Alibaba Cloud AnalyticDB for PostgreSQL

Application Migration

Issue: 20200629

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 individuals 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:** 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:** 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:** 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:** 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` *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 Migrate data from an Oracle database to an AnalyticDB for PostgreSQL instance

This topic describes how to migrate data from an Oracle database to an AnalyticDB for PostgreSQL instance, which is compatible with the syntax of Oracle SQL statements.

Syntax conversion by using Ora2Pg

Ora2Pg is an open source tool. You can use it to convert DDL, view, and package statements in Oracle to PostgreSQL-compatible syntax. For more information, see the Ora2Pg documentation.

Note:

However, you must manually correct converted SQL scripts because the PostgreSQL syntax version after script conversion is later than the kernel version of your AnalyticDB for PostgreSQL instance and the rules on which Ora2Pg depends for conversion may be missing or incorrect.

orafunc

AnalyticDB for PostgreSQL provides the orafunc plug-in, which offers functions that are compatible with the syntax of Oracle SQL statements. These functions can be used by AnalyticDB for PostgreSQL without modifications or conversions.

Before using this plug-in, execute the `create extension orafunc;` statement.

```
postgres=> create extension orafunc;
CREATE EXTENSION
```

The following table lists the orafunc-provided functions that are compatible with the syntax of Oracle SQL statements.

Table 1-1: orafunc-provided functions that are compatible with the syntax of Oracle SQL statements

Function	Description	Example

nvl(anyelement, anyelement)	• If the value of the first parameter is null, this function returns the value of the second parameter.
	• If the value of the first parameter is not null, this function returns the value of the first parameter.
	Note: The two parameters must be of the same data type.
		postgres=# select nvl(null,1);
		nvl
		----- 	
		1
		(1 row)
		
		postgres=# select nvl(0,1);
		nvl
		----- 	
		0
		(1 row)
		
		postgres=# select nvl(0,null);
		nvl
		----- 	
		0
		(1 row)

add_months(day date, value int) RETURNS date	This function adds the number of months (specified by using the second parameter) to a date (specified by using the first parameter) and returns a date.	postgres=# select add_months(current_date, 2); add_months ------------ 2019-08-31 (1 row)
last_day(value date)	This function returns the last day of the month for a specified date. The return value is a date.	postgres=# select last_day('2018-06 -01'); last_day ------------ 2018-06-30 (1 row)
next_day(value date, weekday text)	• The first parameter specifies a start date. • The second parameter specifies the day of a week, such as Friday. This function returns the date that represents a day of the second week since the start date, such as the date that represents the second Friday.	postgres=# select next_day(current_date, 'FRIDAY'); next_day ------------ 2019-07-05 (1 row)
next_day(value date, weekday integer)	• The first parameter specifies a start date. • The second parameter specifies a number that	postgres=# select next_day('2019-06 -22', 1); next_day ------------ 2019-06-23 (1 row)

The four Oracle functions listed in the following table are compatible with AnalyticDB for PostgreSQL. Therefore, they can be used in AnalyticDB for PostgreSQL without installing the orafunc plug-in.

Function	Description	Example
sinh(float)	This function returns a hyperbolic sine value.	postgres=# select sinh(0.1); sinh ------------------ 0.100166750019844 (1 row)
tanh(float)	This function returns a hyperbolic tangent value.	postgres=# select tanh(3); tanh ----------------- 0.99505475368673 (1 row)
cosh(float)	This function returns a hyperbolic cosine value.	postgres=# select cosh(0.2); cosh ----------------- 1.02006675561908 (1 row)

decode(expression, value , return [,value, return]... [, default])	This function searches for a value in expressions. If the value is found, the function returns it. Otherwise, the function returns the default value.	create table t1(id int, name varchar(20));
		postgres=# insert into t1 values(1,' alibaba');
		postgres=# insert into t1 values(2,' adb4pg');
		postgres=# select decode(id, 1, ' alibaba', 2, 'adb4pg', 'not found') from t1;
		case
		--------- 	
		alibaba
		adb4pg
		(2 rows)
		
		postgres=# select decode(id, 3, ' alibaba', 4, 'adb4pg', 'not found') from t1;
		case
		----------- 	
		not found
		not found
		(2 rows)

Mapping between Oracle data types and AnalyticDB for PostgreSQL data types

Oracle	AnalyticDB for PostgreSQL
VARCHAR2	VARCHAR or TEXT

Oracle	AnalyticDB for PostgreSQL
DATE	TIMESTAMP
LONG	TEXT
LONG RAW	BYTEA
CLOB	TEXT
NCLOB	TEXT
BLOB	BYTEA
RAW	BYTEA
ROWID	OID
FLOAT	DOUBLE PRECISION
DEC	DECIMAL
DECIMAL	DECIMAL
DOUBLE PRECISION	DOUBLE PRECISION
INT	INT
INTERGE	INTEGER
REAL	REAL
SMALLINT	SMALLINT
NUMBER	NUMERIC
BINARY_FLOAT	DOUBLE PRECISION
BINARY_DOUBLE	DOUBLE PRECISION
TIMESTAMP	TIMESTAMP
XMLTYPE	XML
BINARY_INTEGER	INTEGER
PLS_INTEGER	INTEGER
TIMESTAMP WITH TIME ZONE	TIMESTAMP WITH TIME ZONE
TIMESTAMP WITH LOCAL TIME ZONE	TIMESTAMP WITH TIME ZONE

Mapping between Oracle functions and AnalyticDB for PostgreSQL functions

Oracle	AnalyticDB for PostgreSQL
sysdate	current timestamp
trunc	trunc or date trunc

Oracle	AnalyticDB for PostgreSQL
dbms_output.put_line	raise
decode	case when or decode
NVL	coalesce

Conversion of data in PL/SQL

Procedural Language/SQL (PL/SQL) is a procedural extension provided by Oracle Corporation for SQL. PL/SQL enables SQL to have the features of general programming languages and can be used to implement complex business logic. PL/SQL corresponds to Procedural Language/PostgreSQL (PL/pgSQL) in AnalyticDB for PostgreSQL.

Packages

PL/pgSQL does not support packages. You must convert packages to schemas. In addition, all the procedures and functions in packages must be converted to functions supported by AnalyticDB for PostgreSQL.

