the qualifier name and the period (.) has no white space in between. This syntax is called dot notation.

The following is an example of the syntax that is used for a qualified object.

```
qualifier. [ qualifier. ]... object
```

**qualifier** is the name of the object owner. **object** is the name of the entity that belongs to qualifier. It is possible to have a chain of qualifications where the preceding qualifier owns the entity identified by the subsequent qualifier(s) and object.

Almost any identifier can be qualified. What an identifier is qualified by depends upon what the identifier represents and the context of its usage.

Some examples of qualification are described as follows:

- Procedure and function names qualified by the schema to which they belong, such as schema_name.procedure_name(...)
- Trigger names qualified by the schema to which they belong, such as schema_name. trigger_name
- Column names qualified by the table to which they belong, such as emp.empno
- Table names qualified by the schema to which they belong, such as public.emp
- Column names qualified by table and schema, such as public.emp.empno

As a general rule, wherever a name appears in the syntax of an SPL statement, its qualified name can also be used.

A qualified name is used if two procedures that have the same name but belong to two different schemas are invoked from within a program. A qualified name is also used if the same name is used for a table column and SPL variable within the same program.

We do not recommend that you use qualified names. In this chapter, the following conventions are adopted to avoid naming conflicts:

- All variables declared in the declaration section of an SPL program are prefixed by v_, such as v_empno.
- All formal parameters declared in a procedure or function definition are prefixed by p_, such as p_empno.
- Column names and table names do not have any special prefix conventions, such as column empno in table emp.

**Constants**

In SPL programs, constants or literals are fixed values that can be used to represent values of various types, such as numbers, strings, and dates. Constants can be of the following types:

- Numeric (integer and real number)
- Character and string
- Date/time

## 18.2.2 User-defined PL/SQL subtypes

POLARDB compatible with Oracle supports user-defined PL/SQL subtypes and subtype aliases. A subtype is a data type with an optional set of constraints that restrict the values that can be stored in a column of that type. The rules that apply to the type on which the subtype is based are still enforced. However, you can use additional constraints to limit the precision or scale of values that match the type.

You can define a subtype in the declaration of a PL function, stored procedure, anonymous block, or package. Syntax:

```
SUBTYPE subtype_name IS type_name[(constraint)] [NOT NULL]
```

Where **constraint** is:

```
{precision [, scale]} | length
```

Where:

- **subtype_name** specifies the name of the subtype.

- **type_name** specifies the name of the original type on which the subtype is based.

> **Note:**
>
> Valid values of **type_name** are as follows:
>
> - The name of a type supported by POLARDB compatible with Oracle.
> - The name of a composite type.
> - A column anchored by a %TYPE operator.
> - The name of another subtype.

You can include the constraint clause to define restrictions for types that support precision or scale.

- **precision** specifies the total number of digits permitted in a value of the subtype.
- **scale** specifies the number of fractional digits permitted in a value of the subtype.
- **length** specifies the total length permitted in a value of CHARACTER, VARCHAR, or TEXT base types.

You can include the NOT NULL clause to specify that NULL values may not be stored in a column of the specified subtype.

> **Note:**
>
> A subtype that is based on a column will inherit the column size constraints, but the subtype will not inherit NOT NULL or CHECK constraints.

**Unconstrained subtypes**

To create an unconstrained subtype, use the SUBTYPE command to specify the new subtype name and the name of the type on which the subtype is based. For example, the following command creates a subtype named address that has all the attributes of the CHAR type:

```
SUBTYPE address IS CHAR;
```

You can also create a subtype (constrained or unconstrained) that is a subtype of another subtype:

```
SUBTYPE cust_address IS address NOT NULL;
```

This command creates a subtype named cust_address that shares all the attributes of the address subtype. You can include the NOT NULL clause to specify that a value of the cust_address may not be NULL.

### Constrained subtypes

You can include a length value when creating a subtype that is based on a character type to define the maximum length of the subtype. Example:

```
SUBTYPE acct_name IS VARCHAR (15);
```

This example creates a subtype named acct_name that is based on a VARCHAR data type, but is limited to 15 characters in length.

You can include values for precision to specify the maximum number of digits in a value of the subtype. You can also include scale to specify the number of digits to the right of the decimal point when constraining a numeric base type. Example:

```
SUBTYPE acct_balance IS NUMBER (5, 2);
```

This example creates a subtype named acct_balance that shares all the attributes of a NUMBER type. The subtype cannot exceed 3 digits to the left of the decimal point and 2 digits to the right of the decimal.

An argument declaration in a function or procedure header is a formal argument. The value passed to a function or stored procedure is an actual argument. When calling a function or stored procedure, the caller provides zero or more actual arguments. Each actual argument is assigned to a formal argument that holds the value within the body of the function or stored procedure.

If a formal argument is declared as a constrained subtype, then:

- POLARDB compatible with Oracle does not enforce subtype constraints when assigning an actual argument to a formal argument in a function call.
- POLARDB compatible with Oracle enforces subtype constraints when assigning an actual argument to a formal argument in a call of a stored procedure.

### Use the %TYPE operator

You can use the %TYPE notation to declare a subtype anchored to a column. Example:

```
SUBTYPE emp_type IS emp.empno%TYPE
```

This command creates a subtype named emp_type with a base type that matches the type of the empno column in the emp table. A subtype that is based on a column will share the column size constraints. The NOT NULL and CHECK constraints are not inherited.

### Subtype conversion

Unconstrained subtypes are aliases for the type on which they are based. Any variable of type or subtype (unconstrained) is interchangeable with a variable of the base type without conversion, and vice versa.

A variable of a constrained subtype can be interchanged with a variable of the base type without conversion. However, a variable of the base type can only be interchanged with a constrained subtype if it complies with the constraints of the subtype. A variable of a constrained subtype can be implicitly converted to another subtype. This happens if the variable is based on the same subtype, and the constraint values are within the values of the subtype to which it is being converted.

# 18.3 SPL programs

# 18.3.1 SPL block structure

An SPL program has the same block structure regardless of whether the program is a stored procedure, function, or trigger. A block consists of up to three sections: an optional declaration section, a mandatory executable section, and an optional exception section. A simplest block has an executable section that consists of one or more SPL statements within the keywords, BEGIN and END. The optional declaration section is used to declare variables , cursors, and types that are used by the statements within the executable and exception sections.

The declaration section appears before the BEGIN keyword of the executable section. The declaration section can begin with the keyword DECLARE, depending upon the context of where the block is used.

You can include an exception section within the BEGIN - END block. The exception section begins with the keyword, EXCEPTION, and continues until the end of the block in which it appears. If an exception is thrown by a statement within the block, program control goes to the exception section. In the exception section, the thrown exception may or may not be handled, depending on the exception and the contents of the exception section.

The following is the general structure of a block:

```
[ [ DECLARE ]
   declarations ]
  BEGIN
   statements
 [ EXCEPTION
```

```
    WHEN exception_condition THEN
      statements [, ...] ]
  END;
```

**Note:**

- **declarations** are one or more variable, cursor, or type declarations that are local to the block. Each declaration must be terminated by a semicolon (;). The use of the DECLARE keyword depends on the context in which the block appears.
- **statements** are one or more SPL statements. Each statement must be terminated by a semicolon (;). The end of the block denoted by the END keyword must also be terminated by a semicolon (;).

The EXCEPTION keyword marks the beginning of the exception section. exception_condition is a conditional expression that is used for the testing of one or more exception types. If an exception matches one of the exceptions in exception_condition, the statements that follow the WHEN exception_condition clause are run. One or more WHEN exception_condition clauses can exist and each clause is followed by statements.

**Note:**

Blocks can be nested because a BEGIN/END block in itself is considered a statement. The exception section can also contain nested blocks.

The following example shows the simplest possible block that consists of the NULL statement within the executable section. The NULL statement is an executable statement that does not have effect.

```
BEGIN
  NULL;
END;
```

The following block contains a declaration section and an executable section.

```
DECLARE
  v_numerator     NUMBER(2);
  v_denominator   NUMBER(2);
  v_result        NUMBER(5,2);
BEGIN
  v_numerator := 75;
  v_denominator := 14;
  v_result := v_numerator / v_denominator;
  DBMS_OUTPUT.PUT_LINE(v_numerator || ' divided by ' || v_denominator ||
    ' is ' || v_result);
```

```
END;
```

In the preceding example, three numeric variables are declared of data type NUMBER. Values are assigned to two of the variables, and one number is divided by the other number. The result is stored in a third variable that is used to display the result. The output of this block is as follows:

```
75 divided by 14 is 5.36
```

The following block consists of a declaration section, an executable section, and an exception section:

```
DECLARE
    v_numerator     NUMBER(2);
    v_denominator   NUMBER(2);
    v_result        NUMBER(5,2);
BEGIN
    v_numerator := 75;
    v_denominator := 0;
    v_result := v_numerator / v_denominator;
    DBMS_OUTPUT.PUT_LINE(v_numerator || ' divided by ' || v_denominator ||
        ' is ' || v_result);
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('An exception occurred');
END;
```

The following output shows that the statement within the exception section is executed as a result of the division by zero.

```
An exception occurred
```

## 18.3.2 Anonymous blocks

The preceding section describes the general structure of a block. Blocks facilitate code execution in POLARDB compatible with Oracle.

An anonymous block is unnamed and is not stored in the database. After an anonymous block is executed, it is cleared from the application buffer. This anonymous block cannot be re-executed unless the block code is re-entered into the application.

Anonymous blocks are useful for quick execution of programs, such as testing programs.

In most cases, the same block of code will be re-executed many times. To repeatedly run a block of code without re-entering the code each time, you can convert an anonymous block into a procedure or function with some simple modifications. The following section describes how to create a procedure or function that can be stored in the database and repeatedly called by another procedure, function, or application.

# 18.4 Procedure overview

Procedures are standalone SPL programs that are called as an individual SPL program statement. When called, stored procedures can receive values from the caller in the form of input parameters and return values to the caller in the form of output parameters.

## Create a stored procedure

The CREATE PROCEDURE command defines and names a standalone procedure that will be stored in the database.

```
CREATE [ OR REPLACE ] PROCEDURE name [ (parameters) ] [ AUTHID { DEFINER |
CURRENT_USER } ] { IS | AS }
[ declarations ] BEGIN
statements END [ name ];
```

name is the identifier of the stored procedure. If you specify the [OR REPLACE] clause and a procedure with the same name already exists in the schema, the new procedure will overwrite the existing one. If you do not specify the [OR REPLACE] clause, the new procedure will not overwrite the existing procedure with the same name in the same schema. parameters is a list of formal parameters. If the AUTHID clause is omitted or if AUTHID DEFINER is specified, the rights of the stored procedure owner are used to determine access privileges to database objects. In addition, the search path of the procedure owner is used to resolve unqualified object references. If the CURRENT_USER clause is specified, the rights of the current user who call the stored procedure are used to determine access privileges. In addition, the search path of the current user is used to resolve unqualified object references. declarations are variable, cursor, or type declarations . statements are SPL program statements. The BEGIN - END block can contain an EXCEPTION section.

The following example shows a simple stored procedure that does not require any parameters.

```
CREATE OR REPLACE PROCEDURE simple_procedure
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('That''s all folks!') ;
END simple_procedure;
```

As shown in the preceding example, you can store the procedure in the database by entering the procedure code in POLARDB compatible with Oracle.

### Call a stored procedure

To call a stored procedure from another SPL program, you can specify the stored procedure name followed by parameters and a semicolon.

```
name [ ([ parameters ]) ];
```

name is the identifier of the stored procedure. parameters is a list of actual parameters.

**Note:**

- If no actual parameters are specified, the stored procedure can be called with an empty parameter list, or the opening and closing parenthesis can be omitted.
- The syntax for calling a stored procedure is the same as that in the preceding syntax diagram when executing it with the EXEC command in psql or POLARDB compatible with Oracle.

The following example shows how to call the stored procedure from an anonymous block.

```
BEGIN
   simple_procedure;
END;

That's all folks!
```

**Note:**

Each application has its own unique method to call a stored procedure. For example, in a Java application, the application programming interface, JDBC, is used.

### Delete a stored procedure

You can run the DROP PROCEDURE command to delete a procedure from the database.

```
DROP PROCEDURE name;
```

name is the name of the stored procedure to be deleted.

