ALIBABA CLOUD

Alibaba Cloud

云原生数据仓库 AnalyticDB PostgreSQL 版

开发入门

文档版本：20220329

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.数据类型

AnalyticDB PostgreSQL支持丰富的数据类型，您还可以使用CREATE TYPE命令定义新的数据类型。

AnalyticDB PostgreSQL内建的数据类型

下表显示了AnalyticDB PostgreSQL内建的数据类型。

名称	别名	存储大小	范围	描述
bigint	int8	8 bytes	-9223372036854775808到9223372036854775807	大范围整数
bigserial	serial8	8 bytes	1 到 9223372036854775807	大的自动增量整数
bit [(n)]	无	n bits	bit 常量	固定长度位串
bit varying [(n)]	varbit	bit实际长度	bit 常量	可变长度位串
boolean	bool	1 byte	true/false, t/f, yes/no, y/n, 1/0	布尔值（true / false）
box	无	32 bytes	((x1,y1),(x2,y2))	平面中的矩形框 - 分配键列中不允许。
bytea	无	1 byte + binary string	八进制数序列	可变长度二进制字符串
character [(n)]	char [(n)]	1 byte + n	n长度字符串	定长的空白填充。
character varying [(n)]	varchar [(n)]	1 byte + string size	n长度字符串	受限的可变长度。
cidr	无	12 or 24 bytes	无	IPv4和IPv6网络
circle	无	24 bytes	<(x,y),r> (中心点和半径)	平面的圆 - 不允许在分配键列中。
date	无	4 bytes	4713 BC - 294,277 AD	日历日期（年，月，日）
decimal [(p, s)]	numeric [(p, s)]	variable	无限制	用户指定的精度，精确
double precision	float8 float	8 bytes	15位数字精度	可变精度，不精确

名称	别名	存储大小	范围	描述
inet	无	12 or 24 bytes	无	IPv4和IPv6主机和网络
integer	int, int4	4 bytes	-2.1E+09到 +2147483647	通常选择整数类型
interval [(p)]	无	12 bytes	-178000000 years - 178000000 years	时间跨度
json	无	1 byte + json size	Json字符串	不受限制的可变长度
lseg	无	32 bytes	((x1,y1),(x2,y2))	平面中的线段 - 分配键列中不允许。
macaddr	无	6 bytes	无	MAC 地址
money	无	8 bytes	-92233720368547758.08到 +92233720368547758.07	货币金额
path	无	16+16n bytes	[(x1,y1),...]	平面上的几何路径 - 分布关键列中不允许。
point	无	16 bytes	(x,y)	平面上的几何点 - 分布关键列中不允许。
polygon	无	40+16n bytes	((x1,y1),...)	在平面中封闭的几何路径 - 分配关键列中不允许。
real	float4	4 bytes	6位数字精度	可变精度，不准确
serial	serial4	4 bytes	1 到 2147483647	自动增量整数
smallint	int2	2 bytes	-32768 到 +32767	小范围整数
text	无	1 byte + string size	变长字符串	变量无限长
time [(p)] [without time zone]	无	8 bytes	00:00:00[.000000] - 24:00:00[.000000]	时间只有一天
time [(p)] with time zone	timetz	12 bytes	00:00:00+1359 - 24:00:00-1359	时间只有一天，带时区

名称	别名	存储大小	范围	描述
timestamp [(p)] [without time zone]	无	8 bytes	4713 BC - 294,277 AD	日期和时间
timestamp [(p)] with time zone	timestamptz	8 bytes	4713 BC - 294,277 AD	日期和时间，带时区
xml	无	1 byte + xml size	任意长度xml	变量无限长
uuid	无	32 bytes	无	6.0版本原生支持uuid数据类型；4.3版本需要先创建扩展插件，请参见使用UUID-OSSP

更多信息

详情请参考Pivotal Greenplum官方文档。

2.SQL语法

本文介绍AnalyticDB PostgreSQL版支持的SQL命令及语法参考。

> **说明** AnalyticDB PostgreSQL版Serverless版本暂不支持部分语法，具体限制，请参见Serverless版本。

- ABORT
- ALTER AGGREGATE
- ALTER CONVERSION
- ALTER DATABASE
- ALTER DOMAIN
- ALTER EXTERNAL TABLE
- ALTER FUNCTION
- ALTER GROUP
- ALTER INDEX
- ALTER OPERATOR
- ALTER RESOURCE QUEUE
- ALTER ROLE
- ALTER SCHEMA
- ALTER SEQUENCE
- ALTER TABLE
- ALTER TYPE
- ALTER USER
- ANALYZE
- BEGIN
- CHECKPOINT
- CLOSE
- CLUSTER
- COMMENT
- COMMIT
- COPY
- CREATE AGGREGATE
- CREATE CAST
- CREATE CONVERSION
- CREATE DATABASE
- CREATE DOMAIN
- CREATE EXTENSION
- CREATE EXTERNAL TABLE
- CREATE FUNCTION
- CREATE GROUP

- CREATE INDEX
- CREATE LIBRARY
- CREATE OPERATOR
- CREATE RESOURCE QUEUE
- CREATE ROLE
- CREATE RULE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TABLE AS
- CREATE TYPE
- CREATE USER
- CREATE VIEW
- DEALLOCATE
- DECLARE
- DELETE
- DROP AGGREGATE
- DROP CAST
- DROP CONVERSION
- DROP DATABASE
- DROP DOMAIN
- DROP EXTENSION
- DROP EXTERNAL TABLE
- DROP FUNCTION
- DROP GROUP
- DROP INDEX
- DROP LIBRARY
- DROP OPERATOR
- DROP OWNED
- DROP RESOURCE QUEUE
- DROP ROLE
- DROP RULE
- DROP SCHEMA
- DROP SEQUENCE
- DROP TABLE
- DROP TYPE
- DROP USER
- DROP VIEW
- END
- EXECUTE

- EXPLAIN
- FETCH
- GRANT
- INSERT
- LOAD
- LOCK
- MOVE
- PREPARE
- REASSIGN OWNED
- REINDEX
- RELEASE SAVEPOINT
- RESET
- REVOKE
- ROLLBACK
- ROLLBACK TO SAVEPOINT
- SAVEPOINT
- SELECT
- SELECT INTO
- SET
- SET ROLE
- SET SESSION AUTHORIZATION
- SET TRANSACTION
- SHOW
- START TRANSACTION
- TRUNCATE
- UPDATE
- VACUUM
- VALUES

ABORT

终止当前事务。

```
ABORT [WORK | TRANSACTION]
```

更多信息，请参见ABORT。

ALTER AGGREGATE

改变聚集函数的定义。

```
ALTER AGGREGATE name ( type [ , ... ] ) RENAME TO new_name
ALTER AGGREGATE name ( type [ , ... ] ) OWNER TO new_owner
ALTER AGGREGATE name ( type [ , ... ] ) SET SCHEMA new_schema
```