Example:

```
create or replace package pkg is
...
end;
```

Conversion result:

```
create schema pkg;
```

- Variables defined in packages

  Local variables of procedures and functions remain unchanged, and global variables can be stored in temporary tables in AnalyticDB for PostgreSQL.
- Package initialization blocks

  You must remove package initialization blocks. If they cannot be removed, encapsulate them in functions and call the functions when necessary.

- Procedures and functions defined in packages

  Convert procedures and functions defined in packages to functions supported by AnalyticDB for PostgreSQL. Each function must be defined in the schema that correspond s to the involved package.

  For example, a package named pkg has the following function:

```
FUNCTION test_func (args int) RETURN int is
var number := 10;
BEGIN
... ...
END;
```

  The preceding function must be converted to the following function supported by AnalyticDB for PostgreSQL:

```
CREATE OR REPLACE FUNCTION pkg. test_func(args int) RETURNS int AS
$$

  ... ...

$$
 LANGUAGE plpgsql;
```

Procedures and functions

Convert package-specific and global procedures and functions in Oracle.

Example:

```
CREATE OR REPLACE FUNCTION test_func (v_name varchar2, v_version varchar2)
RETURN varchar2 IS
   ret varchar(32);
BEGIN
   IF v_version IS NULL THEN
      ret := v_name;
ELSE
   ret := v_name || '/' || v_version;
   END IF;
   RETURN ret;
END;
```

Conversion result:

```
CREATE OR REPLACE FUNCTION test_func (v_name varchar, v_version varchar)
RETURNS varchar AS
$$

DECLARE
   ret varchar(32);
BEGIN
   IF v_version IS NULL THEN
      ret := v_name;
ELSE
   ret := v_name || '/' || v_version;
```

```
    END IF;
    RETURN ret;
END;

$$
 LANGUAGE plpgsql;
```

Note the following when you convert a procedure or function:

- Convert the RETURN keyword to RETURNS.
- Use $\$ ... $\$ to encapsulate a function body.
- Pay attention to function language declarations.
- Convert a subprocedure to a function supported by AnalyticDB for PostgreSQL.

PL statements

- **FOR statements**

  In PL/SQL and PL/pgSQL, integer FOR loops with REVERSE statements work differently:

  - PL/SQL counts down from the second number to the first number.
  - PL/pgSQL counts down from the first number to the second number.

  Therefore, loop boundaries need to be exchanged during conversion. Example:

  ```
  FOR i IN REVERSE 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;
  ```

  Conversion result:

  ```
  FOR i IN REVERSE 3..1 LOOP
    RAISE '%' ,i;
  END LOOP;
  ```

- **PRAGMA statements**

  AnalyticDB for PostgreSQL does not support PRAGMA statements. Therefore, you must delete such statements.

- **Transaction processing**

  Functions of AnalyticDB for PostgreSQL do not support transaction control statements such as BEGIN, COMMIT, and ROLLBACK.

  These statements must be processed as follows:

  - Delete the transaction control statements in function bodies, and put them outside the function bodies.
  - Split functions based on COMMIT and ROLLBACK statements.

- **EXECUTE statements**

  AnalyticDB for PostgreSQL supports dynamic SQL statements that are similar to those in Oracle. The differences are as follows:

  - The dynamic SQL statements in AnalyticDB for PostgreSQL do not support the USING syntax. You must join parameters into SQL strings.
  - Database identifiers are packaged by using quote_ident, and numeric values are packaged by using quote_literal.

  Example:

```
EXECUTE 'UPDATE employees_temp SET commission_pct = :x' USING a_null;
```

  Conversion result:

```
EXECUTE 'UPDATE employees_temp SET commission_pct = ' || quote_literal(a_null);
```

- **PIPE ROW statements**

  Use the table functions in AnalyticDB for PostgreSQL to replace PIPE ROW statements.

  Example:

```
TYPE pair IS RECORD(a int, b int);
TYPE numset_t IS TABLE OF pair;

FUNCTION f1(x int) RETURN numset_t PIPELINED IS
DECLARE
   v_p pair;
BEGIN
   FOR i IN 1..x LOOP
     v_p.a := i;
     v_p.b := i+10;
     PIPE ROW(v_p);
   END LOOP;
   RETURN;
END;

select * from f1(10);
```

  Conversion result:

```
create type pair as (a int, b int);

create or replace function f1(x int) returns setof pair as
$$

declare
rec pair;
begin
   for i in 1..x loop
      rec := row(i, i+10);
      return next rec;
   end loop;
```

```
    return ;
end

$$
 language 'plpgsql';

select * from f1(10);
```

Notes:

- Convert a custom pair to a composite pair.
- You do not need to define the Table Of type. Replace it with Set Of in AnalyticDB for PostgreSQL.
- Convert a PIPE ROW statement to the following two statements:

  ```
  rec := row(i);
  return next rec;
  ```

  The preceding Oracle function can also be converted to the following statement:

  ```
  create or replace function f1(x int) returns setof record as
  $$

  declare
  rec record;
  begin
     for i in 1..x loop
        rec := row(i, i+10);
        return next rec;
     end loop;
     return ;
  end

  $$
  language 'plpgsql';
  ```

  In the second conversion method, you do not need to pre-define the NUMSET_T data type, which differs from the first conversion method. This requires you to specify the data type of the return value during a query, such as `select * from f1(10) as (a int, b int );`.

- **Exception handling**
  - Use the raise function to throw an exception.
  - After the exception is caught, the involved transaction cannot be rolled back. Rollback is only allowed outside user-defined functions.
  - For error codes supported in AnalyticDB for PostgreSQL, visit https://www.postgresql.org/docs/8.3/errcodes-appendix.html.

- **Functions with both return and out parameters**

  In AnalyticDB for PostgreSQL, a function cannot contain both a return parameter and an out parameter. Therefore, you must convert the return parameter to an out parameter.

  Example:

```
CREATE OR REPLACE FUNCTION test_func(id int, name varchar(10), out_id out int)
returns varchar(10)
AS $body$
BEGIN
    out_id := id + 1;
    return name;
end
$body$
LANGUAGE PLPGSQL;
```

  Conversion result:

```
CREATE OR REPLACE FUNCTION test_func(id int, name varchar(10), out_id out int,
out_name out varchar(10))
AS $body$
BEGIN
    out_id := id + 1;
    out_name := name;
end
$body$
LANGUAGE PLPGSQL;
```

  Then execute the `select * from test_func(1,'1') into rec;` statement to obtain the return value of the corresponding field from rec.