The following example shows how to run the DROP PROCEDURE command to delete a procedure.

```
DROP PROCEDURE simple_procedure;
```

## 18.5 Function overview

Functions are Stored Procedure Language (SPL) programs that are called as expressions. When evaluated, a function returns a value that is substituted in the expression in which the function is embedded. Functions can receive values from the calling program in the form of input parameters. In addition to the fact that the function, itself, returns a value, a function can return extra values to the caller in the form of output parameters. However, we do not recommend that you use output parameters in functions.

### Create a function

The CREATE FUNCTION command defines and names a standalone function that will be stored in the database.

```
CREATE [ OR REPLACE ] FUNCTION name [ (parameters) ]
RETURN data_type [ AUTHID { DEFINER | CURRENT_USER } ] { IS | AS }
[ declarations ] BEGIN
statements
END [ name ];
```

**name** is the identifier of the function. If you specify the [OR REPLACE] clause and a function with the same name already exists in the schema, the new function will overwrite the existing one. If you do not specify the [OR REPLACE] clause, the new function will not overwrite the existing function with the same name in the same schema. **parameters** is a list of formal parameters. **data_type** is the data type of the value returned by the RETURN statement of the function. If the AUTHID clause is omitted or if AUTHID DEFINER is specified, the rights of the function owner are used to determine access privileges to database objects. In addition, the search path of the function owner is used to resolve unqualified object references. If the CURRENT_USER clause is specified, the rights of the current user who calls the function are used to determine access privileges. In addition, the search path of the current user is used to resolve unqualified object references. **declarations** are variable, cursor, or type declarations. **statements** are SPL program statements. The BEGIN - END block may contain an EXCEPTION section.

The following example shows a simple function that requires no parameters.

```
CREATE OR REPLACE FUNCTION simple_function
    RETURN VARCHAR2
```

```
IS
BEGIN
    RETURN 'That''s All Folks!' ;
END simple_function;
```

The following function requires two input parameters. For more information about function parameters, see the following sections.

```
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal           NUMBER,
    p_comm          NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END emp_comp;
```

### Call a function

A function can be used anywhere an expression can appear within an SPL statement. You can call a function by specifying its name followed by its parameters enclosed in parenthesis.

```
name [ ([ parameters ]) ]
```

**name** is the name of the function. **parameters** is a list of actual parameters.

**Note:**

If no actual parameters are specified, the function may be called with an empty parameter list, or the opening and closing parenthesis may be omitted.

The following example shows how to call the function from another SPL program.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(simple_function);
END;

That's All Folks!
```

A function is typically used within a SQL statement, as shown in the following example.

```
SELECT empno "EMPNO", ename "ENAME", sal "SAL", comm "COMM",
    emp_comp(sal, comm) "YEARLY COMPENSATION" FROM emp;

 EMPNO | ENAME  |   SAL   |  COMM   | YEARLY COMPENSATION
-------+--------+---------+---------+---------------------
  7369 | SMITH  |  800.00 |         |            19200.00
  7499 | ALLEN  | 1600.00 |  300.00 |            45600.00
  7521 | WARD   | 1250.00 |  500.00 |            42000.00
  7566 | JONES  | 2975.00 |         |            71400.00
  7654 | MARTIN | 1250.00 | 1400.00 |            63600.00
  7698 | BLAKE  | 2850.00 |         |            68400.00
  7782 | CLARK  | 2450.00 |         |            58800.00
  7788 | SCOTT  | 3000.00 |         |            72000.00
  7839 | KING   | 5000.00 |         |      120000.00
  7844 | TURNER | 1500.00 |    0.00 |       36000.00
  7876 | ADAMS  | 1100.00 |         |       26400.00
  7900 | JAMES  |  950.00 |         |       22800.00
  7902 | FORD   | 3000.00 |         |       72000.00
  7934 | MILLER | 1300.00 |         |       31200.00
(14 rows)
```

**Delete a function**

You can run the DROP FUNCTION command to remove a function from the database.

```
DROP FUNCTION name [ (parameters) ];
```

**name** is the name of the function to be deleted.

**Note:**

You must specify the parameter list in POLARDB compatible with Oracle under specific circumstances such as an overloaded function. However, Oracle requires that the parameter list always be omitted.

In the following example, a function is deleted.

```
DROP FUNCTION simple_function;
```

## 18.6 Parameters in stored procedures and functions

## 18.6.1 Overview

An important capability of stored procedures and functions is to receive data from the calling program and return data. This is achieved by using parameters.

Parameters are declared after the names of stored procedures or functions, enclosed in parentheses. Parameters defined in stored procedures or functions are called formal parameters. When a stored procedure or function is called, the calling program provides actual values for the called function or stored procedure. The calling program also provides the called function or stored procedure with the variables used to receive the results. The values and variables provided by a program when the program calls a stored procedure or function are called actual parameters.

The following code provides the syntax of a parameter declaration:

```
(name [ IN | OUT | IN OUT ] data_type [ DEFAULT value ])
```

name specifies the identifier assigned to the formal parameter. If an IN clause is specified, the IN parameter receives input data that is intended to be used by the stored procedure or function. You can use default values to initialize the input parameters. If an OUT clause is specified, the OUT parameter returns the results of the stored procedure or function to the calling program. If an IN OUT clause is specified, the IN OUT parameter can be used as both input and output parameters. If no IN, OUT, or IN OUT clause is specified, the parameter is defined as an input parameter by default. The use of a parameter is determined by IN, OUT , and IN OUT. data_type specifies the data type of the parameter. value specifies the default value assigned to an IN parameter if the actual parameter is not specified during a call.

The following example shows a stored procedure with parameters:

```
CREATE OR REPLACE PROCEDURE emp_query (
    p_deptno       IN     NUMBER,
    p_empno        IN OUT NUMBER,
    p_ename        IN OUT VARCHAR2,
    p_job          OUT    VARCHAR2,
    p_hiredate     OUT    DATE,
    p_sal          OUT    NUMBER
)
IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
        INTO p_empno, p_ename, p_job, p_hiredate, p_sal
        FROM emp
        WHERE deptno = p_deptno
          AND (empno = p_empno
           OR  ename = UPPER(p_ename));
END;
```

In this example, p_deptno is an IN formal parameter. p_empno and p_ename are IN OUT formal parameters. p_job, p_hiredate, and p_sal are OUT formal parameters.

**Note:**

In the preceding example, the maximum length of the VARCHAR2 type parameter and the precision and scale of the NUMBER type parameter are not specified. In the parameter declarations, you cannot specify the length, precision, value range, or other limits. The limits are automatically inherited from the actual parameters that are used when you call a stored procedure or function.

Other programs can call the emp_query stored procedure and pass actual parameters to it. The following example describes another SPL program that calls the emp_query stored procedure.

```
DECLARE
    v_deptno        NUMBER(2);
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER;
BEGIN
    v_deptno := 30;
    v_empno  := 7900;
    v_ename  := '';
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
END;
```

In the preceding example, v_deptno, v_empno, v_ename, v_job, v_hiredate, and v_sal are actual parameters.

The output of the preceding example is provided as follows:

```
Department : 30
Employee No: 7900
Name       : JAMES
Job        : CLERK
Hire Date  : 03-DEC-81
Salary     : 950
```

## 18.6.2 Positional and named parameter notation

When you pass a parameter to a function or stored procedure, you can either use positional parameter notation or named parameter notation. If you use positional notation, you must list the parameters in the declared order. If you use named notation, the order of parameters is not important.

If you use named notation, you must list the name of each parameter followed by an arrow (=>) and a parameter value. Your workloads increase if you use named notation, but named notation makes your code easy to read and maintain.

### Example

The following example describes how to use positional parameter notation and named parameter notation:

```
CREATE OR REPLACE PROCEDURE emp_info (
    p_deptno       IN     NUMBER,
    p_empno        IN OUT NUMBER,
    p_ename        IN OUT VARCHAR2,
)
IS
BEGIN
    dbms_output.put_line('Department Number =' || p_deptno);
    dbms_output.put_line('Employee Number =' || p_empno);
    dbms_output.put_line('Employee Name =' || p_ename;
END;
```

If you use positional notation to call a stored procedure, pass the following information:

```
emp_info(30, 7455, 'Clark');
```

If you use named notation to call a stored procedure, pass the following information:

```
emp_info(p_ename =>'Clark', p_empno=>7455, p_deptno=>30);
```

If the parameter list is changed, the parameters are reordered, or an optional parameter is added, named notation can reduce the need to rearrange the parameter list of a stored procedure.

If you specify a default value for a parameter and this parameter is not a trailing parameter, you must use named notation to call a stored procedure or function. The following example describes a stored procedure that has two leading default parameters:

```
CREATE OR REPLACE PROCEDURE check_balance (
    p_customerID  IN NUMBER DEFAULT NULL,
    p_balance     IN NUMBER DEFAULT NULL,
    p_amount      IN NUMBER
)
IS
DECLARE
    balance NUMBER;
BEGIN
   IF (p_balance IS NULL AND p_customerID IS NULL) THEN
      RAISE_APPLICATION_ERROR
          (-20010, 'Must provide balance or customer');
   ELSEIF (p_balance IS NOT NULL AND p_customerID IS NOT NULL) THEN
      RAISE_APPLICATION_ERROR
          (-20020,'Must provide balance or customer, not both');
   ELSEIF (p_balance IS NULL) THEN
      balance := getCustomerBalance(p_customerID);
   ELSE
      balance := p_balance;
   END IF;

   IF (amount > balance) THEN
```

```
      RAISE_APPLICATION_ERROR
        (-20030, 'Balance insufficient');
   END IF;
END;
```

You can only ignore non-trailing parameter values if you use named notation to call the preceding stored procedure. If positional notation is applied, you can only assign default values to trailing parameters. You can call the preceding stored procedure by specifying parameters as follows:

- `check_balance(p_customerID => 10, p_amount = 500.00)`
- `check_balance(p_balance => 1000.00, p_amount = 500.00)`

You can specify parameters by using a combination of positional and named notation, which is called mixed notation. The following example describes how to use mixed parameter notation:

```
CREATE OR REPLACE PROCEDURE emp_info (
    p_deptno       IN     NUMBER,
    p_empno        IN OUT NUMBER,
    p_ename        IN OUT VARCHAR2,
)
IS
BEGIN
    dbms_output.put_line('Department Number =' || p_deptno);
    dbms_output.put_line('Employee Number =' || p_empno);
    dbms_output.put_line('Employee Name =' || p_ename;
END;
```

You can use mixed notation to call the stored procedure.

```
emp_info(30, p_ename =>'Clark', p_empno=>7455);
```

If you use mixed notation to call a stored procedure, named parameters must not precede positional parameters.

## 18.6.3 Parameter modes

A parameter has the three possible modes: IN, OUT, and IN OUT. The following features of a formal parameter depend on the parameter mode:

- The initial value of the formal parameter when the stored procedure or function is called.
- Whether the called stored procedure or function can modify the formal parameter.
- The process of passing the value of the actual parameter from the calling program to the called program.
- The output value of the formal parameter when an unhandled exception occurs in the called program.

The following table summarizes the behavior of each parameter based on the parameter mode.

| Mode property | IN | IN OUT | OUT |
|---|---|---|---|
| The initial value of the formal parameter | The actual parameter value | The actual parameter value | The actual parameter value |
| Whether the called program can modify the formal parameter | No | Yes | Yes |
| The value of the actual parameter after normal termination of the called program | The original actual parameter value prior to the call | The last value of the formal parameter | The last value of the formal parameter |
| The value of the actual parameter after a handled exception in the called program | The original actual parameter value prior to the call | The last value of the formal parameter | The last value of the formal parameter |
| The value of the actual parameter after an unhandled exception in the called program | The original actual parameter value prior to the call | The original actual parameter value prior to the call | The original actual parameter value prior to the call |

As shown in the table, an IN formal parameter is initialized to an actual parameter only when called, unless it is explicitly initialized with a default value. The IN parameter can be referenced in the called program, but the called program may not assign a new value to the IN parameter. When the called program ends and control returns to the calling program, the actual parameter contains the same value as the parameter is set to before the call.

The OUT formal parameter is initialized to the actual parameter only when called. The called program can reference and assign a new value to the formal parameter. If the called program ends without an exception, the value of the actual parameter is the last value that is assigned to the formal parameter. If a handled exception occurs, the value of the actual parameter is the last value that is assigned to the formal parameter. If an unhandled exception occurs, the value of the actual parameter is the value that is assigned before the call.

Similar to an IN parameter, an IN OUT formal parameter is initialized to the actual parameter only when it is called. Similar to an OUT parameter, an IN OUT formal parameter can be modified by the called program. If the called program ends with no exceptions, the last value of the formal parameter is passed to the actual parameter. If a handled exception occurs, the value of the actual parameter is the last value that is assigned to the formal parameter. If an unhandled exception occurs, the value of the actual parameter is the value that is assigned before the call.

## 18.6.4 Use default values in parameters

In the CREATE PROCEDURE or CREATE FUNCTION statement, you can set a default value for a formal parameter by including the DEFAULT clause or using the assignment operator (:=).

The following syntax describes the general format of a parameter declaration:

```
(name [ IN|OUT|IN OUT ] data_type [{DEFAULT | := } expr ])
```

name specifies the identifier assigned to the parameter. IN|OUT|IN OUT specifies the parameter mode. data_type specifies the data type assigned to the variable. expr specifies the default value assigned to the parameter. You must provide a value for the parameter if a DEFAULT clause is not included.

Each time you call a function or stored procedure, the default value is evaluated. For example, if you assign SYSDATE to a parameter of the DATE type, the value of the parameter will be the time of the current call. The parameter value no longer indicates the time when the stored procedure or function is created.

The following example describes how to use the assignment operator to set the hiredate parameter to a default value of SYSDATE for a stored procedure:

```
CREATE OR REPLACE PROCEDURE hire_emp (
    p_empno         NUMBER,
    p_ename         VARCHAR2,
    p_hiredate      DATE := SYSDATE
) RETURN
IS
BEGIN
    INSERT INTO emp(empno, ename, hiredate)
                VALUES(p_empno, p_ename, p_hiredate);

    DBMS_OUTPUT.PUT_LINE('Hired!') ;
END emp_comp;
```

When calling the function, you can omit a parameter from the actual parameter list if the parameter uses a default value in the parameter declaration. For the preceding example, the stored procedure (hire_emp) must contain two parameters: the employee number (p_empno) and employee name (p_empno). The default value of the third parameter (p_hiredate) is SYSDATE.

```
hire_emp 7575, Clark
```

If you include a value for the actual parameter when calling the function, the value takes precedence over the default value.

```
hire_emp 7575, Clark, 15-FEB-2010
```

This example indicates that you add a new employee whose employment date is February 15, 2010 regardless of the current value of SYSDATE.

You can also use the DEFAULT keyword to replace the assignment operator to write the same function.

```
CREATE OR REPLACE PROCEDURE hire_emp (
   p_empno         NUMBER,
   p_ename         VARCHAR2,
   p_hiredate      DATE DEFAULT SYSDATE
) RETURN
IS
BEGIN
   INSERT INTO emp(empno, ename, hiredate)
               VALUES(p_empno, p_ename, p_hiredate);

   DBMS_OUTPUT.PUT_LINE('Hired!') ;
END emp_comp;
```

## 18.7 Compilation errors in stored procedures and functions

POLARDB compatible with Oracle supports parsers for compiling functions and stored procedures. Parsers verify that the CREATE statement and the program body (the program portion following the AS keyword) conform to the SPL and SQL syntax. If a parser detects an error, the server automatically stops the compilation process. Note that the parser detects syntax errors in expressions, rather than semantic errors. For example, if an expression references a nonexistent column, table, function, or a value of the incorrect type, an exception is thrown.

You can instruct the server to stop parsing if the parser finds one or more errors in SPL code or an error in SQL code. You can specify the spl.max_error_count parameter to control the maximum number of errors that are allowed in SPL code. The default value of the spl.max_error_count parameter is 10. The maximum value is 1000. You can set the value of spl.max_error_count to 1, which instructs the server to stop parsing when the first error in SPL or SQL code occurs.

You can use the SET command in the current session to specify a value for spl.max_error_count. Syntax:

```
SET spl.max_error_count = number_of_errors
```

number_of_errors specifies the number of SPL code errors that is allowed to occur before the server stops the compilation process. Example:

```
SET spl.max_error_count = 6
```

In this example, the server continues parsing regardless of the first five SPL code errors. When the sixth error occurs, the server stops parsing, and the six detailed error messages and an error summary are displayed.

When developing new code or importing existing code from other sources, you can set the spl.max_error_count parameter to a large value to save time.

You can instruct the server to continue parsing when an error occurs in the SPL code of a program body. The parser may then encounter an error in an SQL code segment. In this case, errors may still exist in any SPL or SQL code that follows the invalid SQL code. For example, two errors exist in the following code:

```
CREATE FUNCTION computeBonus(baseSalary number) RETURN number AS
BEGIN

    bonus := baseSalary * 1.10;
    total := bonus + 100;

    RETURN bonus;
END;

ERROR:  "bonus" is not a known variable
LINE 4:     bonus := baseSalary * 1.10;
            ^
ERROR:  "total" is not a known variable
LINE 5: total: = bonus + 100;
            ^
ERROR:  compilation of SPL function/procedure "computebonus" failed due to 2 errors
```

In the following example, a new SELECT statement is added to the preceding example. The error in the SELECT statement masks other errors that follow.

```
CREATE FUNCTION computeBonus(employeeName number) RETURN number AS
BEGIN
    SELECT salary INTO baseSalary FROM emp
      WHERE ename = employeeName;

    bonus := baseSalary * 1.10;
    total := bonus + 100;

    RETURN bonus;
```

```
END;

ERROR:  "basesalary" is not a known variable
LINE 3:     SELECT salary INTO baseSalary FROM emp WHERE ename = emp...
```

# 18.8 Program security

## 18.8.1 Overview

The following factors determine the security concerning what users can execute an SPL program and what database objects can be accessed by users who are executing the SPL program:

- Permission to execute the program.
- Permissions granted on the database objects (including other SPL programs) that the program attempts to access.
- Whether the SPL program is created with the definer's or caller's permission.

## 18.8.2 EXECUTE permission

An SPL program (including functions, stored procedures, and packages) can be executed only when any of the following conditions is true:

- The current user who calls the SPL program is a superuser.
- The current user who calls the SPL program has been granted the EXECUTE permission on the SPL program.
- The current user who calls the SPL program inherits the EXECUTE permission by becoming a member of the group that has been granted the EXECUTE permission on the SPL program.
- The EXECUTE permission has been granted to the PUBLIC group.

When the SPL program is created in POLARDB with Oracle, the EXECUTE permission is granted to the PUBLIC group by default. Therefore, any user can execute the program.

You can remove the default setting by running the REVOKE EXECUTE command. For more information, see REVOKE command. The following code provides an example of the command:

```
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
```

You can grant the EXECUTE permission on the SPL program to the specified user or group.

```
GRANT EXECUTE ON PROCEDURE list_emp TO john;
```

In this example, the user john can execute the list_emp program, but the users who do not meet the conditions listed at the beginning of this topic cannot execute the program.

After a program starts execution, permission checks are required before you perform any of the following operations on database objects:

- Read or modify data in tables and views.
- Create, modify, or delete database objects, such as tables, views, indexes, and sequences.
- Retrieve the current or next value from a sequence.
- Call another program, such as a function, stored procedure, or package.

You can ensure the security of operations by limiting the permissions on database objects.

Note that a database may have multiple objects that have the same name and type, but belong to different schemas. For more information about which object is to be referenced by an SPL program in this case, see the next topic.

## 18.8.3 Database object name resolution

The database objects in the SPL program can be referenced by using either a qualified name or an unqualified name. The form of a qualified name is schema.name, in which schema specifies the name of the schema for a database object and name specifies the name of the database object. An unqualified name does not contain the "schema." portion . If a qualified name is used, the database object is precisely specified. In the specified schema, the object either exists or does not exist.

If you use an unqualified name to locate an object, you must use the search path of the current user. If a user becomes the current user of the session, the default search path is used to associate with the user. A search path consists of a list of schemas. The search sequence is from left to right when a data object with an unqualified name is referenced. If no corresponding object is found from the list of schemas in the search path, the object does not exist. You can use the SHOW search_path command in POLARDB compatible with Oracle to display the default search path.

```
SHOW search_path;

    Search_path
----------------------
 $ User, public, sys, dbo
(1 row)
```

In this example, $user is a placeholder that refers to the current user. If the current user in the preceding session is enterprisedb, a database object with an unqualified name is searched in order in the following schemas: enterprisedb, public, sys, and dbo.

When an unqualified database object is parsed in the search path, the system checks whether the current user is authorized to perform the corresponding operations on the specified object.

**Note:**

The concept of the search path in POLARDB compatible with Oracle is incompatible with that in Oracle databases. For an unqualified reference, Oracle database services only search for database objects with the specified name in the schema of the current user. In Oracle databases, a user and the schema are the same entity. However, in POLARDB compatible with Oracle, the user and the schema are two different objects.

## 18.8.4 Database object permissions

Before an SPL program is executed, the system checks whether the current user is authorized to access the database objects that are referenced in the program. The GRANT and REVOKE commands are used to grant and remove related permissions on database objects. If the current user attempts to access a database object without permissions, the program will generate an exception.

## 18.8.5 Comparison of the definer's permission and caller's permission

Before an SPL program is executed, the system identifies the user who is associated with the process. The user associated with the process is called the current user. The search path of the current user is used to parse unqualified object references. The database object permissions of the current user determine whether the related database objects can be referenced in the program.

The selection of the current user is based on whether the SPL program is created with the definer's permission or caller's permission. The AUTHID clause is used to determine this selection. The AUTHID DEFINER clause is used to grant the definer's permission to the program. The AUTHID clause is omitted by default. The AUTHID CURRENT_USER clause is used to grant the definer's permission to the program. The following section summarizes the differences between the preceding two permissions:

- If a program has the definer's permission, the owner of the program becomes the current user when the program is executed. The search path of the program owner is used to parse unqualified object references. The database object permissions of the program owner can be used to determine whether access to a referenced object is allowed. For a program created with the definer's permission, the current user is irrelevant to the caller of the program.
- If a program has the caller's permission, the current user when the program is called remains the current user during the program execution (but not necessarily in called subprograms). For more information, see the following bullet points. When a program with the caller's permission is called, the current user is the user who starts the session, for example, establishing database connections. The SET ROLE command can be used to change the current user after the session starts. For a program created with the caller's permission, the current user is irrelevant to the owner of the program.

The following section summarizes the observations generated from the preceding definitions:

- The first observation details the status of the current user when a program created with the definer's permission calls another program created with the definer's permission . The current user changes from the owner of the calling program to the owner of the called program during the execution of the called program.
- The second observation details the status of the current user when a program created with the definer's permission calls another program created with the caller's permission. The owner of the calling program remains the current user during the execution of both the calling and called programs.
- The third observation details the status of the current user when a program created with the caller's permission calls another program created with the caller's permission. The current user of the calling program remains the current user during the execution of the called program.

- The fourth observation details the status of the current user when a program created with the caller's permission calls another program created with the definer's permission. The current user changes to the owner of the called program during the execution of the called program.

If the called program in turn calls another program in the preceding cases, the corresponding principles still apply.

## 18.8.6 Example of the security mechanism

In the following example, a new database is created with two users. One user is hr_mgr, who owns the hr_mgr schema that contains a copy of the entire sample app. The other user is sales_mgr, who owns the sales_mgr schema that contains a copy of the emp table. The table provides a list of sales employees.

In this example, the list_emp stored procedure, hire_clerk function, and emp_admin package are used. To present a more secure environment, all the default permissions that are granted when the sample app is installed are removed. Then, required permissions are re-granted to the sample app.

The list_emp and hire_clerk programs are changed from the default definer's permission to the caller's permission. When the sales_mgr user runs the programs, the programs act on the emp table in the sales_mgr schema. This occurs because search path and permissions of the sale_mgr user are used for name resolution and authorization checks.

Then, the sale_mgr user executes the get_dept_name and hire_emp programs that are included in the emp_admin package. The dept and emp tables in the hr_mgr schema can be accessed because the hr_mgr user is the owner of the emp_admin package that is using the definer's permission.

### Step 1: Create a database and two users

Use the user identity enterprisedb to create the hr database.

```
CREATE DATABASE hr;
```

Switch to the hr database and create the required users:

```
\c hr enterprisedb
CREATE USER hr_mgr IDENTIFIED BY password;
CREATE USER sales_mgr IDENTIFIED BY password;
```

### Step 2: Create the sample app

Create the sample app owned by the hr_mgr user in the hr_mgr schema.