更多信息，请参见ALTER AGGREGATE。

ALTER CONVERSION

修改转换的定义。

```
ALTER CONVERSION name RENAME TO newname
ALTER CONVERSION name OWNER TO newowner
```

更多信息，请参见ALTER CONVERSION。

ALTER DATABASE

修改数据库属性。

```
ALTER DATABASE name [ WITH CONNECTION LIMIT connlimit ]
ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name RESET parameter
ALTER DATABASE name RENAME TO newname
ALTER DATABASE name OWNER TO new_owner
```

更多信息，请参见ALTER DATABASE。

ALTER DOMAIN

改变域的定义。

```
ALTER DOMAIN name { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name { SET | DROP } NOT NULL
ALTER DOMAIN name ADD domain_constraint
ALTER DOMAIN name DROP CONSTRAINT constraint_name [RESTRICT | CASCADE]
ALTER DOMAIN name OWNER TO new_owner
ALTER DOMAIN name SET SCHEMA new_schema
```

更多信息请参阅ALTER DOMAIN。

ALTER EXTERNAL TABLE

改变外部表的定义。

```
ALTER EXTERNAL TABLE name RENAME [COLUMN] column TO new_column
ALTER EXTERNAL TABLE name RENAME TO new_name
ALTER EXTERNAL TABLE name SET SCHEMA new_schema
ALTER EXTERNAL TABLE name action [, ... ]
```

更多信息，请参见ALTER EXTERNAL TABLE。

ALTER FUNCTION

改变函数的定义。

```
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
    action [, ... ] [RESTRICT]
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
    RENAME TO new_name
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
    OWNER TO new_owner
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
    SET SCHEMA new_schema
```

更多信息，请参见ALTER FUNCTION。

ALTER GROUP

改变角色名字或者成员信息。

```
ALTER GROUP groupname ADD USER username [, ... ]
ALTER GROUP groupname DROP USER username [, ... ]
ALTER GROUP groupname RENAME TO newname
```

更多信息，请参见ALTER GROUP。

ALTER INDEX

改变索引的定义。

```
ALTER INDEX name RENAME TO new_name
ALTER INDEX name SET TABLESPACE tablespace_name
ALTER INDEX name SET ( FILLFACTOR = value )
ALTER INDEX name RESET ( FILLFACTOR )
```

更多信息，请参见ALTER INDEX。

ALTER OPERATOR

改变操作符的定义。

```
ALTER OPERATOR name ( {lefttype | NONE} , {righttype | NONE} )
    OWNER TO newowner
```

更多信息，请参见ALTER OPERATOR。

ALTER RESOURCE QUEUE

修改资源队列的限制。

```
ALTER RESOURCE QUEUE name WITH ( queue_attribute=value [, ... ] )
```

更多信息，请参见ALTER RESOURCE QUEUE。

ALTER ROLE

修改一个数据库角色（用户或组）。

```
ALTER ROLE name RENAME TO newname
ALTER ROLE name SET config_parameter {TO | =} {value | DEFAULT}
ALTER ROLE name RESET config_parameter
ALTER ROLE name RESOURCE QUEUE {queue_name | NONE}
ALTER ROLE name [ [WITH] option [ ... ] ]
```

更多信息，请参见ALTER ROLE。

ALTER SCHEMA

改变模式的定义。

```
ALTER SCHEMA name RENAME TO newname
ALTER SCHEMA name OWNER TO newowner
```

更多信息，请参见ALTER SCHEMA。

ALTER SEQUENCE

改变序列生成器的定义。

```
ALTER SEQUENCE name [INCREMENT [ BY ] increment]
     [MINVALUE minvalue | NO MINVALUE]
     [MAXVALUE maxvalue | NO MAXVALUE]
     [RESTART [ WITH ] start]
     [CACHE cache] [[ NO ] CYCLE]
     [OWNED BY {table.column | NONE}]
ALTER SEQUENCE name SET SCHEMA new_schema
```

更多信息，请参见ALTER SEQUENCE。

ALTER TABLE

改变表的定义。

```
ALTER TABLE [ONLY] name RENAME [COLUMN] column TO new_column
ALTER TABLE name RENAME TO new_name
ALTER TABLE name SET SCHEMA new_schema
ALTER TABLE [ONLY] name SET
     DISTRIBUTED BY (column, [ ... ] )
   | DISTRIBUTED RANDOMLY
   | WITH (REORGANIZE=true|false)
ALTER TABLE [ONLY] name action [, ... ]
ALTER TABLE name
   [ ALTER PARTITION { partition_name | FOR (RANK(number))
   | FOR (value) } partition_action [...] ]
   partition_action
```

更多信息，请参见ALTER TABLE。

ALTER TYPE

改变数据类型的定义。

```
ALTER TYPE name
   OWNER TO new_owner | SET SCHEMA new_schema
```

更多信息，请参见ALTER TYPE。

ALTER USER

修改数据库角色（用户）的定义。

```
ALTER USER name RENAME TO newname
ALTER USER name SET config_parameter {TO | =} {value | DEFAULT}
ALTER USER name RESET config_parameter
ALTER USER name [ [WITH] option [ ... ] ]
```

更多信息，请参见ALTER USER。

ANALYZE

收集关于数据库的数据。

```
ANALYZE [VERBOSE] [ROOTPARTITION [ALL] ]
   [table [ (column [, ...] ) ]]
```

更多信息，请参见 ANALYZE。

BEGIN

启动事务块。

```
BEGIN [WORK | TRANSACTION] [transaction_mode]
      [READ ONLY | READ WRITE]
```

更多信息，请参见BEGIN 。

CHECKPOINT

强制事务记录检查点。

```
CHECKPOINT
```

更多信息，请参见CHECKPOINT。

CLOSE

关闭游标。

```
CLOSE cursor_name
```

更多信息，请参见CLOSE 。

CLUSTER

根据索引对磁盘上的堆存储表进行物理重新排序。不推荐使用该操作。

```
CLUSTER indexname ON tablename
CLUSTER tablename
CLUSTER
```

更多信息，请参见CLUSTER。

COMMENT

定义或者修改对一个对象的注释。

```
COMMENT ON
{ TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type [, ...]) |
  CAST (sourcetype AS targettype) |
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  FILESPACE object_name |
  FUNCTION func_name ([[argmode] [argname] argtype [, ...]]) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  OPERATOR op (leftoperand_type, rightoperand_type) |
  OPERATOR CLASS object_name USING index_method |
  [PROCEDURAL] LANGUAGE object_name |
  RESOURCE QUEUE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TABLESPACE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name }
IS 'text'
```