- **Quotation marks (') contained in variables in string connections**

  In the following example, the variable param2 is of the STRING data type. Assume that the value of this variable is `adb'-'pg`. If sql_str is directly used in AnalyticDB for

PostgreSQL, - is identified as an operator, which causes an error. Therefore, you must use the quote_literal function to convert the variable.

Example:

```
sql_str := 'select * from test1 where col1 = ' || param1 || ' and col2 = '''|| param2 || '''and col3 = 3';
```

Conversion result:

```
sql_str := 'select * from test1 where col1 = ' || param1 || ' and col2 = '|| quote_literal( param2) || 'and col3 = 3';
```

- **Obtain the number of days between two timestamps**

  Example:

  ```
  SELECT to_date('2019-06-30 16:16:16') - to_date('2019-06-29 15:15:15') + 1 INTO v_days from dual;
  ```

  Conversion result:

  ```
  SELECT extract('days' from '2019-06-30 16:16:16'::timestamp - '2019-06-29 15:15:15 '::timestamp + '1 days'::interval)::int INTO v_days;
  ```

PL data types

- **RECORD**

  Convert the RECORD data type to the composite data type in AnalyticDB for PostgreSQL.

  Example:

  ```
  TYPE rec IS RECORD (a int, b int);
  ```

  Conversion result:

  ```
  CREATE TYPE rec AS (a int, b int);
  ```

- **NESTED TABLE**

  - As a variable in PL, the NESTED TABLE data type can be converted to the ARRAY data type in AnalyticDB for PostgreSQL.

    Example:

    ```
    DECLARE
      TYPE Roster IS TABLE OF VARCHAR2(15);
      names Roster :=
      Roster('D Caruso', 'J Hamil', 'D Piro', 'R Singh');
    BEGIN
      FOR i IN names.FIRST .. names.LAST
      LOOP
         IF names(i) = 'J Hamil' THEN
    ```

```
      DBMS_OUTPUT.PUT_LINE(names(i));
    END IF;
  END LOOP;
END;
```

Conversion result:

```
create or replace function f1() returns void as
$$

declare
   names varchar(15)[] := '{"D Caruso", "J Hamil", "D Piro", "R Singh"}';
   len int := array_length(names, 1);
begin
   for i in 1..len loop
      if names[i] = 'J Hamil' then
         raise notice '%', names[i];
      end if;
   end loop;
   return ;
end

$$
 language 'plpgsql';

select f();
```

- If a nested table is used as the return value of a function, you can use the table function to replace it.

- **ASSOCIATIVE ARRAY**

  No replacement is available for this data type.

- **VARIABLE-SIZE ARRAY**

  Similar to the NESTED TABLE data type, the VARIABLE-SIZE ARRAY data type can be converted to the ARRAY data type.

- **Global variable**

  AnalyticDB for PostgreSQL does not support global variables. You can store all global variables of a package in a temporary table and define a function used to obtain them.

  Example:

```
create temporary table global_variables (
      id int,
      g_count int,
      g_set_id varchar(50),
      g_err_code varchar(100)
);

insert into global_variables values(0, 1, null, null);

CREATE OR REPLACE FUNCTION get_variable() returns setof global_variables AS

$$
```

```
DECLARE
   rec global_variables%rowtype;
BEGIN
   execute 'select * from global_variables' into rec;
   return next rec;
END;

$$
 LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION set_variable(in param varchar(50), in value anyelement
) returns void AS

$$

BEGIN
   execute 'update global_variables set ' ||  quote_ident(param) || ' = ' || quote_literal(
value);
END;

$$
 LANGUAGE plpgsql;
```

In the global_variables temporary table, the id field is the distribution key of the table. As AnalyticDB for PostgreSQL does not allow the modification of a distribution key, you must add the `tmp_rec record;` field to the table.

To modify a global variable, execute `select * from set_variable('g_error_code', 'error'::varchar) into tmp_rec;`.

To obtain a global variable, execute `select * from get_variable() into tmp_rec; error_code := tmp_rec.g_error_code;`.

SQL

- **Connect By**

  Connect By is used to process hierarchical queries in Oracle. No equivalent SQL statements can be found in AnalyticDB for PostgreSQL to replace a Connect By statement. You can use circular traversal by hierarchy to convert it.

  Example:

```
create table employee(
      emp_id numeric(18),
      lead_id numeric(18),
      emp_name varchar(200),
      salary numeric(10,2),
      dept_no varchar(8)
);
insert into employee values('1',0,'king','1000000.00','001');
insert into employee values('2',1,'jack','50500.00','002');
insert into employee values('3',1,'arise','60000.00','003');
insert into employee values('4',2,'scott','30000.00','002');
insert into employee values('5',2,'tiger','25000.00','002');
insert into employee values('6',3,'wudde','23000.00','003');
```

```
insert into employee values('7',3,'joker','21000.00','003');
insert into employee values('3',7,'joker','21000.00','003');
```

```
select emp_id,lead_id,emp_name,prior emp_name as lead_name,salary
    from employee
    start with  lead_id=0
    connect by prior emp_id =  lead_id
```

Conversion result:

```
create or replace function f1(tablename text, lead_id int, nocycle boolean) returns
setof employee as
$$

declare
   idx int := 0;
   res_tbl varchar(265) := 'result_table';
   prev_tbl varchar(265) := 'tmp_prev';
   curr_tbl varchar(256) := 'tmp_curr';

   current_result_sql varchar(4000);
   tbl_count int;

   rec record;
begin

   execute 'truncate ' || prev_tbl;
   execute 'truncate ' || curr_tbl;
   execute 'truncate ' || res_tbl;
   loop
      -- Query the current hierarchical result and insert it into the tmp_curr table.
      current_result_sql := 'insert into ' || curr_tbl || ' select t1.* from ' || tablename || ' t1';

      if idx > 0 then
         current_result_sql := current_result_sql || ', ' || prev_tbl || ' t2 where t1.lead_id = t2
.emp_id';
      else
         current_result_sql := current_result_sql || ' where t1.lead_id = ' || lead_id;
      end if;
      execute current_result_sql;

      -- If there is a loop, delete the data that has been traversed.
      if nocycle is false then
         execute 'delete from ' || curr_tbl || ' where (lead_id, emp_id) in (select lead_id,
emp_id from ' || res_tbl || ') ';
      end if;

      -- Exit if there is no data.
      execute 'select count(*) from ' || curr_tbl into tbl_count;
      exit when tbl_count = 0;

      -- Save data in the tmp_curr table to the result table.
      execute 'insert into ' || res_tbl || ' select * from ' || curr_tbl;
      execute 'truncate ' || prev_tbl;
      execute 'insert into ' || prev_tbl || ' select * from ' || curr_tbl;
      execute 'truncate ' || curr_tbl;
      idx := idx + 1;
   end loop;

   -- Return results.
   current_result_sql := 'select * from ' || res_tbl;
   for rec in execute current_result_sql loop
```

```
        return next rec;
    end loop;
    return;
end

$$
 language plpgsql;
```