```
\c - hr_mgr
\i C:/Program Files/PostgresPlus/9.3AS/installer/server/edb-sample.sql

BEGIN
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE VIEW
CREATE SEQUENCE
        .
        .
        .
CREATE PACKAGE
CREATE PACKAGE BODY
COMMIT
```

### Step 3: Create the emp table in the sales_mgr schema

Create a subset of the emp table owned by the sales_mgr user in the sales_mgr schema.

```
\c – hr_mgr
GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
\c – sales_mgr
CREATE TABLE emp AS SELECT * FROM hr_mgr.emp WHERE job = 'SALESMAN';
```

In the preceding example, the GRANT USAGE ON SCHEMA command is used to authorize the sales_mgr user to access the hr_mgr schema so that the user can make a copy of the emp table owned by the hr_mgr user. This step is only required in POLARDB compatible with Oracle and is incompatible with Oracle databases, which regard the schema and its user as the same entity.

### Step 4: Remove the default permissions

Remove all permissions, and illustrate the minimum required permissions.

```
\c – hr_mgr
REVOKE USAGE ON SCHEMA hr_mgr FROM sales_mgr;
REVOKE ALL ON dept FROM PUBLIC;
REVOKE ALL ON emp FROM PUBLIC;
REVOKE ALL ON next_empno FROM PUBLIC;
REVOKE EXECUTE ON FUNCTION new_empno() FROM PUBLIC;
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
REVOKE EXECUTE ON FUNCTION hire_clerk(VARCHAR2,NUMBER) FROM PUBLIC;
REVOKE EXECUTE ON PACKAGE emp_admin FROM PUBLIC;
```

**Step 5: Grant the caller's permission to the list_emp stored procedure**

When you connect to the database by using the user identity hr_mgr, add the AUTHID CURRENT_USER clause to the list_emp program and save the clause in POLARDB compatible with Oracle. When you perform this step, make sure that you have logged on as the hr_mgr user. Otherwise, the modified program may be saved in the public schema rather than the hr_mgr schema.

```
CREATE OR REPLACE PROCEDURE list_emp
AUTHID CURRENT_USER
IS
   v_empno        NUMBER(4);
   v_ename        VARCHAR2(10);
   CURSOR emp_cur IS
      SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
   OPEN emp_cur;
   DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
   DBMS_OUTPUT.PUT_LINE('-----    -------');
   LOOP
      FETCH emp_cur INTO v_empno, v_ename;
      EXIT WHEN emp_cur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(v_empno ‖ '     ' ‖ v_ename);
   END LOOP;
   CLOSE emp_cur;
END;
```

**Step 6: Grant the caller's permission to the hire_clerk program and qualify the calling of the new_empno function to the hr_mrg schema**

When you connect to the database by using the user identity hr_mgr, add the AUTHID CURRENT_USER clause to the hire_clerk program.

To ensure that the hire_clerk function calls the new_empno function in the hr_mgr schema , you must fully qualify new_empno to hr_mgr.new_empno after the BEGIN keyword. The high_clerk function is a program with the caller's permission. If you call the new_empno function but do not qualify it, the new_empno function in the search path of the caller is executed, rather than that in the hr_mrg schema.

When you save the program, make sure that you have logged on as the hr_mgr user. Otherwise, the modified program may be saved in the public schema rather than the hr_mgr schema.

```
CREATE OR REPLACE FUNCTION hire_clerk (
   p_ename        VARCHAR2,
   p_deptno       NUMBER
) RETURN NUMBER
AUTHID CURRENT_USER
IS
```

```
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_mgr           NUMBER(4);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_deptno        NUMBER(2);
BEGIN
    v_empno := hr_mgr.new_empno;
    INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
        TRUNC(SYSDATE), 950.00, NULL, p_deptno);
    SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
        FROM emp WHERE empno = v_empno;
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Manager    : ' || v_mgr);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission : ' || v_comm);
    RETURN v_empno;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
        RETURN -1;
END;
```

**Step 7: Grant the required permissions**

When you connect to the database by using the user identity hr_mgr, you must grant the required permissions to the sales_mgr user for accessing the list_emp stored procedure, hire_clerk function, and emp_admin package. Note that the emp table in the sales_mgr schema is the only data object that the sales_mgr user can access. sales_mgr has no permission to access tables in the hr_mgr schema.

```
GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
GRANT EXECUTE ON PROCEDURE list_emp TO sales_mgr;
GRANT EXECUTE ON FUNCTION hire_clerk(VARCHAR2,NUMBER) TO sales_mgr;
GRANT EXECUTE ON FUNCTION new_empno() TO sales_mgr;
GRANT EXECUTE ON PACKAGE emp_admin TO sales_mgr;
```

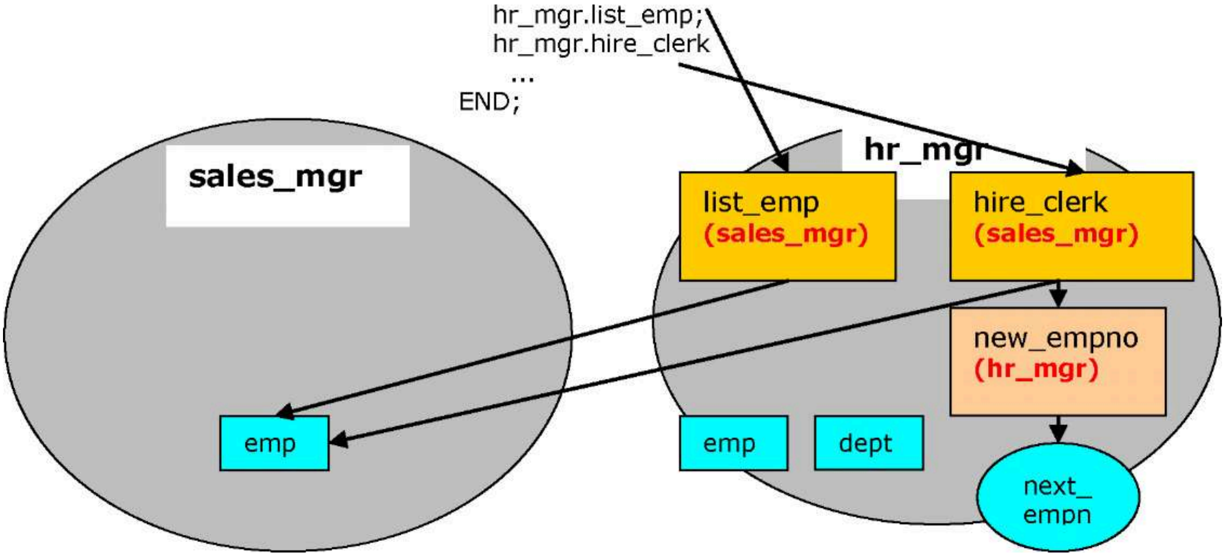
**Step 8: Run the list_emp and hire_clerk programs**

Connect to the database by using the user identity sales_mgr and run the following anonymous code block:

```
\c – sales_mgr
DECLARE
    v_empno         NUMBER(4);
BEGIN
    hr_mgr.list_emp;
    DBMS_OUTPUT.PUT_LINE('*** Adding new employee ***');
    v_empno := hr_mgr.hire_clerk('JONES',40);
    DBMS_OUTPUT.PUT_LINE('*** After new employee added ***');
    hr_mgr.list_emp;
END;

EMPNO   ENAME
-----   -------
7499    ALLEN
7521    WARD
7654    MARTIN
7844    TURNER
*** Adding new employee ***
Department : 40
Employee No: 8000
Name       : JONES
Job        : CLERK
Manager    : 7782
Hire Date  : 08-NOV-07 00:00:00
Salary     : 950.00
*** After new employee added ***
EMPNO   ENAME
-----   -------
7499    ALLEN
7521    WARD
7654    MARTIN
7844    TURNER
8000    JONES
```

The following diagram shows the tables and sequences accessed by the anonymous code block. The gray ovals represent the sales_mgr and hr_mgr schemas. The current user during the execution of each program is displayed in bold red font within parenthesis.



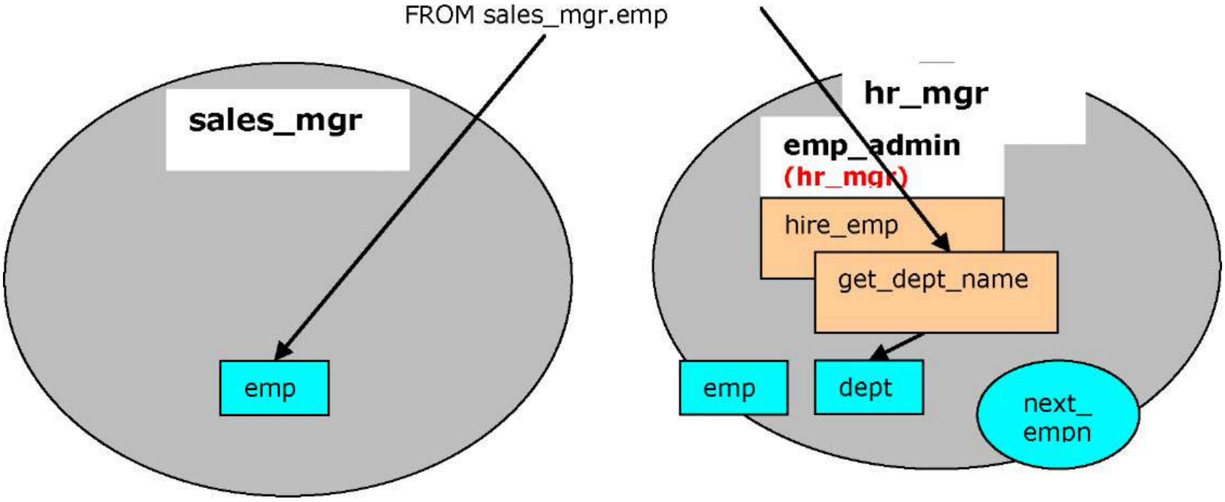
The following query result on the emp table in the sales_mgr schema shows that the update is applied to the table.

```
SELECT empno, ename, hiredate, sal, deptno, hr_mgr.emp_admin.get_dept_name(
deptno) FROM sales_mgr.emp;

empno | ename |     hiredate      |  sal  | deptno | get_dept_name
-------+--------+--------------------+---------+--------+---------------
```

```
  7499 | ALLEN  | 20-FEB-81 00:00:00 | 1600.00 |     30 | SALES
  7521 | WARD   | 22-FEB-81 00:00:00 | 1250.00 |     30 | SALES
  7654 | MARTIN | 28-SEP-81 00:00:00 | 1250.00 |     30 | SALES
  7844 | TURNER | 08-SEP-81 00:00:00 | 1500.00 |     30 | SALES
  8000 | JONES  | 08-NOV-07 00:00:00 |  950.00 |     40 | OPERATIONS
(5 rows)
```

The following diagram shows that the SELECT command references the emp table in the sales_mgr schema. The dept table referenced by the get_dept_name function in the emp_admin package is from the hr_mgr schema because the emp_admin package has the definer's permission and is owned by hr_mgr.

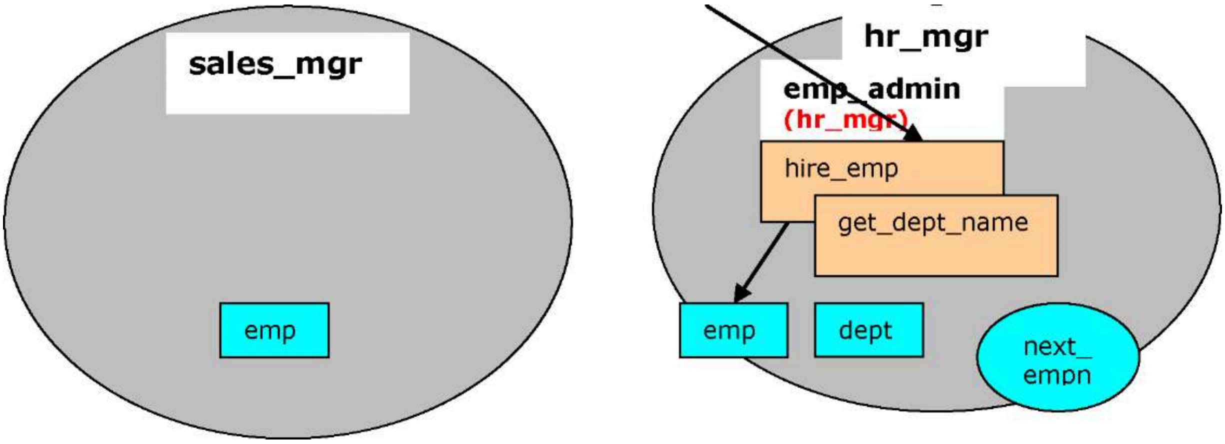


**Step 9: Run the hire_emp program in the emp_admin package**

When you connect to the database by using the user identity sales_mgr, run the hire_emp program in the emp_admin package.