更多信息，请参见COMMENT。

COMMIT

提交当前事务。

```
COMMIT [WORK | TRANSACTION]
```

更多信息，请参见COMMIT。

COPY

在文件和表之间拷贝数据。

```
COPY table [(column [, ...])] FROM {'file' | STDIN}
     [ [WITH]
       [BINARY]
       [OIDS]
       [HEADER]
       [DELIMITER [ AS ] 'delimiter']
       [NULL [ AS ] 'null string']
       [ESCAPE [ AS ] 'escape' | 'OFF']
       [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
       [CSV [QUOTE [ AS ] 'quote']
            [FORCE NOT NULL column [, ...]]
       [FILL MISSING FIELDS]
       [[LOG ERRORS]
       SEGMENT REJECT LIMIT count [ROWS | PERCENT] ]
COPY {table [(column [, ...])] | (query)} TO {'file' | STDOUT}
      [ [WITH]
        [ON SEGMENT]
        [BINARY]
        [OIDS]
        [HEADER]
        [DELIMITER [ AS ] 'delimiter']
        [NULL [ AS ] 'null string']
        [ESCAPE [ AS ] 'escape' | 'OFF']
        [CSV [QUOTE [ AS ] 'quote']
             [FORCE QUOTE column [, ...]] ]
      [IGNORE EXTERNAL PARTITIONS ]
```

更多信息，请参见COPY 。

CREATE AGGREGATE

定义一个新的聚集函数。

```
CREATE [ORDERED] AGGREGATE name (input_data_type [ , ... ])
      ( SFUNC = sfunc,
        STYPE = state_data_type
        [, PREFUNC = prefunc]
        [, FINALFUNC = ffunc]
        [, INITCOND = initial_condition]
        [, SORTOP = sort_operator] )
```

更多信息，请参见CREATE AGGREGATE。

CREATE CAST

定义一个新的CAST。

```
CREATE CAST (sourcetype AS targettype)
       WITH FUNCTION funcname (argtypes)
       [AS ASSIGNMENT | AS IMPLICIT]
CREATE CAST (sourcetype AS targettype) WITHOUT FUNCTION
       [AS ASSIGNMENT | AS IMPLICIT]
```

更多信息，请参见CREATE CAST。

CREATE CONVERSION

定义一个新的编码转换。

```
CREATE [DEFAULT] CONVERSION name FOR source_encoding TO
     dest_encoding FROM funcname
```

更多信息，请参见CREATE CONVERSION。

CREATE DATABASE

创建一个新的数据库。

```
CREATE DATABASE name [ [WITH] [OWNER [=] dbowner]
                     [TEMPLATE [=] template]
                     [ENCODING [=] encoding]
                     [CONNECTION LIMIT [=] connlimit ] ]
```

更多信息，请参见CREATE DATABASE。

CREATE DOMAIN

定义一个新的域。

```
CREATE DOMAIN name [AS] data_type [DEFAULT expression]
       [CONSTRAINT constraint_name
       | NOT NULL | NULL
       | CHECK (expression) [...]]
```

更多信息，请参见CREATE DOMAIN。

CREATE EXTENSION

在数据库中注册一个EXTENSION。

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
  [ WITH ] [ SCHEMA schema_name ]
           [ VERSION version ]
           [ FROM old_version ]
           [ CASCADE ]
```

更多信息，请参见CREATE EXTENSION。

CREATE EXTERNAL TABLE

定义一张外部表。

```
CREATE [READABLE] EXTERNAL TABLE tablename
( columnname datatype [, ...] | LIKE othertable )
LOCATION ('ossprotocol')
FORMAT 'TEXT'
            [( [HEADER]
               [DELIMITER [AS] 'delimiter' | 'OFF']
               [NULL [AS] 'null string']
               [ESCAPE [AS] 'escape' | 'OFF']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
          | 'CSV'
            [( [HEADER]
               [QUOTE [AS] 'quote']
               [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [FORCE NOT NULL column [, ...]]
               [ESCAPE [AS] 'escape']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
[ ENCODING 'encoding' ]
[ [LOG ERRORS [INTO error_table]] SEGMENT REJECT LIMIT count
       [ROWS | PERCENT] ]
CREATE WRITABLE EXTERNAL TABLE table_name
( column_name data_type [, ...] | LIKE other_table )
LOCATION ('ossprotocol')
FORMAT 'TEXT'
               [( [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [ESCAPE [AS] 'escape' | 'OFF'] )]
          | 'CSV'
               [([QUOTE [AS] 'quote']
               [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [FORCE QUOTE column [, ...]] ]
               [ESCAPE [AS] 'escape'] )]
[ ENCODING 'encoding' ]
[ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
ossprotocol:
   oss://oss_endpoint prefix=prefix_name
    id=userossid key=userosskey bucket=ossbucket compressiontype=[none|gzip] async=[true|fa
lse]
ossprotocol:
   oss://oss_endpoint dir=[folder/[folder/]...]/file_name
    id=userossid key=userosskey bucket=ossbucket compressiontype=[none|gzip] async=[true|fa
lse]
ossprotocol:
   oss://oss_endpoint filepath=[folder/[folder/]...]/file_name
id=userossid key=userosskey bucket=ossbucket compressiontype=[none|gzip] async=[true|false]
```

更多信息，请参见CREATE EXTERNAL TABLE。

CREATE FUNCTION

定义一个新的函数。

```
CREATE [OR REPLACE] FUNCTION name
    ( [ [argmode] [argname] argtype [ { DEFAULT | = } defexpr ] [, ...] ] )
      [ RETURNS { [ SETOF ] rettype
        | TABLE ([{ argname argtype | LIKE other table }
          [, ...]])
        } ]
    { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINE
    | COST execution_cost
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol' } ...
    [ WITH ({ DESCRIBE = describe_function
           } [, ...] ) ]
```

更多信息，请参见CREATE FUNCTION。

CREATE GROUP

定义一个新的数据库角色。

```
CREATE GROUP name [ [WITH] option [ ... ] ]
```

更多信息，请参见CREATE GROUP。

CREATE INDEX

定义一个新的索引。

```
CREATE [UNIQUE] INDEX name ON table
       [USING btree|bitmap|gist]
       ( {column | (expression)} [opclass] [, ...] )
       [ WITH ( FILLFACTOR = value ) ]
       [TABLESPACE tablespace]
       [WHERE predicate]
```

更多信息，请参见CREATE INDEX。

CREATE LIBRARY

定义一个用户自定义软件表。