- **Rownum**

  1. This statement is used to limit the size of a result set. You can use a limit statement to replace it.

     Example:

     ```
     select * from t where rownum < 10;
     ```

     Conversion result:

     ```
     select * from t limit 10;
     ```

  2. Use row_number() over() to generate rownum.

     Example:

     ```
     select rownum, * from t;
     ```

     Conversion result:

     ```
     select row_number() over() as rownum, * from t;
     ```

- **DUAL table**

  1. Remove dual.

     Example:

     ```
     select sysdate from dual;
     ```

     Conversion result:

     ```
     select current_timestamp;
     ```

  2. Create a DUAL table.

- **User-defined functions in SELECT**

  AnalyticDB for PostgreSQL allows you to call user-defined functions in SELECT. However , these functions cannot contain SQL statements. If they contain SQL statements, the following error message is displayed:

  ```
  ERROR: function cannot execute on segment because it accesses relation "public.t2" ( functions.c:155) (seg1 slice1 127.0.0.1:25433 pid=52153) (cdbdisp.c:1326)
  DETAIL:
  ```

```
SQL statement "select b from t2 where a = $1 "
```

To prevent this error message, convert the user-defined functions to SQL expressions or subqueries.

Example:

```
create or replace FUNCTION f1(arg int) RETURN int IS
   v int;
BEGIN
   select b into v from t2 where a = arg;
   return v;
END;

select a, f1(b) from t1;
```

Conversion result:

```
select t1.a, t2.b from t1, t2 where t1.b = t2.a;
```

- **Multi-table outer join (+)**

  AnalyticDB for PostgreSQL does not support the (+) syntax. You must convert it to the standard outer join syntax.

  Example:

  ```
  oracle
  select * from a,b where a.id=b.id(+)
  ```

  Conversion result:

  ```
  select * from a left join b on a.id=b.id
  ```

  If the (+) syntax involves a join of three tables, use wte to join two tables first, and then perform an outer join on the wte table and the table connected with +.

  Example:

  ```
  Select * from test1 t1, test2 t2, test3 t3 where t1.col1(+) between NVL(t2.col1, t3.col1) and NVL(t3.col1, t2.col1);
  ```

  Conversion result:

  ```
  with cte as (select t2.col1 as low, t2.col2, t3.col1 as high, t3.col2 as c2 from t2, t3)
  select * from t1 right outer join cte on t1.col1 between coalesce(cte.low, cte.high) and coalesce(cte.high,cte.low);
  ```

- **Merge Into**

  To convert the Merge Into syntax, execute an UPDATE statement in AnalyticDB for PostgreSQL first, and use the `GET DIAGNOSTICS rowcount := ROW_COUNT;` statement

to obtain the number of updated rows. If the number of updated rows is 0, execute an INSERT statement to insert data.

```
MERGE INTO test1 t1
        USING (SELECT t2.col1 col1, t3.col2 col2,
              FROM test2 t2, test3 t3) S
        ON S.col1 = 1 and S.col2 = 2
WHEN MATCHED THEN
          UPDATE
          SET test1.col1 = S.col1+1,
              test1.col2 = S.col2+2
WHEN NOT MATCHED THEN
          INSERT (col1, col2)
          VALUES
           (S.col1+1, S.col2+2);
```

Conversion result:

```
Update test1 t1 SET t1.col1 = test2.col1+1, test3.col2 = S.col2+2 where test2.col1 = 1
and test2.col2 = 2;
GET DIAGNOSTICS rowcount := ROW_COUNT;
if rowcount = 0 then
   insert into test1 values(test2.col1+1, test3.col2+2);
end if;
```

- **Sequence**

  Example:

  ```
  create sequence seq1;
  select seq1.nextval from dual;
  ```

  Conversion result:

  ```
  create SEQUENCE seq1;
  select nextval('seq1');
  ```

- **Cursor**

  - In Oracle, you can use the following statement to traverse cursors.

    Example:

    ```
    FUNCTION test_func() IS
       Cursor data_cursor IS SELECT * from test1;
    BEGIN
       FOR I IN data_cursor LOOP
          Do something with I;
    END LOOP;
    END;
    ```

    Conversion result:

    ```
    CREATE OR REPLACE FUNCTION test_func()
    AS $body$
    DECLARE
    data_cursor cursor for select * from test1;
    ```

```
I record;
BEGIN
   Open data_cursor;
   LOOP
     Fetch data_cursor INTO I;
    If not found then
         Exit;
    End if;
    Do something with I;
   END LOOP;
   Close data_cursor;
END;
$body$
LANGUAGE PLPGSQL;
```

- In Oracle, cursors with the same name can be opened in recursively called functions. However, this is not allowed in AnalyticDB for PostgreSQL. Equivalent statements in AnalyticDB for PostgreSQL must be in the format of For I in query.

  Example:

```
FUNCTION test_func(level IN numer) IS
   Cursor data_cursor IS SELECT * from test1;
BEGIN
If level > 5 then
      return;
  End if;

   FOR I IN data_cursor LOOP
      Do something with I;
      test_func(level + 1);
END LOOP;
END;
```

  Conversion result:

```
CREATE OR REPLACE FUNCTION test_func(level int) returns void
AS $body$
DECLARE
data_cursor cursor for select * from test1;
I record;
BEGIN
   If level > 5 then
      return;
   End if;
   For I in select * from test1 LOOP
     Do something with I;
     PERFORM test_func(level+1);
   END LOOP;
END;
$body$
LANGUAGE PLPGSQL;
```

2 Migrate data from a Teradata database to an AnalyticDB for PostgreSQL instance

AnalyticDB for PostgreSQL is compatible with Teradata syntax. This helps you migrate data in a Teradata database to an AnalyticDB for PostgreSQL instance after only a few modifications. This topic describes how to migrate data from a Teradata database to an AnalyticDB for PostgreSQL instance.