```
EXEC hr_mgr.emp_admin.hire_emp(9001, 'ALICE','SALESMAN',8000,TRUNC(SYSDATE),1000
,7369,40);
```

The following diagram shows that the emp table in the hr_mgr schema is updated by the hire_emp stored procedure in the emp_admin package that has the definer's permission.



Connect to the database by using the user identity hr_mgr. Use the following SELECT command to check whether the information of new employees has been added to the emp table owned by the hr_mgr user. The emp_admin package has the definer's permission and hr_mgr is the owner of the emp_admin package.

```
\c – hr_mgr
SELECT empno, ename, hiredate, sal, deptno, hr_mgr.emp_admin.get_dept_name(
deptno) FROM hr_mgr.emp;

empno | ename  |      hiredate      |   sal   | deptno | get_dept_name
-------+--------+--------------------+---------+--------+---------------
  7369 | SMITH  | 17-DEC-80 00:00:00 |  800.00 |     20 | RESEARCH
  7499 | ALLEN  | 20-FEB-81 00:00:00 | 1600.00 |     30 | SALES
  7521 | WARD   | 22-FEB-81 00:00:00 | 1250.00 |     30 | SALES
  7566 | JONES  | 02-APR-81 00:00:00 | 2975.00 |     20 | RESEARCH
  7654 | MARTIN | 28-SEP-81 00:00:00 | 1250.00 |     30 | SALES
  7698 | BLAKE  | 01-MAY-81 00:00:00 | 2850.00 |     30 | SALES
  7782 | CLARK  | 09-JUN-81 00:00:00 | 2450.00 |     10 | ACCOUNTING
  7788 | SCOTT  | 19-APR-87 00:00:00 | 3000.00 |     20 | RESEARCH
  7839 | KING   | 17-NOV-81 00:00:00 | 5000.00 |     10 | ACCOUNTING
  7844 | TURNER | 08-SEP-81 00:00:00 | 1500.00 |     30 | SALES
  7876 | ADAMS  | 23-MAY-87 00:00:00 | 1100.00 |     20 | RESEARCH
  7900 | JAMES  | 03-DEC-81 00:00:00 |  950.00 |     30 | SALES
  7902 | FORD   | 03-DEC-81 00:00:00 | 3000.00 |     20 | RESEARCH
  7934 | MILLER | 23-JAN-82 00:00:00 | 1300.00 |     10 | ACCOUNTING
  9001 | ALICE  | 08-NOV-07 00:00:00 | 8000.00 |     40 | OPERATIONS
(15 rows)
```

# 19 Develop PL/SQL packages

## 19.1 Overview

This topic introduces the concept of packages in POLARDB compatible with Oracle. A package is a named collection of functions, procedures, variables, cursors, user-defined record types, and records that are referenced by using a common qualifier, the package identifier. Packages have the following characteristics:

- Packages provide a convenient means of organizing the functions and procedures that perform a related purpose. Permissions to use the package functions and procedures are dependent upon one privilege granted to the entire package. All of the package programs must be referenced with a common name.
- Certain functions, procedures, variables, and types in the package can be declared as public. Public entities are visible and can be referenced by other programs that are given the EXECUTE privilege on the package. For public functions and procedures, only their signatures are visible: the program names, parameters if any, and return types of functions. The SPL code of these functions and procedures is not accessible to others, therefore applications that utilize a package are dependent only upon the information available in the signature instead of the procedural logic itself.
- Other functions, procedures, variables, and types in the package can be declared as private. Private entities can be referenced and used by functions and procedures within the package, but not by other external applications.
- Function and procedure names can be overloaded within a package. One or more functions or procedures can be defined with the same name, but with different signatures. This provides the capability to create identically named programs that perform the same job, but on different types of input.

## 19.2 Package components

Packages consist of two main components:

- The package specification: This is the public interface. These are the elements which can be referenced outside the package. The specification declares all database objects that are to be a part of the package.

- The package body: This contains the actual implementation of all the database objects declared within the package specification.

The package body implements the specifications in the package specification. It contains implementation details and private declarations which are invisible to the application. You can debug, enhance, or replace a package body without changing the specifications. Similarly, you can change the body without recompiling the calling programs because the implementation details are invisible to the application.

**Package specification syntax**

The package specification defines the user interface for a package (the API). The specificat ion lists the functions, procedures, types, exceptions, and cursors that are visible to a user of the package.

The syntax used to define the interface for a package is:

```
CREATE [ OR REPLACE ] PACKAGE package_name
  [ authorization_clause ]
  { IS | AS }
  [ declaration; ] ...
  [ procedure_or_function_declaration; ] ...
END [ package_name ] ;
```

Where authorization clause : =

```
{ AUTHID DEFINER } | { AUTHID CURRENT_USER }
```

Where procedure or function declaration :=

```
procedure declaration | function declaration
```

Where procedure declaration :=

```
PROCEDURE proc name[ argument list ] [restriction pragma];
```

Where function_declaration :=

```
FUNCTION func_name [ argument_list ]

RETURN rettype [ restriction pragma ];
```

```
```

Where argument_list :=

```
( argument declaration [, ...] )
```

Where argument declaration :=

```
argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
```

Where restriction pragma : =

```
PRAGMA RESTRICT_REFERENCES(name, restrictions)
```

Where restrictions :=

```
restriction [, ... ]
```

### Parameters

| Parameter | Description |
|---|---|
| package_name | package_name is an identifier assigned to the package and each package must have a name unique within the schema. |
| AUTHID DEFINER | If you omit the AUTHID clause or specify AUTHID DEFINER: The privileges of the package owner are used to determine access privileges to database objects and the search paths of the package owner are used to resolve the reference of the unqualified database object. |
| AUTHID CURRENT_USER | If you specify AUTHID CURRENT_USER: the privileges of the current user that executes a program in the package are used to determine access privileges to database objects and search paths of the current user that executes a program in the package are used to resolve the reference of the unqualified database object. |

| Parameter | Description |
|---|---|
| declaration | declaration is an identifier of a public variable. A public variable can be accessed from outside of the package by using the package_name.variable syntax. There can be zero, one, or more public variables. Public variable definitions must come before procedure or function declarations. declaration can be any of the following types:<br>• Variable declaration<br>• Record declaration<br>• Collection declaration<br>• REF CURSOR and cursor variable declaration<br>• TYPE definitions for records, collections, and REF CURSORs<br>• Exception<br>• Object variable declaration |
| argname | The name of an argument. The argument is referenced by this name within the function or procedure body. |
| IN \| IN OUT \| OUT | The argument mode.<br>• IN declares the argument for input only. This is the default mode.<br>• IN OUT allows the argument to receive a value or return a value.<br>• OUT specifies the argument is for output only. |
| argtype | The data types of an argument. An argument type may be a base data type, a copy of the type of an existing column that uses %TYPE, or a user-defined type such as a nested table or an object type. A length cannot be specified for any base type, for example, specify VARCHAR2, not VARCHAR2(10).<br>The type of a column is referenced by writing tablename. columnname%TYPE. Using tablename. columnname%TYPE can sometimes help make a procedure independent from changes to the definition of a table. |
| DEFAULT value | The DEFAULT clause supplies a default value for an input argument if one is not supplied in the invocation. DEFAULT cannot be specified for arguments with modes IN OUT or OUT. |
| name | name is the name of the function or procedure. |

| Parameter | Description |
|---|---|
| restriction | The following keywords are accepted for compatibility and ignored:<br>• RNDS<br>• RNPS<br>• TRUST<br>• WNDS<br>• WNPS |

**Package body syntax**

Package implementation details reside in the package body. The package body may contain objects that are not visible to the package user. POLARDB compatible with Oracle supports the following syntax for the package body:

```
CREATE [ OR REPLACE ] PACKAGE BODY package_name
  { IS | AS }
  [ private_declaration; ] ...
  [ procedure_or_function_definition; ] ...
  [ package_initializer ]
END [ package_name ] ;
```

Where procedure or function definition :=

```
procedure definition | function definition
```

Where procedure definition :=

```
PROCEDURE proc name[ argument list ] [ options list ] { IS | AS }

procedure body END [ proc name ] ;
```

Where procedure_body :=

```
[ declaration; ] [, ... ] BEGIN

statement; [... ] [ EXCEPTION

{ WHEN exception [OR exception] [...]] THEN statement; } [...]

]
```

Where function definition :=

```
FUNCTION func_name [ argument_list ] RETURN rettype [DETERMINISTIC] [ options list ] { IS
 | AS }

function body END [ func name ] ;
```

```

```

Where function_body :=

```
[ declaration; ] [, ... ] BEGIN

statement; [... ]

[ EXCEPTION

{ WHEN exception [ OR exception ] [... ] THEN statement; } [...]

]
```

Where argument_list :=

```
( argument declaration [, ...] )
```

Where argument declaration :=

```
argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
```

Where options_list :=

```
option [ ... ]
```

Where option :=

```
COST execution cost ROWS result rows

SET config_param { TO value | = value | FROM CURRENT }
```

Where package initializer :=

```
BEGIN

statement; [... ] END;
```

### Parameters

| Parameter | Description |
|---|---|
| package_name | package_name is the name of the package for which this is the package body. There must be an existing package specification with this name. |

| Parameter | Description |
|---|---|
| private_declaration | private_declaration is an identifier of a private variable that can be accessed by any procedure or function within the package. There can be zero, one, or more private variables. private_declaration can be any of the following types:<br>• Variable declaration<br>• Record declaration<br>• Collection declaration<br>• REF CURSOR and cursor variable declaration<br>• TYPE definitions for records, collections, and REF CURSORs<br>• Exception<br>• Object variable declaration |
| proc_name | The name of the procedure being created. |
| declaration | A variable, type, or REF CURSOR declaration. |
| statement | An SPL program statement. Note that a DECLARE - BEGIN - END block is considered an SPL statement unto itself. Thus, the function body may contain nested blocks. |
| exception | An exception condition name, such as NO_DATA_FOUND and OHERS. |
| func_name | The name of the function being created. |
| rettype | The return data type, which may be any of the types listed for argtype. As for argtype, a length cannot be specified for rettype. |
| DETERMINISTIC | Includes DETERMINISTIC to specify that the function will always return the same result when given the same argument values. A DETERMINISTIC function cannot modify the database.<br>**Note:**<br>The DETERMINISTIC keyword is equivalent to the PostgreSQL IMMUTABLE option. |
| declaration | A variable, type, or REF CURSOR declaration. |
| argname | The name of a formal argument. The argument is referenced by this name within the procedure body. |

| Parameter | Description |
|---|---|
| IN \| IN OUT \| OUT | The argument mode.<br>• IN declares the argument for input only. This is the default mode.<br>• IN OUT allows the argument to receive a value or return a value.<br>• OUT specifies the argument is for output only. |
| argtype | The data types of an argument. An argument type may be a base data type, a copy of the type of an existing column that uses %TYPE, or a user-defined type such as a nested table or an object type. A length cannot be specified for any base type, for example, specify VARCHAR2, not VARCHAR2(10).<br>The type of a column is referenced by writing tablename.columnname%TYPE. Using tablename.columnname%TYPE can sometimes help make a procedure independent from changes to the definition of a table. |
| DEFAULT value | The DEFAULT clause supplies a default value for an input argument if one is not supplied in the procedure call. DEFAULT cannot be specified for arguments with modes IN OUT or OUT.<br>**Note:**<br>The following options are not compatible with Oracle databases. They are extensions to Oracle package syntax provided by POLARDB compatible with Oracle only. |
| STRICT | The STRICT keyword specifies that the function will not be executed if called with a NULL argument. Instead the function will return NULL. |
| LEAKPROOF | The LEAKPROOF keyword specifies that the function will not reveal any information about arguments, other than through a return value. |
| execution_cost | execution_cost specifies a positive number giving the estimated execution cost for the function, in units of cpu_operator_cost. If the function returns a set, this is the cost per returned row. Default value: 0.0025. |
| result_rows | result_rows is the estimated number of rows that the query planner expect the function to return. Default value: 1000. |

| Parameter | Description |
|---|---|
| SET | Use the SET clause to specify a parameter value for the duration of the function:<br>• config_param specifies the parameter name.<br>• value specifies the parameter value.<br>• FROM CURRENT guarantees that the parameter value is restored when the function ends. |
| package_initializer | The statements in the package_initializer are executed once per user's session when the package is first referenced. |

**Note:**

The STRICT, LEAKPROOF, COST, ROWS and SET keywords provide extended functionality for POLAR DB compatible with Oracle and are not supported by Oracle.

## 19.3 Create a package

A package is not an executable piece of code but a repository of code. When you use a package, you actually execute or make reference to an element within a package.

### Create the package specification

The package specification contains the definition of all the elements in the package that can be referenced from outside of the package. These are called the public elements of the package, and they act as the package interface. The following code sample is a package specification:

This code sample creates the emp_admin package specification. This package specification consists of two functions and two stored procedures. You can also add the OR REPLACE clause to the CREATE PACKAGE statement for convenience.

### Create the package body

The body of the package contains the actual implementation behind the package specification. For the above emp_admin package specification, create a package body which implements the specifications. The body will contain the implementation of the functions and stored procedures in the specification.