```
CREATE LIBRARY library_name LANGUAGE [JAVA] FROM oss_location OWNER ownername
CREATE LIBRARY library_name LANGUAGE [JAVA] VALUES file_content_hex OWNER ownername
```

更多信息，请参见CREATE LIBRARY。

CREATE OPERATOR

定义一个新的操作符。

```
CREATE OPERATOR name (
       PROCEDURE = funcname
       [, LEFTARG = lefttype] [, RIGHTARG = righttype]
       [, COMMUTATOR = com_op] [, NEGATOR = neg_op]
       [, RESTRICT = res_proc] [, JOIN = join_proc]
       [, HASHES] [, MERGES]
       [, SORT1 = left_sort_op] [, SORT2 = right_sort_op]
       [, LTCMP = less_than_op] [, GTCMP = greater_than_op] )
```

更多信息，请参见CREATE OPERATOR。

CREATE RESOURCE QUEUE

定义一个新的资源队列。

```
CREATE RESOURCE QUEUE name WITH (queue_attribute=value [, ... ])
```

更多信息，请参见CREATE RESOURCE QUEUE。

CREATE ROLE

定义一个新的数据库角色（用户或组）。

```
CREATE ROLE name [[WITH] option [ ... ]]
```

更多信息，请参见CREATE ROLE。

CREATE RULE

定义一个新的重写规则。

```
CREATE [OR REPLACE] RULE name AS ON event
  TO table [WHERE condition]
  DO [ALSO | INSTEAD] { NOTHING | command | (command; command
  ...) }
```

更多信息，请参见CREATE RULE。

CREATE SCHEMA

定义一个新的SCHEMA。

```
CREATE SCHEMA schema_name [AUTHORIZATION username]
   [schema_element [ ... ]]
CREATE SCHEMA AUTHORIZATION rolename [schema_element [ ... ]]
```

更多信息，请参见CREATE SCHEMA。

CREATE SEQUENCE

定义一个新的序列生成器。

```
CREATE [TEMPORARY | TEMP] SEQUENCE name
       [INCREMENT [BY] value]
       [MINVALUE minvalue | NO MINVALUE]
       [MAXVALUE maxvalue | NO MAXVALUE]
       [START [ WITH ] start]
       [CACHE cache]
       [[NO] CYCLE]
       [OWNED BY { table.column | NONE }]
```

更多信息，请参见CREATE SEQUENCE。

CREATE TABLE

定义一个新的表。

```
CREATE [[GLOBAL | LOCAL] {TEMPORARY | TEMP}] TABLE table_name (
[ { column_name data_type [ DEFAULT default_expr ]
   [column_constraint [ ... ]
[ ENCODING ( storage_directive [,...] ) ]
]
   | table_constraint
   | LIKE other_table [{INCLUDING | EXCLUDING}
                      {DEFAULTS | CONSTRAINTS}] ...}
   [, ... ] ]
   )
   [ INHERITS ( parent_table [, ... ] ) ]
   [ WITH ( storage_parameter=value [, ... ] )
   [ ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP} ]
   [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
   [ PARTITION BY partition_type (column)
       [ SUBPARTITION BY partition_type (column) ]
          [ SUBPARTITION TEMPLATE ( template_spec ) ]
       [...]
    ( partition_spec )
        | [ SUBPARTITION BY partition_type (column) ]
          [...]
    ( partition_spec
      [ ( subpartition_spec
           [(...)]
        ) ]
    )
```

> **说明** 目前Serverless版本不支持WITH子句，系统会根据数据类型自动选择最优算法。

更多信息，请参见CREATE TABLE。

CREATE TABLE AS

从查询的结果中定义一个新的表。

```
CREATE [ [GLOBAL | LOCAL] {TEMPORARY | TEMP} ] TABLE table_name
   [(column_name [, ...] )]
   [ WITH ( storage_parameter=value [, ... ] ) ]
   [ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP}]
   [TABLESPACE tablespace]
   AS query
   [DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY]
```

更多信息，请参见CREATE TABLE AS。

CREATE TRIGGER

定义一个新的触发器。

```
CREATE TRIGGER name {BEFORE | AFTER} {event [OR ...]}
       ON table [ FOR [EACH] {ROW | STATEMENT} ]
       EXECUTE PROCEDURE funcname ( arguments )
```

更多信息，请参见CREATE TRIGGER。

CREATE TYPE

定义一个新的类型。

```
CREATE TYPE name AS ( attribute_name data_type [, ... ] )
CREATE TYPE name AS ENUM ( 'label' [, ... ] )
CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [, RECEIVE = receive_function]
    [, SEND = send_function]
    [, TYPMOD_IN = type_modifier_input_function ]
    [, TYPMOD_OUT = type_modifier_output_function ]
    [, INTERNALLENGTH = {internallength | VARIABLE}]
    [, PASSEDBYVALUE]
    [, ALIGNMENT = alignment]
    [, STORAGE = storage]
    [, DEFAULT = default]
    [, ELEMENT = element]
    [, DELIMITER = delimiter] )
CREATE TYPE name
```

更多信息，请参见CREATE TYPE。

CREATE USER

定义一个默认带有LOGIN权限的数据库角色。

```
CREATE USER name [ [WITH] option [ ... ] ]
```

更多信息，请参见CREATE USER。

CREATE VIEW

定义一个新的视图。

```
CREATE [OR REPLACE] [TEMP | TEMPORARY] VIEW name
       [ ( column_name [, ...] ) ]
       AS query
```

更多信息，请参见CREATE VIEW。

DEALLOCATE

取消分配一个预编译的语句。

```
DEALLOCATE [PREPARE] name
```

更多信息，请参见DEALLOCATE。

DECLARE

定义一个游标。

```
DECLARE name [BINARY] [INSENSITIVE] [NO SCROLL] CURSOR
     [{WITH | WITHOUT} HOLD]
     FOR query [FOR READ ONLY]
```

更多信息，请参见DECLARE。

DELETE

从表中删除行。

```
DELETE FROM [ONLY] table [[AS] alias]
      [USING usinglist]
      [WHERE condition | WHERE CURRENT OF cursor_name ]
```

更多信息，请参见DELETE。

DROP AGGREGATE

删除聚集函数。

```
DROP AGGREGATE [IF EXISTS] name ( type [, ...] ) [CASCADE | RESTRICT]
```

更多信息，请参见DROP AGGREGATE。

DROP CAST

删除一个CAST。

```
DROP CAST [IF EXISTS] (sourcetype AS targettype) [CASCADE | RESTRICT]
```

更多信息，请参见DROP CAST。

DROP CONVERSION

删除一个转换。

```
DROP CONVERSION [IF EXISTS] name [CASCADE | RESTRICT]
```

更多信息，请参见DROP CONVERSION。

DROP DATABASE

删除一个数据库。