Data types

The core data types of AnalyticDB for PostgreSQL and Teradata are mutually compatible . Only a few data types need to be modified. The following table lists data types in AnalyticDB for PostgreSQL and Teradata.

Teradata	AnalyticDB for PostgreSQL
CHAR	CHAR
VARCHAR	VARCHAR
LONG VARCHAR	VARCHAR(64000)
VARBYTE(size)	BYTEA
BYTEINT	BYTEA
SMALLINT	SMALLINT
INTEGER	INTEGER
DECIMAL(size,dec)	DECIMAL(size,dec)
NUMERIC(precision,dec)	NUMERIC(precision,dec)
FLOAT	FLOAT
REAL	REAL
DOUBLE PRECISION	DOUBLE PRECISION
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP

Table creation statements

This section uses examples to describe the differences between AnalyticDB for PostgreSQL and Teradata.

To create a table in Teradata, execute the following statement:

```
CREATE MULTISET TABLE test_table,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO
    (
     first_column DATE FORMAT 'YYYYMMDD' TITLE 'COLUMN 1' NOT NULL,
     second_column INTEGER TITLE 'COLUMN 2' NOT NULL ,
     third_column CHAR(6) CHARACTER SET LATIN CASESPECIFIC TITLE 'COLUMN 3' NOT
NULL ,
     fourth_column CHAR(20) CHARACTER SET LATIN CASESPECIFIC TITLE 'COLUMN 4' NOT
NULL,
     fifth_column CHAR(1) CHARACTER SET LATIN CASESPECIFIC TITLE 'COLUMN 5' NOT NULL
,
     sixth_column CHAR(24) CHARACTER SET LATIN CASESPECIFIC TITLE 'COLUMN 6' NOT
NULL,
     seventh_column VARCHAR(18) CHARACTER SET LATIN CASESPECIFIC TITLE 'COLUMN 7'
NOT NULL,
     eighth_column DECIMAL(18,0) TITLE 'COLUMN 8' NOT NULL ,
     nineth_column DECIMAL(18,6) TITLE 'COLUMN 9' NOT NULL )
PRIMARY INDEX ( first_column ,fourth_column )
PARTITION BY RANGE_N(first_column  BETWEEN DATE '1999-01-01' AND DATE '2050-12-31
' EACH INTERVAL '1' DAY );

CREATE INDEX test_index (first_column, fourth_column) ON test_table;
```

To create a table in AnalyticDB for PostgreSQL, execute the following statement:

```
CREATE TABLE test_table
    (
     first_column DATE NOT NULL,
     second_column INTEGER NOT NULL ,
     third_column CHAR(6) NOT NULL ,
     fourth_column CHAR(20) NOT NULL,
     fifth_column CHAR(1) NOT NULL,
     sixth_column CHAR(24) NOT NULL,
     seventh_column VARCHAR(18) NOT NULL,
     eighth_column DECIMAL(18,0) NOT NULL ,
     nineth_column DECIMAL(18,6) NOT NULL )
DISTRIBUTED BY ( first_column ,fourth_column )
PARTITION BY RANGE(first_column)
(START (DATE '1999-01-01')  INCLUSIVE
END (DATE '2050-12-31')  INCLUSIVE
EVERY (INTERVAL '1 DAY' ) );

create index test_index on test_table(first_column, fourth_column);
```

Based on the preceding examples, the similarities and differences between table creation statements in AnalyticDB for PostgreSQL and Teradata are as follows:

- Core data types are compatible with each other and require no modifications.
- Both AnalyticDB for PostgreSQL and Teradata support distribution columns, but the syntax is different. The PRIMARY INDEX clause is used in Teradata, while DISTRIBUTED BY is used in AnalyticDB for PostgreSQL.

- Both AnalyticDB for PostgreSQL and Teradata support the PARTITION BY clause. Such clauses in AnalyticDB for PostgreSQL and Teradata have the same semantics but different syntax.
- Both AnalyticDB for PostgreSQL and Teradata allow you to create indexes on tables, but the syntax used is different.
- AnalyticDB for PostgreSQL does not support the TITLE keyword, but allows you to add a comment to a specific column by executing the following statement: `COMMENT ON COLUMN table_name.column_name IS 'XXX';`.
- AnalyticDB for PostgreSQL cannot declare the encoding type when you define the CHAR or VARCHAR data type. You can use the `SET client_encoding = latin1;` statement to declare the encoding type.

Data import and export formats

Both AnalyticDB for PostgreSQL and Teradata allow you to import or export data in the .txt or .csv format. The difference lies in the separators used in data files.

- Teradata uses two-character separators.
- AnalyticDB for PostgreSQL uses single-character separators.

SQL statements

AnalyticDB for PostgreSQL is compatible with the syntax of most SQL statements in Teradata . You only need to modify the following syntax:

- **cast**

  In Teradata:

  ```
  cast(XXX as int format '999999')
  cast(XXX as date format 'YYYYMMDD')
  ```

  In AnalyticDB for PostgreSQL:

  ```
  cast(XXX as int)
  cast(XXX as date)
  ```

  AnalyticDB for PostgreSQL does not declare a format in a CAST statement.

  - For the `cast(XXX as int format '999999')` statement, you must write a function for replacement.
  - You do not need to modify the `cast(XXX as date format 'YYYYMMDD')` statement because AnalyticDB for PostgreSQL supports the `'YYYY-MM-DD'` format for a date.

- **QUALIFY**

  The QUALIFY keyword in Teradata is used to further filter the results of the sorting function based on user-specified conditions.

  Example:

```
SELECT itemid, sumprice, RANK() OVER (ORDER BY sumprice DESC)
    FROM (SELECT a1.item_id, SUM(a1.sale)
        FROM sales AS a1
        GROUP BY a1.itemID) AS t1 (itemid, sumprice)
    QUALIFY RANK() OVER (ORDER BY sum_price DESC) <=100;
```

  AnalyticDB for PostgreSQL does not support the QUALIFY keyword. You must change an SQL statement with this keyword to a nested subquery.

```
SELECT itemid, sumprice, rank from
(SELECT itemid, sumprice, RANK() OVER (ORDER BY sumprice DESC) as rank
    FROM (SELECT a1.item_id, SUM(a1.sale)
        FROM sales AS a1
        GROUP BY a1.itemID) AS t1 (itemid,sumprice)
)  AS a
where rank <=100;
```

- **MACRO**

  Teradata uses macro to execute a group of SQL statements. Example:

```
CREATE MACRO Get_Emp_Salary(EmployeeNo INTEGER) AS (
  SELECT
  EmployeeNo,
  NetPay
  FROM
  Salary
  WHERE EmployeeNo = :EmployeeNo;
);
```

  AnalyticDB for PostgreSQL does not support macro, but you can use the FUNCTION statement to implement the macro function of Teradata. Example:

```
CREATE OR REPLACE FUNCTION Get_Emp_Salary(
      EmployeeNo INTEGER,
      OUT EmployeeNo INTEGER,
      OUT NetPay FLOAT
) returns setof record AS
$$

      SELECT EmployeeNo,NetPay
      FROM Salary
      WHERE EmployeeNo = $1

$$
 LANGUAGE SQL;
```

3 Migrate data from Amazon Redshift to ApsaraDB AnalyticDB for PostgreSQL

This topic describes how to migrate data from Amazon Redshift to ApsaraDB AnalyticDB for PostgreSQL.