```
--
--  Package body for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
```

```
--
--  Function that queries the 'dept' table based on the department
--  number and returns the corresponding department name.
--
FUNCTION get_dept_name (
    p_deptno        IN NUMBER DEFAULT 10
)
RETURN VARCHAR2
IS
    v_dname         VARCHAR2(14);
BEGIN
    SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
    RETURN v_dname;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
        RETURN '';
END;
--
--  Function that updates an employee's salary based on the
--  employee number and salary increment/decrement passed
--  as IN parameters.  Upon successful completion the function
--  returns the new updated salary.
--
FUNCTION update_emp_sal (
    p_empno         IN NUMBER,
    p_raise         IN NUMBER
)
RETURN NUMBER
IS
    v_sal           NUMBER := 0;
BEGIN
    SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
    v_sal := v_sal + p_raise;
    UPDATE emp SET sal = v_sal WHERE empno = p_empno;
    RETURN v_sal;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
        RETURN -1;
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
        RETURN -1;
END;
--
--  Procedure that inserts a new employee record into the 'emp' table.
--
PROCEDURE hire_emp (
    p_empno         NUMBER,
    p_ename         VARCHAR2,
    p_job           VARCHAR2,
    p_sal           NUMBER,
    p_hiredate      DATE    DEFAULT sysdate,
    p_comm          NUMBER  DEFAULT 0,
    p_mgr           NUMBER,
    p_deptno        NUMBER  DEFAULT 10
)
AS
BEGIN
    INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
        VALUES(p_empno, p_ename, p_job, p_sal,
           p_hiredate, p_comm, p_mgr, p_deptno);
  END;
  --
  --  Procedure that deletes an employee record from the 'emp' table based
  --  on the employee number.
  --
  PROCEDURE fire_emp (
    p_empno      NUMBER
  )
  AS
  BEGIN
    DELETE FROM emp WHERE empno = p_empno;
  END;
END;
```

## 19.4 Reference a package

To reference the types, items, and subprograms that are declared within a package specification, use the dot notation. Example:

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
```

To invoke a function from the emp_admin package specification, execute the following SQL statement.

```
SELECT emp_admin.get_dept_name(10) FROM DUAL;
```

In the preceding statement, the get_dept_name function declared within the package emp_admin is invoked. The department number is passed as an argument to the function , which will return the name of the department. Here the value returned is ACCOUNTING, which corresponds to department number 10.

## 19.5 Use packages with user-defined types

The following example incorporates the various user-defined types discussed in earlier topics within the context of a package.

The package specification of emp_rpt shows the declaration of a record type, emprec_type, and a weakly typed REF CURSOR, emp_refcur, as publicly accessible along with two functions and two procedures. The open_emp_by_dept function returns the REF CURSOR type, EMP_REFCUR. The fetch_emp and close_refcur procedures both declare a weakly-typed REF CURSOR as a formal parameter.

```
CREATE OR REPLACE PACKAGE emp_rpt
```

```
IS
    TYPE emprec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR(10)
    );
    TYPE emp_refcur IS REF CURSOR;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2;
    FUNCTION open_emp_by_dept (
        p_deptno    IN emp.deptno%TYPE
    ) RETURN EMP_REFCUR;
    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    );
    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    );
END emp_rpt;
```

The package body shows the declaration of several private variables: a static cursor, dept_cur, a table type, depttab_typ, a table variable, t_dept, an integer variable, t_dept_max, and a record variable, r_emp.

```
CREATE OR REPLACE PACKAGE BODY emp_rpt
IS
    CURSOR dept_cur IS SELECT * FROM dept;
    TYPE depttab_typ IS TABLE of dept%ROWTYPE
        INDEX BY BINARY_INTEGER;
    t_dept          DEPTTAB_TYP;
    t_dept_max      INTEGER := 1;
    r_emp           EMPREC_TYP;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2
    IS
    BEGIN
        FOR i IN 1..t_dept_max LOOP
            IF p_deptno = t_dept(i).deptno THEN
                RETURN t_dept(i).dname;
            END IF;
        END LOOP;
        RETURN 'Unknown';
    END;

    FUNCTION open_emp_by_dept(
        p_deptno    IN emp.deptno%TYPE
    ) RETURN EMP_REFCUR
    IS
        emp_by_dept EMP_REFCUR;
    BEGIN
        OPEN emp_by_dept FOR SELECT empno, ename FROM emp
            WHERE deptno = p_deptno;
        RETURN emp_by_dept;
    END;

    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
        DBMS_OUTPUT.PUT_LINE('-----    -------');
        LOOP
            FETCH p_refcur INTO r_emp;
            EXIT WHEN p_refcur%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' || r_emp.ename);
        END LOOP;
    END;

    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        CLOSE p_refcur;
    END;
BEGIN
    OPEN dept_cur;
    LOOP
        FETCH dept_cur INTO t_dept(t_dept_max);
        EXIT WHEN dept_cur%NOTFOUND;
        t_dept_max := t_dept_max + 1;
    END LOOP;
    CLOSE dept_cur;
    t_dept_max := t_dept_max - 1;
END emp_rpt;
```

This package contains an initialization section that loads the private table variable, t_dept , by using the private static cursor, dept_cur. The t_dept private table variable serves as a department name lookup table in the get_dept_name function.

The open_emp_by_dept function returns a REF CURSOR variable for a result set of employee numbers and names for a given department. This REF CURSOR variable can then be passed to the fetch_emp procedure to retrieve and list the individual rows of the result set. Finally , the close_refcur procedure can be used to close the REF CURSOR variable associated with this result set.

The following anonymous block runs the package function and procedures. In the declaration section of the anonymous block, note the declaration of the v_emp_cur cursor variable, which uses EMP_REFCUR (the public REF CURSOR type of the package). v_emp_cur contains the pointer to the result set that is passed between the package function and procedures.

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    emp_rpt.fetch_emp(v_emp_cur);
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    emp_rpt.close_refcur(v_emp_cur);
END;
```

The result of this anonymous block is as follows:

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

The following anonymous block illustrates another means of achieving the same result. Instead of using the package procedures, fetch_emp and close_refcur, the logic of these programs is coded directly into the anonymous block. In the declaration section of the anonymous block, note the addition of the r_emp record variable declared by using EMPREC_TYPE (the public record type of the package).

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
    r_emp           emp_rpt.EMPREC_TYP;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH v_emp_cur INTO r_emp;
        EXIT WHEN v_emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' ||
            r_emp.ename);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    CLOSE v_emp_cur;
END;
```

The result of this anonymous block is as follows:

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
```

```
6 rows were retrieved
```

## 19.6 Drop a package

The syntax for deleting an entire package or the package body is as follows:

```
DROP PACKAGE [ BODY ] package_name;
```

If the BODY keyword is omitted, both the package specification and the package body are deleted, that is, the entire package is dropped. If the BODY keyword is specified, then only the package body is dropped. The package specification remains intact. package_name is the identifier of the package to be dropped.

The following statement will drop only the package body of emp_admin:

```
DROP PACKAGE BODY emp_admin;
```

The following statement will drop the entire emp_admin package:

```
DROP PACKAGE emp_admin;
```

# 20 Custom parameters

| Name | Valid value | Unit | Description |
|---|---|---|---|
| autovacuum_analyze_scale_factor | [0-1] | | The number of tuple inserts, updates, or deletes prior to analyze as a fraction of reltuples. |
| autovacuum_analyze_threshold | [0-2147483647] | | The minimum number of tuple inserts, updates, or deletes prior to analyze. |
| autovacuum_freeze_max_age | [200000000-1500000000] | | The age at which to autovacuum a table to prevent transaction ID wraparound. |
| autovacuum_max_workers | [5-20] | | Sets the maximum number of simultaneously running autovacuum worker processes. |
| autovacuum_vacuum_cost_delay | [1-100] | ms | The vacuum cost delay in milliseconds, for autovacuum. |
| autovacuum_vacuum_cost_limit | [-1-10000] | | The vacuum cost amount available before napping, for autovacuum. |
| autovacuum_vacuum_scale_factor | [0-1] | | The number of tuple updates or deletes prior to vacuum as a fraction of reltuples. |
| autovacuum_vacuum_threshold | [0-2147483647] | | The minimum number of tuple updates or deletes prior to vacuum. |

| Name | Valid value | Unit | Description |
|---|---|---|---|
| checkpoint_completion_target | [0-1] | | The time spent flushing dirty buffers during checkpoint, as fraction of checkpoint interval. |
| checkpoint_timeout | [30-86400] | s | Sets the maximum time between automatic WAL checkpoints. |
| checkpoint_warning | [30-2147483647] | s | Enables warnings if checkpoint segments are filled more frequently than this. |
| commit_delay | [0-100000] | | Sets the delay in microseconds between transaction commit and flushing WAL to disk. |
| commit_siblings | [0-1000] | | Sets the minimum concurrent open transactions before performing commit_delay. |
| default_statistics_target | [1-10000] | | Sets the default statistics target. |
| default_transaction_deferrable | [on\|off] | | Sets the default deferrable status of new transactions. |
| enable_bitmapscan | [on\|off] | | Enables the planner's use of bitmap-scan plans. |
| enable_gathermerge | [on\|off] | | Enables the planner's use of gather merge plans. |
| enable_hashagg | [on\|off] | | Enables the planner's use of hashed aggregation plans. |

| Name | Valid value | Unit | Description |
|---|---|---|---|
| enable_hashjoin | [on\|off] | | Enables the planner's use of hash join plans. |
| enable_indexonlyscan | [on\|off] | | Enables the planner's use of index-only-scan plans. |
| enable_indexscan | [on\|off] | | Enables the planner's use of index-scan plans. |
| enable_material | [on\|off] | | Enables the planner's use of materialization. |
| enable_mergejoin | [on\|off] | | Enables the planner's use of merge join plans. |
| enable_nestloop | [on\|off] | | Enables the planner's use of nested-loop join plans. |
| enable_parallel_append | [on\|off] | | Enables the planner's use of parallel append plans. |
| enable_parallel_hash | [on\|off] | | Enables the planner's use of parallel hash plans. |
| enable_partition_pruning | [on\|off] | | Enables plan-time and run-time partition pruning. |
| enable_partitionwise_aggregate | [on\|off] | | Enables partitionwise aggregation and grouping. |
| enable_partitionwise_join | [on\|off] | | Enables partitionwise join. |
| enable_seqscan | [on\|off] | | Enables the planner's use of sequential-scan plans. |

| Name | Valid value | Unit | Description |
|---|---|---|---|
| enable_sort | [on\|off] | | Enables the planner's use of explicit sort steps. |
| enable_tidscan | [on\|off] | | Enables the planner's use of TID scan plans. |
| gin_fuzzy_search_limit | [0-2147483647] | | Sets the maximum allowed result for exact search by GIN. |
| gin_pending_list_limit | [64-2147483647] | KB | Sets the maximum size of the pending list for GIN index. |
| auto_explain.log_analyze | [on\|off] | | Uses EXPLAIN ANALYZE for plan logging. |
| lock_timeout | [0-2147483647] | ms | Sets the maximum allowed duration of any wait for a lock. |
| log_autovacuum_min_duration | [-1-2147483647] | ms | Sets the minimum execution time above which autovacuum actions will be logged. |
| log_checkpoints | [on\|off] | | Logs each checkpoint. |
| log_connections | [on\|off] | | Logs each successful connection. |
| log_disconnections | [on\|off] | | Logs end of a session, including duration. |
| log_min_duration_statement | [-1-2147483647] | ms | Sets the minimum execution time above which statements will be logged. |
| log_statement | [none,ddl,mod,all] | | Sets the type of statements logged. |

| Name | Valid value | Unit | Description |
|---|---|---|---|
| log_temp_files | [-1-2147483647] | KB | Logs the use of temporary files larger than this number of kilobytes. |
| max_standby_archive_delay | [-1-2147483647] | ms | Sets the maximum delay before canceling queries when a hot standby server is processing archived WAL data. |
| max_standby_streaming_delay | [-1-2147483647] | ms | Sets the maximum delay before canceling queries when a hot standby server is processing streamed WAL data. |
| max_sync_workers_per_subscription | [0-262143] | | The maximum number of table synchronization workers per subscription. |
| min_parallel_index_scan_size | [0-715827882] | 8 KB | Sets the minimum amount of index data for a parallel scan. |
| min_parallel_table_scan_size | [0-715827882] | 8 KB | Sets the minimum amount of table data for a parallel scan. |
| old_snapshot_threshold | [-1-86400] | min | Time before a snapshot is too old to read pages changed after the snapshot was taken. |
| statement_timeout | [0-2147483647] | ms | Sets the maximum allowed duration of any statement. |
| track_activity_query_size | [100-102400] | Byte | Sets the size reserved for pg_stat_activity.query, in bytes. |

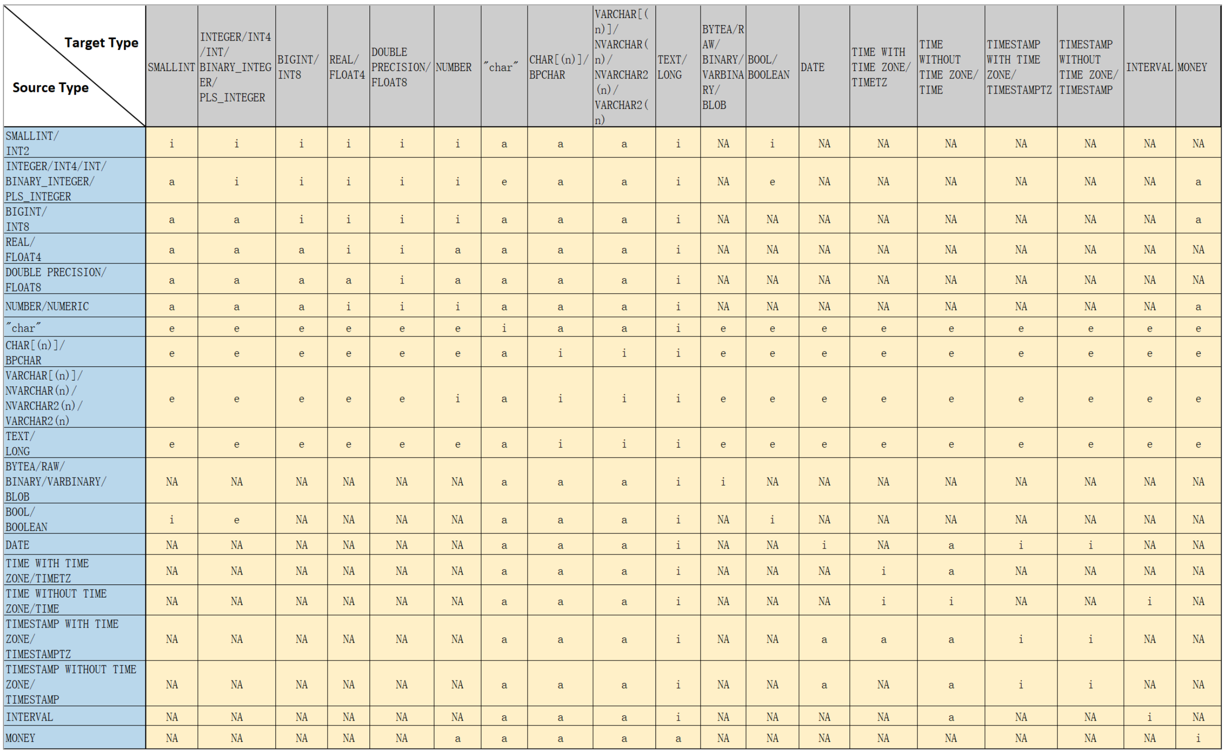
| Name | Valid value | Unit | Description |
|---|---|---|---|
| vacuum_cleanup_index_scale_factor | [0-10000000000] | | The number of tuple inserts prior to index cleanup as a fraction of reltuples. |
| vacuum_freeze_table_age | [150000000-2000000000] | | The age at which VACUUM scan whole table to freeze tuples . |
| wal_keep_segments | [0-100000] | | Sets the number of WAL files held for standby servers. |
| wal_level | [minimal,replica,logical] | | Sets the level of information written to the WAL. |
| auto_explain.log_buffers | [on\|off] | | Log buffers usage. |
| auto_explain.log_format | [text,xml,json,yaml] | | EXPLAIN format to be used for plan logging. |
| auto_explain.log_min_duration | [-1-2147483647] | ms | Sets the minimum execution time above which plans will be logged. |
| auto_explain.log_nested_statements | [on\|off] | | Log nested statements. |
| auto_explain.log_timing | [on\|off] | | Collects timing data and row counts. |
| auto_explain.log_triggers | [on\|off] | | Includes trigger statistics in plans. |
| auto_explain.log_verbose | [on\|off] | | Uses EXPLAIN VERBOSE for plan logging. |
| auto_explain.sample_rate | [0-1] | | The fraction of queries to process. |

| Name | Valid value | Unit | Description |
|---|---|---|---|
| parallel_setup_cost | [0-2147483647] | | Sets the planner's estimate of the cost of starting up worker processes for parallel query. |
| parallel_tuple_cost | [0-2147483647] | | Sets the planner's estimate of the cost of passing each tuple (row) from worker to master backend. |
| work_mem | [1024-1048576] | KB | Sets the maximum memory to be used for query workspaces. |
| idle_in_transaction_session_timeout | [0-2147483647] | ms | Sets the maximum allowed duration of any idling transaction. |

# 21 Implicit conversion rules

This topic lists the rules for implicit conversions of data types in Apsara PolarDB-O.

**Figure 21-1: Table of implicit conversion rules**



| Target Type<br>Source Type | SMALLINT | INTEGER/INT4/INT/BINARY_INTEGER/PLS_INTEGER | BIGINT/INT8 | REAL/FLOAT4 | DOUBLE PRECISION/FLOAT8 | NUMBER | "char" | CHAR[(n)]/BPCHAR | VARCHAR[(n)]/NVARCHAR(n)/NVARCHAR2(n)/VARCHAR2(n) | TEXT/LONG | BYTEA/RAW/BINARY/VARBINARY/BLOB | BOOL/BOOLEAN | DATE | TIME WITH TIME ZONE/TIMETZ | TIME WITHOUT TIME ZONE/TIME | TIMESTAMP WITH TIME ZONE/TIMESTAMPTZ | TIMESTAMP WITHOUT TIME ZONE/TIMESTAMP | INTERVAL | MONEY |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SMALLINT/INT2 | i | i | i | i | i | i | a | a | a | i | NA | i | NA | NA | NA | NA | NA | NA | NA |
| INTEGER/INT4/INT/BINARY_INTEGER/PLS_INTEGER | a | i | i | i | i | i | e | a | a | i | NA | e | NA | NA | NA | NA | NA | NA | a |
| BIGINT/INT8 | a | a | i | i | i | i | a | a | a | i | NA | NA | NA | NA | NA | NA | NA | NA | a |
| REAL/FLOAT4 | a | a | a | i | i | a | a | a | a | i | NA | NA | NA | NA | NA | NA | NA | NA | NA |
| DOUBLE PRECISION/FLOAT8 | a | a | a | a | i | a | a | a | a | i | NA | NA | NA | NA | NA | NA | NA | NA | NA |
| NUMBER/NUMERIC | a | a | a | i | i | i | a | a | a | i | NA | NA | NA | NA | NA | NA | NA | NA | a |
| "char" | e | e | e | e | e | e | i | a | a | i | e | e | e | e | e | e | e | e | e |
| CHAR[(n)]/BPCHAR | e | e | e | e | e | e | a | i | i | i | e | e | e | e | e | e | e | e | e |
| VARCHAR[(n)]/NVARCHAR(n)/NVARCHAR2(n)/VARCHAR2(n) | e | e | e | e | e | i | a | i | i | i | e | e | e | e | e | e | e | e | e |
| TEXT/LONG | e | e | e | e | e | e | a | i | i | i | e | e | e | e | e | e | e | e | e |
| BYTEA/RAW/BINARY/VARBINARY/BLOB | NA | NA | NA | NA | NA | NA | a | a | a | i | i | NA | NA | NA | NA | NA | NA | NA | NA |
| BOOL/BOOLEAN | i | e | NA | NA | NA | NA | a | a | a | i | NA | i | NA | NA | NA | NA | NA | NA | NA |
| DATE | NA | NA | NA | NA | NA | NA | a | a | a | i | NA | NA | i | NA | a | i | i | NA | NA |
| TIME WITH TIME ZONE/TIMETZ | NA | NA | NA | NA | NA | NA | a | a | a | i | NA | NA | NA | i | a | NA | NA | NA | NA |
| TIME WITHOUT TIME ZONE/TIME | NA | NA | NA | NA | NA | NA | a | a | a | i | NA | NA | NA | i | i | NA | NA | i | NA |
| TIMESTAMP WITH TIME ZONE/TIMESTAMPTZ | NA | NA | NA | NA | NA | NA | a | a | a | i | NA | NA | a | a | a | i | i | NA | NA |
| TIMESTAMP WITHOUT TIME ZONE/TIMESTAMP | NA | NA | NA | NA | NA | NA | a | a | a | i | NA | NA | a | NA | a | i | i | NA | NA |
| INTERVAL | NA | NA | NA | NA | NA | NA | a | a | a | i | NA | NA | NA | NA | a | NA | NA | i | NA |
| MONEY | NA | NA | NA | NA | NA | a | a | a | a | a | NA | NA | NA | NA | NA | NA | NA | NA | i |

- NA: indicates that implicit type conversion is not supported. Example:

```
explain verbose select CAST(c1 as timestamp) from t_smallint;
ERROR:  cannot cast type smallint to timestamp without time zone
```

- e: indicates that explicit type conversion is supported only by the CAST or :: syntax.

  Example:

```
create table t_int(c1 integer);
insert into t_int values(2);
select CAST(c1 as boolean) from t_int;
 c1
----
 t
insert into t_int values('true'::boolean);
ERROR:  column "c1" is of type integer but expression is of type boolean
```

- a: indicates that in addition to e, values can be implicitly assigned to the target column (which assigns values by using INSERT VALUES or UPDATE SET). Example:

```
create table t_int(c1 integer);
insert into t_int values(2);
select cast(c1 as smallint) from t_int;  -- ok
```

```
 c1
----
  2
insert into t_int values(3::smallint);  -- ok
```

- i: indicates that in addition to a and e, other implicit conversions are supported, such as expression parameters. Example:

```
-- case 1
CREATE OR REPLACE FUNCTION F_VARCHAR(arg1 VarChar) RETURN void
IS
BEGIN
   dbms_output.put_line(arg1);
   RETURN;
END;