```
DROP DATABASE [IF EXISTS] name
```

更多信息，请参见DROP DATABASE。

DROP DOMAIN

删除一个域。

```
DROP DOMAIN [IF EXISTS] name [, ...]  [CASCADE | RESTRICT]
```

更多信息，请参见DROP DOMAIN。

DROP EXTENSION

从数据库中删除一个扩展。

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

更多信息，请参见DROP EXTENSION。

DROP EXTERNAL TABLE

删除一个外部表定义。

```
DROP EXTERNAL [WEB] TABLE [IF EXISTS] name [CASCADE | RESTRICT]
```

更多信息，请参见DROP EXTERNAL TABLE。

DROP FUNCTION

删除一个函数。

```
DROP FUNCTION [IF EXISTS] name ( [ [argmode] [argname] argtype
    [, ...] ] ) [CASCADE | RESTRICT]
```

更多信息，请参见DROP FUNCTION。

DROP GROUP

删除一个数据库角色。

```
DROP GROUP [IF EXISTS] name [, ...]
```

更多信息，请参见DROP GROUP。

DROP INDEX

删除一个索引。

```
DROP INDEX [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

更多信息，请参见DROP INDEX。

DROP LIBRARY

删除一个用户定义软件包。

```
DROP LIBRARY library_name
```

更多信息，请参见DROP LIBRARY。

DROP OPERATOR

删除一个操作符。

```
DROP OPERATOR [IF EXISTS] name ( {lefttype | NONE} ,
    {righttype | NONE} ) [CASCADE | RESTRICT]
```

更多信息，请参见DROP OPERATOR。

DROP OWNED

删除数据库角色所拥有的数据库对象。

```
DROP OWNED BY name [, ...] [CASCADE | RESTRICT]
```

更多信息，请参见DROP OWNED。

DROP RESOURCE QUEUE

删除一个资源队列。

```
DROP RESOURCE QUEUE queue_name
```

更多信息，请参见DROP RESOURCE QUEUE。

DROP ROLE

删除一个数据库角色。

```
DROP ROLE [IF EXISTS] name [, ...]
```

更多信息，请参见DROP ROLE。

DROP RULE

删除一个重写规则。

```
DROP RULE [IF EXISTS] name ON relation [CASCADE | RESTRICT]
```

更多信息，请参见DROP RULE。

DROP SCHEMA

删除一个SCHEMA。

```
DROP SCHEMA [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

更多信息，请参见DROP SCHEMA。

DROP SEQUENCE

删除一个序列。

```
DROP SEQUENCE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

更多信息，请参见DROP SEQUENCE。

DROP TABLE

删除一个表。

```
DROP TABLE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

更多信息，请参见DROP TABLE。

DROP TYPE

删除一个数据类型。

```
DROP TYPE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

更多信息，请参见DROP TYPE。

DROP USER

删除一个数据库角色。

```
DROP USER [IF EXISTS] name [, ...]
```

更多信息，请参见DROP USER。

DROP VIEW

删除一个视图。

```
DROP VIEW [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

更多信息，请参见DROP VIEW。

END

提交当前事务。

```
END [WORK | TRANSACTION]
```

更多信息，请参见END。

EXECUTE

执行一个已经准备好的SQL语句。

```
EXECUTE name [ (parameter [, ...] ) ]
```

更多信息，请参见EXECUTE。

EXPLAIN

展示语句的查询计划。

```
EXPLAIN [ANALYZE] [VERBOSE] statement
```

更多信息，请参见EXPLAIN。

FETCH

使用游标获取查询结果的行。

```
FETCH [ forward_direction { FROM | IN } ] cursorname
```

更多信息，请参见FETCH 。

GRANT

定义一个访问权限。

```
GRANT { {SELECT | INSERT | UPDATE | DELETE | REFERENCES |
TRIGGER | TRUNCATE } [,...] | ALL [PRIVILEGES] }
    ON [TABLE] tablename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]
GRANT { {USAGE | SELECT | UPDATE} [,...] | ALL [PRIVILEGES] }
    ON SEQUENCE sequencename [, ...]
    TO { rolename | PUBLIC } [, ...] [WITH GRANT OPTION]
GRANT { {CREATE | CONNECT | TEMPORARY | TEMP} [,...] | ALL
[PRIVILEGES] }
    ON DATABASE dbname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]
GRANT { EXECUTE | ALL [PRIVILEGES] }
    ON FUNCTION funcname ( [ [argmode] [argname] argtype [, ...]
] ) [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]
GRANT { USAGE | ALL [PRIVILEGES] }
    ON LANGUAGE langname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]
GRANT { {CREATE | USAGE} [,...] | ALL [PRIVILEGES] }
    ON SCHEMA schemaname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]
GRANT { CREATE | ALL [PRIVILEGES] }
    ON TABLESPACE tablespacename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]
GRANT parent_role [, ...]
    TO member_role [, ...] [WITH ADMIN OPTION]
GRANT { SELECT | INSERT | ALL [PRIVILEGES] }
    ON PROTOCOL protocolname
    TO username
```

更多信息，请参见GRANT。

INSERT

在表中创建新的行。

```
INSERT INTO table [( column [, ...] )]
   {DEFAULT VALUES | VALUES ( {expression | DEFAULT} [, ...] )
   [, ...] | query}
```

更多信息，请参见INSERT。

LOAD

加载或重新加载共享库文件。

```
LOAD 'filename'
```

更多信息，请参见 LOAD。

LOCK

锁住一张表。

```
LOCK [TABLE] name [, ...] [IN lockmode MODE] [NOWAIT]
```

更多信息，请参见LOCK。

MOVE

放置一个游标。

```
MOVE [ forward_direction {FROM | IN} ] cursorname
```

更多信息，请参见MOVE。

PREPARE

准备一个执行的语句。

```
PREPARE name [ (datatype [, ...] ) ] AS statement
```

更多信息，请参见PREPARE。

REASSIGN OWNED

改变数据库角色所拥有的数据库对象的所有权。

```
REASSIGN OWNED BY old_role [, ...] TO new_role
```

更多信息，请参见REASSIGN OWNED。

REINDEX

重新构建索引。

```
REINDEX {INDEX | TABLE | DATABASE | SYSTEM} name
```

更多信息，请参见REINDEX。

RELEASE SAVEPOINT

销毁一个之前定义过的SAVEPOINT。

```
RELEASE [SAVEPOINT] savepoint_name
```

更多信息，请参见RELEASE SAVEPOINT。

RESET

恢复系统配置参数的值为默认值。