Overall procedure

A typical migration process is as follows:

1. Prepare resources: Amazon Redshift, Amazon S3, ApsaraDB AnalyticDB for PostgreSQL, and Alibaba Cloud OSS.
2. Import the data in Redshift to S3.
3. Use OSSImport to import data files in .csv format from S3 to OSS.
4. In AnalyticDB for PostgreSQL, create the required objects, including schemas, tables, views, and functions.
5. Import data from the OSS external table into AnalyticDB for PostgreSQL.

The following figure shows the general workflow:

Preparations on AWS

Prepare information for accessing the S3 service

- Access Key ID and Secret Access Key
- The endpoint of the bucket in S3, for example, **s3.ap-southeast-2.amazonaws.com**
- The bucket name, for example, **alibaba-hybrid-export**

Data format requirements for data export

- The data file must be in CSV format
- The size of the file to be exported cannot exceed 50 MB
- The order of the column values in the file is the same as the column order of the table creation statement
- Ideally, the number of files to be exported is the same as the number of segments in AnalyticDB for PostgreSQL or a multiple of the number of segments

Recommended Redshift UNLOAD command option

The following UNLOAD command is the recommended format for the Redshift UNLOAD option, which is compatible with AnalyticDB for PostgreSQL:

```
unload ('select * from test')
to 's3://xxx-poc/test_export_'
access_key_id '<Your access key id>'
secret_access_key '<Your access key secret>'
DELIMITER AS ',';
ADDQUOTES
ESCAPE
NULL AS 'NULL'
MAXFILESIZE 50 mb ;
```

Specifically, in an UNLOAD command, the following options are recommended:

```
DELIMITER AS ','
ADDQUOTES
ESCAPE
NULL AS 'NULL'
MAXFILESIZE 50 mb
```

Get the DDL statement of the object in the Redshift database

Export all DDL statements from AWS Redshift including, but not limited to, schema, table, function, and view.

Preparations on Alibaba Cloud

Prepare information about the Alibaba Cloud RAM user account

- The RAM user account ID
- The RAM user account password
- The RAM user account AccessKeyId
- The RAM user account AccessKeySecret (paired with the preceding AccessKeyId to form an AccessKey)

Prepare a bucket in OSS

Create a bucket in Alibaba Cloud OSS in the same region as the AWS S3 bucket (for example , the Sydney (ap-southeast-2) region).

After the OSS bucket is created, the Internet endpoint and VPC endpoint (that is, the intranet endpoint) of the bucket can be obtained from the OSS Console.

Download and install OSSImport

- Create an ECS instance in the same area as the OSS bucket, with a network bandwidth of 100Mbps. In the following example, an instance running Windows is created.

- Download and install the latest version of OSSImport.
- After you unzip the OSSImport package, the following folders and files are displayed.

```
ossimport
├─── bin
│   └─── ossimport2.jar  # The JAR package including master, worker, tracker, and
console modules
├─── conf
│   ├─── local_job.cfg   # Standalone job configuration file
│   └─── sys.properties  # Configuration file for the system running
├─── console.bat        # Windows command line, which can run distributed call-in tasks
├─── console.sh         # Linux command line, which can run distributed call-in tasks
├─── import.bat         # The configuration file for one-click import and execution in
Windows is the data migration job configured in conf/local_job.cfg, including start,
migration, validation, and retry
├─── import.sh          # The configuration file of one-click import and execution in Linux
 is the data migration job configured in conf/local_job.cfg, including start, migration,
validation, and retry
├─── logs               # Log directory
└─── README.md          # Description documentation. We recommend that you carefully
 read the documentation before using this feature
```

Migrate data files from S3 to OSS using OSSImport

Configure OSSImport

In the following example, OSSImport is used in the standalone deployment mode.

Edit conf/local_job.cfg file. In this example, only the parameter configuration that must be modified is provided. For detailed configuration instructions for OSSImport, see Architecture and configuration.

```
srcType=s3
srcAccessKey="your AWS Access Key ID"
srcSecretKey="your AWS Access Key Secret"
srcDomain=s3.ap-southeast-2.amazonaws.com
srcBucket=alibaba-hybrid-export
srcBucket=
destAccessKey="your Alibaba Cloud Access Key ID"
destSecretKey="your Alibaba Cloud Access Key Secret"
destDomain=http://oss-ap-southeast-2-internal.aliyuncs.com
destBucket=alibaba-hybrid-export-1
destPrefix=
isSkipExistFile=true
```

Start the OSSImport Migration Task

In the OSSImport stand-alone deployment mode, you can start the migration task by executing import.bat.

Monitor task status

During the data migration process, you can see the output in the command execution window. Additionally, you can review the usage of the network bandwidth through the resource manager.

In this example, because the ECS instance and the OSS bucket are deployed in the same region, the network speed between data uploading from the instance to the bucket is not limited. Notably, because data is downloaded from S3 to OSS through the Internet, the speed of data transfer between ECS and OSS is essentially the same as the speed of data transfer between S3 and ECS. In this case, the upload speed is limited by the download speed.

Failed task retry (optional)

Sub-tasks may fail due to network or other reasons. Failure Retry only retries failed tasks, and will not retry the successful tasks. To retry failed tasks, execute `console.bat retry` in cmd.exe under the instance.

Check the files migrated to the OSS Bucket (optional)

You can check files through the OSS Console. We also recommend using the **ossbrowser** client tool to view and modify files in the bucket.Dowload ossbrowser.

Scrub the csv files (optional)

Note:

If you want to scrub the data of your csv file, the following commands can be used.