SELECT F_VARCHAR(cast('10' as INTEGER)) FROM DUAL;   -- fail
SELECT F_VARCHAR(cast('10' as CHAR(10))) FROM DUAL;  -- ok
-- case 2
create table t_varchar(c1 varchar(10));
insert into t_varchar values(2);
explain verbose select sum(c1) from t_varchar;
                            QUERY PLAN
------------------------------------------------------------------------------
Aggregate  (cost=43.95..43.96 rows=1 width=32)
  Output: sum((c1)::numeric)
  ->  Seq Scan on public.t_varchar  (cost=0.00..29.40 rows=1940 width=14)
       Output: c1
```

**Note:**

The actual type of DATE data stored in an Apsara PolarDB-O database is determined by the parameter edb_redwood_date:

- edb_redwood_date=on (default): The data type is TIMESTAMP WITHOUT TIME ZONE, which is compatible with Oracle mode. For more information about the implicit type conversion rules, see Table of implicit conversion rules.

- edb_redwood_date=off: The data type is DATE, which is compatible with PostgreSQL mode. The following shows some of the implicit type conversion rules. For more information, see Table of implicit conversion rules.
    - When the type of the source data is DATE and the type of the target data is TIME WITHOUT TIME ZONE, the implicit conversion rule is "NA".
    - When the type of the source data is TIMESTAMP WITHOUT TIME ZONE and the type of the target data is DATE, the implicit conversion rule is "a".
    - When the type of the source data is TIMESTAMP WITH TIME ZONE (TIMESTAMPTZ) and the type of the target data is DATE, the implicit conversion rule is "a".
    - When the type of the source data is TIMESTAMP WITHOUT TIME ZONE (TIMESTAMP) and the type of the target data is DATE, the implicit conversion rule is "a".

# 22 Read and write external data files by using oss_fdw

Alibaba Cloud allows you to use the oss_fdw plug-in to load data in OSS to PolarDB compatible with Oracle databases and write data in PolarDB compatible with Oracle databases to OSS.

### oss_fdw parameters

The oss_fdw plug-in uses a method similar to other Foreign Data Wrapper (FDW) interfaces to encapsulate external data stored in OSS. You can use oss_fdw to read data stored in OSS. This process is similar to reading data tables. oss_fdw provides unique parameters to connect and parse file data in OSS.

**Note:**

- oss_fdw can read and write files of the following types in OSS: TEXT and CSV files as well as gzip-compressed TEXT and CSV files.
- The value of each parameter must be enclosed in double quotation marks (") and cannot contain any unnecessary spaces.

### CREATE SERVER parameters

- ossendpoint: the endpoint used to access OSS through the internal network, also known as the host.
- id oss: the ID of your OSS account.
- key oss: the key of your OSS account.
- bucket: the OSS bucket. You must create an OSS account before specifying this parameter.

The following fault tolerance parameters can be used for data import and export. If network connectivity is poor, you can adjust these parameters to ensure successful import and export.

- oss_connect_timeout: indicates the connection timeout period. Default value: 10. Unit: seconds.
- oss_dns_cache_timeout: indicates the DNS timeout period. Default value: 60. Unit: seconds.

- oss_speed_limit: indicates the minimum data transmission rate. Default value: 1. Unit: Kbit/s.
- oss_speed_time: indicates the maximum period when the data transmission rate is lower than the minimum value. Default value: 15. Unit: seconds.

If the default values of oss_speed_limit and oss_speed_time are used, a timeout error occurs when the transmission rate is smaller than 1 Kbit/s for 15 consecutive seconds.

**CREATE FOREIGN TABLE parameters**

- filepath: a file name that contains a path in OSS.
  - A file name contains a path but not a bucket name.
  - This parameter matches multiple files in the corresponding path in OSS. You can load multiple files to a database.
  - You can import files named in the format of filepath or filepath.x to a database. The values of x must be consecutive numbers starting from 1.

    For example, among the files named filepath, filepath.1, filepath.2, filepath.3, and filepath.5, the first four files are matched and imported. The filepath.5 file is not imported.
- dir: the virtual file directory in OSS.
  - dir must end with a forward slash (/).
  - All files (excluding subfolders and files in subfolders) in the virtual file directory specified by dir will be matched and imported to a database.
- prefix: the prefix of the path name corresponding to the data file. The prefix does not support regular expressions. Only one parameter among prefix, filepath, and dir can be specified at a time because they are mutually exclusive.
- format: the file format, which can only be csv.
- encoding: the file data encoding format. It supports common PostgreSQL encoding formats, such as UTF-8.
- parse_errors: the fault-tolerant parsing mode. If an error occurs during the parsing process, the entire row of data is ignored.
- delimiter: the column delimiter.
- quote: the quote character for files.
- escape: the escape character for files.
- null: sets the column matching a specified string to null.

- force_not_null: sets the value of a column to a non-null value. For example, force_not_null 'id' is used to set the values of the 'id' column to empty strings.
- compressiontype: specifies the format of the files to be read or written in OSS.
    - none: uncompressed text files. This is the default value.
    - gzip: The files to be read must be gzip compressed.
- compressionlevel: specifies the compression level of the compression format written to OSS. Valid values: 1 to 9. Default value: 6.

**Note:**

- You must specify filepath and dir in the OPTIONS parameter.
- You must specify either filepath or dir.
- The export mode only supports virtual folders, that is, only dir is supported.

**Export mode parameters for CREATE FOREIGN TABLE**

- oss_flush_block_size: the buffer size for the data written to OSS at a time. Default value: 32 MB. Valid values: 1 MB to 128 MB.
- oss_file_max_size: the maximum file size for the data written to OSS (subsequent data is written in another file when the maximum file size is exceeded). Default value: 1024 MB. Valid values: 8 MB to 4000 MB.
- num_parallel_worker: the number of parallel compression threads in which the OSS data is written. Valid values: 1 to 8. Default value: 3.

**Auxiliary functions**

FUNCTION oss_fdw_list_file (relname text, schema text DEFAULT 'public')

- Obtains the name and size of the OSS file that an external table matches.
- The unit of file size is Byte.

```
select * from oss_fdw_list_file('t_oss');
             name              |   size
--------------------------------+-----------
 oss_test/test.gz.1 | 739698350
 oss_test/test.gz.2 | 739413041
 oss_test/test.gz.3 | 739562048
(3 rows)
```

**Auxiliary features**

oss_fdw.rds_read_one_file: In read mode, it is used to specify a file to match the external table. If the file is specified, the external table only matches this file during data import.

Example: set oss_fdw.rds_read_one_file = 'oss_test/example16.csv.1';

```
set oss_fdw.rds_read_one_file = 'oss_test/test.gz.2';
select * from oss_fdw_list_file('t_oss');
           name             |  size
--------------------------------+-----------
 oss_test/test.gz.2  | 739413041
(1 rows)
```

### oss_fdw example

```
# Create a plug-in
create extension oss_fdw;
# Create a server
CREATE SERVER ossserver FOREIGN DATA WRAPPER oss_fdw OPTIONS
    (host 'oss-cn-hangzhou.aliyuncs.com', id 'xxx', key 'xxx', bucket 'mybucket');
# Create an OSS external table
CREATE FOREIGN TABLE ossexample
   (date text, time text, open float,
    high float, low float, volume int)
    SERVER ossserver
    OPTIONS ( filepath 'osstest/example.csv', delimiter ',' ,
       format 'csv', encoding 'utf8', PARSE_ERRORS '100');
# Create a table to load data to
create table example
       (date text, time text, open float,
        high float, low float, volume int)
# Load data from ossexample to example.
insert into example select * from ossexample;
# Result
# oss_fdw estimates the file size in OSS and formulates a query plan correctly.
explain insert into example select * from ossexample;
                        QUERY PLAN
---------------------------------------------------------------------
 Insert on example (cost=0.00..1.60 rows=6 width=92)
   -> Foreign Scan on ossexample (cost=0.00..1.60 rows=6 width=92)
        Foreign OssFile: osstest/example.csv.0
        Foreign OssFile Size: 728
(4 rows)
# Write the data in the example table to OSS.
insert into ossexample select * from example;
explain insert into ossexample select * from example;
                      QUERY PLAN
---------------------------------------------------------------
 Insert on ossexample (cost=0.00..16.60 rows=660 width=92)
   -> Seq Scan on example (cost=0.00..16.60 rows=660 width=92)
(2 rows)
```

### oss_fdw usage considerations

- oss_fdw is an external table plug-in developed based on the PostgreSQL FOREIGN TABLE framework.
- The data import efficiency is subject to the PolarDB compatible with Oracle cluster resources (CPU, I/O, memory, and MET) and OSS.
- To guarantee data import performance, ensure that PolarDB compatible with Oracle is in the same region as OSS. For more information, see Endpoints.

- If the error "oss endpoint userendpoint not in aliyun white list" is reported during reading of SQL statements for external tables, use the endpoints listed in Regions and endpoints. If the problem persists, submit a ticket.

### Error handling

When an import or export error occurs, the log displays the following error information:

- code: the HTTP status code of the request that has failed.
- error_code: the error code returned by OSS.
- error_msg: the error message returned by OSS.
- req_id: the UUID that identifies the request. If you cannot solve the problem, you can seek help from OSS development engineers by providing the req_id.

For more information about error types, see the following references. Timeout errors can be handled using oss_ext parameters.

- OSS help
-
- OSS error handling
- OSS error response

### ID and key encryption

If id and key parameters for CREATE SERVER are not encrypted, executing the select * from pg_foreign_server statement will display the information in plaintext. Your ID and key will be exposed. You can use symmetric encryption to hide the ID and key. Use different keys for different instances to further protect your information. However, to avoid incompatibility with earlier versions, do not add data types as you do in Greenplum.

Encrypted information:

```
postgres=# select * from pg_foreign_server ;
  srvname  | srvowner | srvfdw | srvtype | srvversion | srvacl |
                  srvoptions
-----------+----------+--------+---------+------------+--------
+------------------------------------------------------------------------------------------------
---------------------------------
 ossserver |       10 |  16390 |         |            |        | {host=oss-cn-hangzhou-zmf.aliyuncs.com,id
=MD5xxxxxxxx,key=MD5xxxxxxxx,bucket=067862}
```

The encrypted information is preceded by the MD5 hash value. The remainder of the total length divided by 8 is 3. Therefore, encryption is not performed again when the exported data is imported. But you cannot create the key and ID preceded by an MD5 hash value.

# 23 Global temporary tables

PolarDB databases compatible with Oracle support global temporary tables and native PostgreSQL local temporary tables.

### Syntax

```
CREATE GLOBAL TEMPORARY|TEMP TABLE table-name
    { column-definition [ , column-definition ] * }
[ ON COMMIT {DELETE | PRESERVE} ROWS ]
```

- The ON COMMIT DELETE ROWS clause is used to delete data from temporary tables after the current transaction is committed.
- The ON COMMIT PRESERVE ROWS clause is used to retain data in global temporary tables after the current transaction is committed.
- The ON COMMIT DROP clause is not supported.
- By default, the ON COMMIT DELETE ROWS clause is used if you do not use the ON COMMIT clause.

### Description

- All database sessions share the table definition of a global temporary table. When a session creates a global temporary table, other sessions can also use this global temporary table.
- The data stored in a global temporary table is private to the session that generates the data. Each session can only access its own data in the global temporary table.
- When a session exits, the data and underlying storage in the global temporary table that the session uses are cleared.
- You can join a global temporary table to other tables, create indexes on a global temporary table, and scan indexes on a global temporary table. The current version supports only B-tree indexes. The table-level and column-level statistics of the global temporary table is also private to the session that generates the data used in statistics. This optimizes the query plan of a query in the global temporary table.
- Global temporary tables support manual VACUUM and ANALYZE operations to clear junk data and collect statistics.

### Examples

```
create global temp table gtt1(a int primary key, b text); #Creates a global temporary
table named gtt1. By default, the global temporary table supports ON COMMIT DELETE
```

```
 ROWS. You can use ON COMMIT DELETE ROWS to delete all data from the global
temporary table after the current transaction is committed.
create global temporary table gtt2(a int primary key, b text) on commit delete rows; #
Creates a global temporary table named gtt2 and specifies ON COMMIT DELETE ROWS to
delete all data from the global temporary table after the current transaction is committed
.
create global temp table gtt3(a int primary key, b text) on commit PRESERVE rows;
 #Creates a global temporary table named gtt3 and specifies ON COMMIT PRESERVE
ROWS to retain all data from the global temporary table after the current transaction is
committed.
```

**Operations and maintenance**

PolarDB databases compatible with Oracle provide a group of functions used in operations and maintenance of global temporary tables.

- polar_gtt_attached_pid is used to view the sessions that are using a global temporary table. You can combine this function with other functions during operations and maintenance.
- polar_gtt_att_statistic is used to view the column-level statistics of a global temporary table.
- polar_gtt_relstats is used to view the table-level statistics of a global temporary table.

These functions work as plug-ins. You must create the plug-ins before you use these functions.

```
create extension polar_gtt;
```

If you want to delete a global temporary table, the global temporary table must be being used in the current session.

To delete the global temporary table, follow these steps:

1. Use polar_gtt_attached_pid to query the sessions that are using the global temporary table.
2. Use pg_backend_pid() to retrieve the process ID (pid) of the current session.
3. Use pg_terminate_backend(pid) to terminate non-current sessions.
4. Execute the DROP TABLE statement to delete the global temporary table.