```
RESET configuration_parameter
RESET ALL
```

更多信息，请参见RESET。

REVOKE

撤销访问权限。

```
REVOKE [GRANT OPTION FOR] { {SELECT | INSERT | UPDATE | DELETE
       | REFERENCES | TRIGGER | TRUNCATE } [,...] | ALL [PRIVILEGES] }
       ON [TABLE] tablename [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]
REVOKE [GRANT OPTION FOR] { {USAGE | SELECT | UPDATE} [,...]
       | ALL [PRIVILEGES] }
       ON SEQUENCE sequencename [, ...]
       FROM { rolename | PUBLIC } [, ...]
       [CASCADE | RESTRICT]
REVOKE [GRANT OPTION FOR] { {CREATE | CONNECT
       | TEMPORARY | TEMP} [,...] | ALL [PRIVILEGES] }
       ON DATABASE dbname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]
REVOKE [GRANT OPTION FOR] {EXECUTE | ALL [PRIVILEGES]}
       ON FUNCTION funcname ( [[argmode] [argname] argtype
                             [, ...]] ) [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]
REVOKE [GRANT OPTION FOR] {USAGE | ALL [PRIVILEGES]}
       ON LANGUAGE langname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [ CASCADE | RESTRICT ]
REVOKE [GRANT OPTION FOR] { {CREATE | USAGE} [,...]
       | ALL [PRIVILEGES] }
       ON SCHEMA schemaname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]
REVOKE [GRANT OPTION FOR] { CREATE | ALL [PRIVILEGES] }
       ON TABLESPACE tablespacename [, ...]
       FROM { rolename | PUBLIC } [, ...]
       [CASCADE | RESTRICT]
REVOKE [ADMIN OPTION FOR] parent_role [, ...]
       FROM member_role [, ...]
       [CASCADE | RESTRICT]
```

更多信息，请参见REVOKE。

ROLLBACK

中止当前事务。

```
ROLLBACK [WORK | TRANSACTION]
```

更多信息，请参见ROLLBACK。

ROLLBACK TO SAVEPOINT

将当前事务回滚到某个SAVEPOINT。

```
ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] savepoint_name
```

更多信息，请参见ROLLBACK TO SAVEPOINT。

SAVEPOINT

在当前事务定义一个新的savepoint。

```
SAVEPOINT savepoint_name
```

更多信息，请参见SAVEPOINT。

SELECT

从表或者视图中检索行。

```
[ WITH with_query [, ...] ]
SELECT [ALL | DISTINCT [ON (expression [, ...])]]
  * | expression [[AS] output_name] [, ...]
  [FROM from_item [, ...]]
  [WHERE condition]
  [GROUP BY grouping_element [, ...]]
  [HAVING condition [, ...]]
  [WINDOW window_name AS (window_specification)]
  [{UNION | INTERSECT | EXCEPT} [ALL] select]
  [ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}] [, ...]]
  [LIMIT {count | ALL}]
  [OFFSET start]
  [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT] [...]]
```

更多信息，请参见SELECT。

SELECT INTO

从查询结果中定义一个新的表。

```
[ WITH with_query [, ...] ]
SELECT [ALL | DISTINCT [ON ( expression [, ...] )]]
    * | expression [AS output_name] [, ...]
    INTO [TEMPORARY | TEMP] [TABLE] new_table
    [FROM from_item [, ...]]
    [WHERE condition]
    [GROUP BY expression [, ...]]
    [HAVING condition [, ...]]
    [{UNION | INTERSECT | EXCEPT} [ALL] select]
    [ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}] [, ...]]
    [LIMIT {count | ALL}]
    [OFFSET start]
    [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT]
    [...]]
```

更多信息，请参见SELECT INTO。

SET

改变数据库配置参数的值。

```
SET [SESSION | LOCAL] configuration_parameter {TO | =} value |
    'value' | DEFAULT}
SET [SESSION | LOCAL] TIME ZONE {timezone | LOCAL | DEFAULT}
```

更多信息，请参见SET。

SET ROLE

设置当前会话当前角色的标识符。

```
SET [SESSION | LOCAL] ROLE rolename
SET [SESSION | LOCAL] ROLE NONE
RESET ROLE
```

更多信息，请参见SET ROLE。

SET SESSION AUTHORIZATION

设置会话角色标识符和当前会话当前角色的标识符。

```
SET [SESSION | LOCAL] SESSION AUTHORIZATION rolename
SET [SESSION | LOCAL] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

更多信息，请参见SET SESSION AUTHORIZATION。

SET TRANSACTION

设置当前事务的特征。

```
SET TRANSACTION [transaction_mode] [READ ONLY | READ WRITE]
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode
     [READ ONLY | READ WRITE]
```

更多信息，请参见SET TRANSACTION。

SHOW

显示当前系统配置参数的值。

```
SHOW configuration_parameter
SHOW ALL
```

更多信息，请参见SHOW。

START TRANSACTION

开始一个事务块。

```
START TRANSACTION [SERIALIZABLE | READ COMMITTED | READ UNCOMMITTED]
                  [READ WRITE | READ ONLY]
```

更多信息，请参见START TRANSACTION。

TRUNCATE

清空表的所有行。

```
TRUNCATE [TABLE] name [, ...] [CASCADE | RESTRICT]
```

更多信息，请参见TRUNCATE。

UPDATE

更新表的行。

```
UPDATE [ONLY] table [[AS] alias]
   SET {column = {expression | DEFAULT} |
   (column [, ...]) = ({expression | DEFAULT} [, ...])} [, ...]
   [FROM fromlist]
   [WHERE condition | WHERE CURRENT OF cursor_name ]
```

更多信息，请参见UPDATE。

VACUUM

垃圾收集和选择性分析数据库。

```
VACUUM [FULL] [FREEZE] [VERBOSE] [table]
VACUUM [FULL] [FREEZE] [VERBOSE] ANALYZE
              [table [(column [, ...] )]]
```

更多信息，请参见VACUUM。

VALUES

计算一组行。

```
VALUES ( expression [, ...] ) [, ...]
   [ORDER BY sort_expression [ASC | DESC | USING operator] [, ...]]
   [LIMIT {count | ALL}] [OFFSET start]
```

更多信息，请参见VALUES。

3.用户权限管理

本文介绍了用户和权限管理的方法。

用户管理

实例创建过程中，会提示用户指定初始用户名和密码，这个初始用户为“根用户”。实例创建好后，用户可以使用该根用户连接数据库。使用psql（PostgreSQL或Greenplum的客户端工具）连接数据库后，通过 `\du+` 命令可以查看所有用户的信息，示例如下：

> **注意** 除了根用户外，还有其他内部管理用户被创建。

```
postgres=> \du+
```

返回结果如下：

```
                                   List of roles
  Role name   |            Attributes             | Member of |  Description
--------------+-----------------------------------+-----------+---------------
 root_user    |                                   |           | rds_superuser
 ...
```