- Replace `NULL` in the csv files with blank spaces.
- Replace `\,` with `,` in the csv files.

We recommend you perform data scrubbing operations on locally stored data. Specifically, you need to first download the data file to be scrubbed to ECS through the ossbrowser tool , and then scrub the data. After that, you need to upload the scrubbed data files to another newly created bucket (so as to be distinguished from the original CSV files). In later uses, when downloading the original csv files or uploading the scrubbed files, we recommend that ossbrowser use the intranet endpoint of OSS to reduce unnecessary charges to your account.

DDL conversion from Redshift to AnalyticDB for PostgreSQL

This section describes the preparations required before creating a AnalyticDB for PostgreSQL database object. Specifically, it describes how to convert the DDL statements

in Redshift syntax format to AnalyticDB for PostgreSQL syntax format. This section also describes the corresponding syntax conventions.

CREATE SCHEMA

The following statement is an example that conforms to the PostgreSQL syntax format, which you can save as **create schema.sql**

```
CREATE SCHEMA schema1
  AUTHORIZATION xxxpoc;
GRANT ALL ON SCHEMA schema1 TO xxxpoc;
GRANT ALL ON SCHEMA schema1 TO public;
COMMENT ON SCHEMA model IS 'for xxx migration  poc test';

CREATE SCHEMA oss_external_table
  AUTHORIZATION xxxpoc;
```

CREATE FUNCTION

Because Redshift provides some SQL functions of which the corresponding functions are not yet supported in HybridDB, you can choose to customize these functions or rewrite them. Specific examples are described as follows.

- Replace CONVERT_TIMEZONE(a,b,c) with following code:

  ```
  timezone(b, timezone(a,c))
  ```

- Replace GETDATE() with following code:

  ```
  current_timestamp(0):timestamp
  ```

- Replace and optimize user defined functions (UDFs).

  For example, a SQL function of Redshift is as follows:

  ```
  CREATE OR REPLACE FUNCTION public.f_jdate(dt timestamp without time zone)
  RETURNS character varying AS
  '     from datetime import timedelta, datetime
        if dt.hour < 4:
              d = timedelta(days=-1)
              dt = dt + d
        return str(dt.date())'
  LANGUAGE plpythonu IMMUTABLE;
  COMMIT;
  ```

```
```

Replace the preceding function with the following SQL statement:

```
to_char(a - interval '4 hour', 'yyyy-mm-dd')
```

- Other Redshift standard SQL functions.

  In your actual scenario, we recommend that you query the standard SQL function library of PostgreSQL at Functions and Operators in PostgreSQL8.2. In doing so, you can determine which functions you need to manually modify and implement so that they are compatible with AnalyticDB for PostgreSQL. The following is a list of commonly used functions:

- - ISNULL()
  - DATEADD()
  - DATEDIFF()
  - REGEXP_COUNT()
  - LEFT()
  - RIGHT()

CREATE TABLE

- Change compression encoding. AnalyticDB for PostgreSQL does not support the full list of Redshift Compression Encoding. Compression encodings which are not supported are listed as follows:
  - BYTEDICT
  - DELTA
  - DELTA32K
  - LZO
  - MOSTLY8
  - MOSTLY16
  - MOSTLY32
  - RAW (no compression)
  - RUNLENGTH
  - TEXT255
  - TEXT32K
  - ZSTD

  ENCODE XXX should be removed and replaced with following option in CREATE TABLE statement:

  ```
  with (COMPRESSTYPE={ZLIB|QUICKLZ|RLE_TYPE|NONE})
  ```
- Change distribution keys. Redshift supports three types of distribution keys. For more information, see Distribution Styles. The following information indicates the rules you need to apply to modify the distribution keys so that the keys are compatible with AnalyticDB for PostgreSQL.
  - DISTSTYLE EVEN: Replace with `distributed randomly`
  - DISTKEY: Replace with `distributed by (colname1,...)`
  - ALL: Remove (not supported)

Change SORT key. Replace the COMPOUND or INTERLEAVED options in Redshift sort key clause `[ COMPOUND | INTERLEAVED ] SORTKEY (column_name [, ...] ) ]` with following clause:

```
with(APPENDONLY=true,ORIENTATION=column)
sortkey (volume);
```

Example 1

The following statement is a CREATE TABLE statement that conforms to Redshift syntax:

```
CREATE TABLE schema1.table1
(
   filed1 VARCHAR(100) ENCODE lzo,
   filed2 INTEGER DISTKEY,
   filed3 INTEGER,
   filed4 BIGINT ENCODE lzo,
   filed5 INTEGER,
)
INTERLEAVED SORTKEY
(
   filed1,
   filed2
);
```

After conversion, the CREATE TABLE statement that conforms to the AnalyticDB for PostgreSQL syntax is as follows:

```
CREATE TABLE schema1.table1
(
   filed1 VARCHAR(100) ,
   filed3 INTEGER,
   filed5 INTEGER
)
WITH(APPENDONLY=true,ORIENTATION=column,COMPRESSTYPE=zlib)
DISTRIBUTED BY (filed2)
SORTKEY
(
   filed1,
   filed2
)
```

Example 2

The following statement is a CREATE TABLE statement that conforms to Redshift syntax. It includes the ENCODE and SORTKEY options:

```
CREATE TABLE schema2.table2
(
   filed1 VARCHAR(50) ENCODE lzo,
   filed2 VARCHAR(50) ENCODE lzo,
   filed3 VARCHAR(20) ENCODE lzo,
)
DISTSTYLE EVEN
INTERLEAVED SORTKEY
(
   filed1
);
```

After conversion, the CREATE TABLE statement that conforms to the AnalyticDB for PostgreSQL syntax is as follows:

```
CREATE TABLE schema2.table2
```

```
(
   filed1 VARCHAR(50),
   filed2 VARCHAR(50),
   filed3 VARCHAR(20),
)
WITH(APPENDONLY=true, ORIENTATION=column, COMPRESSTYPE=zlib)
DISTRIBUTED randomly
SORTKEY
(
   filed1
);
```

CREATE VIEW

Similar to the CREATE TABLE statements in the preceding section, if you need to use a CREATE VIEW statement, you need to first convert the statement so that it conforms to the AnalyticDB for PostgreSQL syntax.

Create and Configure a AnalyticDB for PostgreSQL instance

For more information, see:

- #unique_6
- #unique_7
- #unique_8

Create Database Objects

Follow the instructions in #unique_9, and use **psql** or **pgAdmin III 1.6.3** to connect to an instance.