目前，AnalyticDB PostgreSQL版没有开放SUPERUSER权限，对应的是 RDS_SUPERUSER，这一点与云数据库RDS（PostgreSQL）中的权限体系一致。所以，根用户（如上面的示例中的root_user）具有RDS_SUPERUSER权限，这个权限属性只能通过查看用户的描述（Description）来识别。根用户具有如下权限：

- 具有CREATEROLE、CREATEDB和LOGIN权限，即可以用来创建数据库和用户，但没有SUPERUSER权限。
- 查看和修改其它非超级用户的数据表，执行SELECT、UPDATE、DELTE或更改所有者（Owner）等操作。
- 查看其它非超级用户的连接信息、撤销（Cancel）其SQL或终止（Kill）其连接。
- 执行CREATE EXTENSION和DROP EXTENSION命令，创建和删除插件。
- 创建其他具有RDS_SUPERUSER权限的用户，示例如下：

  ```
  CREATE ROLE root_user2 RDS_SUPERUSER LOGIN PASSWORD 'xyz';
  ```

权限管理

用户可以在数据库（Database）、模式（Schema）、表等多个层次管理权限。赋予一个用户表上的读取权限，但收回修改权限，示例如下：

```
GRANT SELECT ON TABLE t1 TO normal_user1;
REVOKE UPDATE ON TABLE t1 FROM normal_user1;
REVOKE DELETE ON TABLE t1 FROM normal_user1;
```

参考文档

关于具体的用户与权限管理方法，请参见Managing Roles and Privileges。

4.使用DML插入、更新、删除数据

本文介绍在AnalyticDB PostgreSQL数据库中，如何使用DML语法插入、更新、删除数据。

插入行（INSERT）

> **说明** 如果要插入大量数据，推荐使用外部表或者COPY命令，相比INSERT性能更好。

使用 `INSERT` 命令在一个表中插入行。语法如下：

```
INSERT INTO table [( column [, ...] )]
   {DEFAULT VALUES | VALUES ( {expression | DEFAULT} [, ...] )
   [, ...] | query}
```

更多信息请参考INSERT。

示例：

插入单行数据。

```
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

在一条命令中插入多行数据。

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

使用标量表达式插入数据。

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod <
'2016-05-07';
```

更新行（UPDATE）

使用 `UPDATE` 命令在一个表中更新行。语法如下：

```
UPDATE [ONLY] table [[AS] alias]
   SET {column = {expression | DEFAULT} |
   (column [, ...]) = ({expression | DEFAULT} [, ...])} [, ...]
   [FROM fromlist]
   [WHERE condition | WHERE CURRENT OF cursor_name ]
```

更多信息请参考UPDATE。

使用限制：

- 分布键列不能被更新。
- 分区键列不能被更新。
- 不能在UPDATE语句中使用STABLE或VOLATILE函数。

- 不支持RETURNING子句。

示例：

将所有价格为5的产品更新价格为10。

```
UPDATE products SET price = 10 WHERE price = 5;
```

删除行（DELETE）

使用 `DELETE` 命令从一个表中删除行。语法如下：

```
DELETE FROM [ONLY] table [[AS] alias]
      [USING usinglist]
      [WHERE condition | WHERE CURRENT OF cursor_name ]
```

更多信息请参考DELETE。

使用限制：

- 不能在DELETE语句中使用STABLE或VOLATILE函数。
- 不支持RETURNING子句。

示例：

从产品表中删除所有价格为10的行。

```
DELETE FROM products WHERE price = 10;
```

从表中删除所有行。

```
DELETE FROM products;
```

截断表（TRUNCATE）

使用 `TRUNCATE` 命令可以快速地移除一个表中的所有行，TRUNCATE不扫描该表，因此它不会处理继承的子表或者ON DELETE的重写规则，只会截断命令中的表中的行。语法如下：

```
TRUNCATE [TABLE] name [, ...] [CASCADE | RESTRICT]
```

更多信息请参考TRUNCAT。

示例：

清空mytable表中的所有行。

```
TRUNCATE mytable;
```

5.使用INSERT ON CONFLICT覆盖写入数据

本文介绍在AnalyticDB PostgreSQL版数据库中，如何使用INSERT ON CONFLICT语法覆盖写入数据。

针对数据写入时有主键冲突的情况，INSERT ON CONFILICT语法可以将冲突主键的INSERT行为转换为UPDATE行为，从而实现冲突主键的覆盖写入。该特性又称UPSERT覆盖写，与MySQL的REPLACE INTO类似。

注意事项

- 仅支持存储弹性模式实例，暂不支持Serverless版本实例。
- 仅支持行存表，不支持列存表（由于列存表不支持唯一索引，所以该特性无法支持列存表）。
- 仅V6.3.6.1及以上内核版本支持在分区表中使用。如何升级内核版本，请参见版本升级。
- 不支持在UPDATE的SET子句中更新分布列和主键列。
- 不支持在UPDATE的WHERE子句中使用子查询。
- 不支持Updatable View（可更新视图）。
- 不支持在同一条INSERT语句中对同一主键插入多条数据（国际SQL标准约束）。

SQL语法

覆盖写入语法基于INSERT语句，INSERT语句的语法大纲如下：

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
    [ ON CONFLICT [ conflict_target ] conflict_action ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
其中，conflict_target为：
    ( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ opclass ] [, ...
] )
其中，conflict_action为：
    DO NOTHING
    DO UPDATE SET { column_name = { expression | DEFAULT } |
                    ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] )
                  } [, ...]
              [ WHERE condition ]
```

ON CONFLICT子句可以实现覆盖写入。该子句由conflict_target和conflict_action组成。

参数	说明
conflict_target	• conflict_action取值为Do Update时，conflict_target需要指定用来定义冲突的主键列或唯一索引列。 • conflick_action取值为Do Nothing时，conflict_target可省略。

参数	说明
conflict_action	用于指定冲突后需要执行的动作。取值说明： • DO NOTHING：如果conflict_target指定的列有冲突，则丢弃待插入的数据。 • DO UPDATE：如果conflict_target指定的列有冲突，则按照后面的UPDATE子句进行数据覆盖。

示例

创建一个表t1，表中拥有4列，其中a列为主键，建表语句如下：

```
CREATE TABLE t1 (a int PRIMARY KEY, b int, c int, d int DEFAULT 0);
```

对表t1插入一行数据，主键列a的值为0，插入数据语句如下：

```
INSERT INTO t1 VALUES (0,0,0,0);
```

查看表数据：

```
SELECT * FROM t1;
```

返回信息如下：

```
 a | b | c | d
---+---+---+---
 0 | 0 | 0 | 0
(1 row)
```

如果再对表t1插入一行数据，主键列a的值还是0，则会返回一个报错，插入数据语句如下：