Then, modify the DDL statements in Redshift syntax to DDL statements that conform to the AnalyticDB for PostgreSQL syntax, and then execute these DDL statements to create database objects.

CREATE EXTERNAL TABLE

AnalyticDB for PostgreSQL supports parallel import from OSS and export to OSS through external tables (which is called the gpossext function). It can also compress external table files in gzip format to reduce the storage space and the costs. The gpossext function can read or write text and csv files, or text and csv files in gzip format. For more information, see #unique_10.

Import data by using INSERT INTO script

After external tables in OSS and database objects in AnalyticDB for PostgreSQL are created, you need to prepare an INSERT script to import data from the external tables to the target

tables in AnalyticDB for PostgreSQL. Then, you need to save the INSERT script as insert.sql, and then execute this file.

The format of the INSERT statement is `INSERT INTO <TABLE NAME> SELECT * FROM <OSS EXTERNAL TABLE NAME>;`.

Example:

```
INSERT INTO schema1.table1 SELECT * FROM oss_external_table.table1;
```

After the import is completed, you can use SELECT statements to verify the imported data and compare them with the source data.

Run a VACUUM script to defragment the database

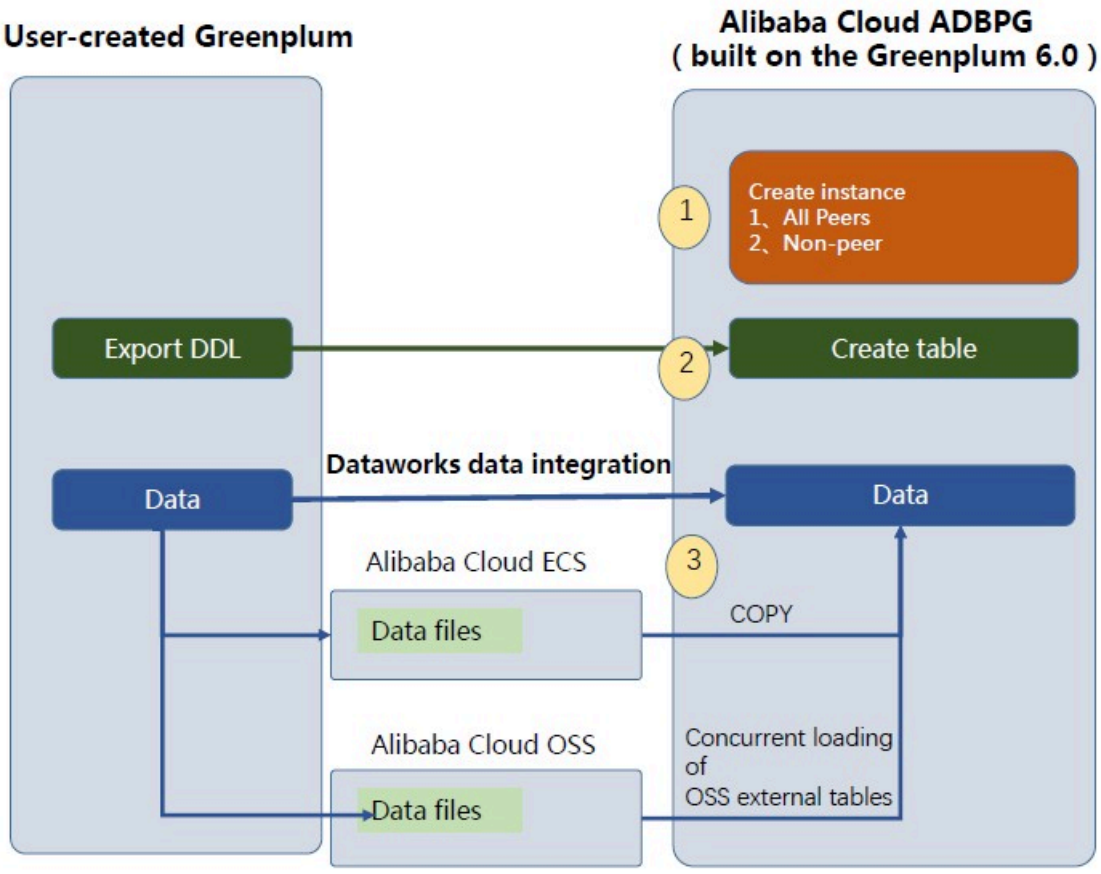
After the external tables in OSS are imported into AnalyticDB for PostgreSQL, you need to defragment the database by running `VACUUM` script. Then, you need to save the VACUUM script as vacuum.sql, and then execute this file. For more information about VACUUM, see VACUUM.

4 Migrate data from a user-created Greenplum database to AnalyticDB for PostgreSQL

AnalyticDB for PostgreSQL is fully compatible with the open-source Greenplum database and supports smooth migration of applications. This topic describes how to migrate data from a user-created Greenplum database to AnalyticDB for PostgreSQL.

Migration solution

AnalyticDB for PostgreSQL V6.0 is optimized by Alibaba Cloud based on the open source Greenplum 6.0 architecture. AnalyticDB for PostgreSQL V6.0 supports vector computing and transaction processing within a multi-coordinator node architecture and uses the same interfaces as open source Greenplum. A complete migration migrates both applications and data. You can seamlessly migrate your application and select from a variety of data migration solutions to migrate your database.



1. Create an AnalyticDB for PostgreSQL instance. For more information about instance specifications, see #unique_12.

2. Migrate Data Definition Language (DDL) statements from user-created Greenplum tables to the AnalyticDB for PostgreSQL instance to create table schemas.

```
pg_dumpall --gp-syntax --schema-only > db_dump.sql
```

3. Migrate your data by using one of the following solutions:

   - Migrate full data table by table on the cloud by using the data integration feature of DataWorks.
   - Export data from the user-created Greenplum database, upload data to Elastic Compute Service (ECS), and use the COPY statement of AnalyticDB for PostgreSQL to import data to the AnalyticDB for PostgreSQL instance. For more information, see #unique_13.
   - Export data from the user-created Greenplum database, upload data to Object Storage Service (OSS), and use the OSS external table feature to concurrently import data to the AnalyticDB for PostgreSQL instance. For more information, see #unique_10.

   You can use DataWorks to easily synchronize data and use the COPY statement or OSS to quickly synchronize data. The OSS external table feature costs less time than the COPY statement.