```
INSERT INTO t1 VALUES (0,1,1,1);
```

报错信息如下：

```
ERROR:  duplicate key value violates unique constraint "t1_pkey"
DETAIL:  Key (a)=(0) already exists.
```

如果不希望出现上述报错信息，可以使用本文介绍的覆盖写入特性来进行处理：

- 使用ON CONFLICT DO NOTHING子句：主键冲突的情况下，不执行任何操作（适用于有冲突丢弃冲突数据的场景）。

  插入数据语句如下：

  ```
  INSERT INTO t1 VALUES (0,1,1,1) ON CONFLICT DO NOTHING;
  ```

  查看表数据：

```
SELECT * FROM t1;
```

表t1没有进行任何操作，返回示例如下：

```
 a | b | c | d
---+---+---+---
 0 | 0 | 0 | 0
(1 row)
```

- 使用ON CONFLICT DO UPDATE子句：主键冲突的情况下，更新非主键的列（适用于全部列覆盖写入的场景）。

  插入数据语句如下：

  ```
  INSERT INTO t1 VALUES (0,2,2,2) ON CONFLICT (a) DO UPDATE SET (b, c, d) = (excluded.b, excluded.c, excluded.d);
  ```

  或

  ```
  INSERT INTO t1 VALUES (0,2,2,2) ON CONFLICT (a) DO UPDATE SET b = excluded.b, c = excluded.c, d = excluded.d;
  ```

  在DO UPDATE SET子句中，可以使用excluded表示冲突的数据构成的伪表，在主键冲突的情况下，引用伪表中列的值覆盖原来列的值。上述语句中，新插入的数据 `(0,2,2,2)` 构成了一个伪表，伪表包含1行4列数据，表名为excluded，可以使用 `excluded.b, excluded.c, excluded.d` 去引用伪表中的列。

  查看表数据：

  ```
  SELECT * FROM t1;
  ```

  表t1中的非主键列进行了更新，返回示例如下：

  ```
   a | b | c | d
  ---+---+---+---
   0 | 2 | 2 | 2
  (1 row)
  ```

除了上述两种情况，覆盖写入功能支持更多使用场景，场景如下：

- 主键冲突的情况下，在部分列中覆盖写入数据（适用于基于冲突数据覆盖部分列的场景）：

  例如主键冲突后，仅覆盖c列的数据，插入数据语句如下：

  ```
  INSERT INTO t1 VALUES (0,0,3,0) ON CONFLICT (a) DO UPDATE SET c = excluded.c;
  ```

  查看表数据：

  ```
  SELECT * FROM t1;
  ```

  返回示例如下：

  ```
   a | b | c | d
  ---+---+---+---
   0 | 2 | 3 | 2
  (1 row)
  ```

- 主键冲突的情况下，更新部分列的数据（适用于基于原始数据更新部分列场景）：

  例如主键冲突后，将d列的数据加1，插入数据语句如下：

  ```
  INSERT INTO t1 VALUES (0,0,3,0) ON CONFLICT (a) DO UPDATE SET d = t1.d + 1;
  ```

  查看表数据：

  ```
  SELECT * FROM t1;
  ```

  返回示例如下：

  ```
   a | b | c | d
  ---+---+---+---
   0 | 2 | 3 | 3
  (1 row)
  ```

- 主键冲突的情况下，更新数据为默认值（适用于冲突后，回退数据到默认值的场景）：

  例如主键冲突后，将d列恢复到默认值（上文中d列的默认值为0），插入数据语句如下：

  ```
  INSERT INTO t1 VALUES (0,0,3,0) ON CONFLICT (a) DO UPDATE SET d = default;
  ```

  查看表数据：

  ```
  SELECT * FROM t1;
  ```

  返回示例如下：

  ```
   a | b | c | d
  ---+---+---+---
   0 | 2 | 3 | 0
  (1 row)
  ```

- 插入多条数据：
  - 例如插入2行数据，其中主键冲突的行不进行任何操作，主键不冲突的行正常插入，插入数据语句如下：

    ```
    INSERT INTO t1 VALUES (0,0,0,0), (1,1,1,1) ON CONFLICT DO NOTHING;
    ```

    查看表数据：

    ```
    SELECT * FROM t1;
    ```

    返回示例如下：

    ```
     a | b | c | d
    ---+---+---+---
     0 | 2 | 3 | 0
     1 | 1 | 1 | 1
    (2 rows)
    ```

- 例如插入2行数据，主键冲突的行进行覆盖写入，主键不冲突的行正常插入，插入数据语句如下：

```
INSERT INTO t1 VALUES (0,0,0,0), (2,2,2,2) ON CONFLICT (a) DO UPDATE SET (b, c, d) = (e
xcluded.b, excluded.c, excluded.d);
```

查看表数据：

```
SELECT * FROM t1;
```

返回示例如下：

```
 a | b | c | d
---+---+---+---
 0 | 0 | 0 | 0
 1 | 1 | 1 | 1
 2 | 2 | 2 | 2
(3 rows)
```

- 插入的数据来自于子查询，如果主键冲突，则覆盖写入（用于合并两表数据或更复杂的INSERT INTO SELECT场景）：

创建表t2，数据结构与表t1一致，建表语句如下：

```
CREATE TABLE t2 (like t1);
```

在表t2中插入两行数据，插入数据语句如下：

```
INSERT INTO t2 VALUES (2,22,22,22),(3,33,33,33);
```

将表t2的数据插入表t1，如果主键冲突，则覆盖写入非主键的列，插入数据语句如下：

```
INSERT INTO t1 SELECT * FROM t2 ON CONFLICT (a) DO UPDATE SET (b, c, d) = (excluded.b, ex
cluded.c, excluded.d);
```

查看表数据：

```
SELECT * FROM t1;
```

返回示例如下：

```
 a | b  | c  | d
---+----+----+----
 0 |  0 |  0 |  0
 1 |  1 |  1 |  1
 2 | 22 | 22 | 22
 3 | 33 | 33 | 33
(4 rows)
```

6.使用COPY ON CONFLICT覆盖导入数据

AnalyticDB PostgreSQL版支持COPY ON CONFLICT覆盖导入数据。目前COPY ON CONFLICT仅支持全表约束检查及全列覆盖写入。

在AnalyticDB PostgreSQL版中，您可以通过COPY快速导入数据，但是在COPY导入数据的过程中，如果数据与表的约束冲突，COPY任务会报错并终止。AnalyticDB PostgreSQL提供了COPY ON CONFLICT功能，支持在约束冲突时进行覆盖写入或忽略写入，避免COPY任务因为约束冲突而失败。

> **说明**
>
> 仅内核编译日期为20210528及以后的AnalyticDB PostgreSQL 6.0版实例支持COPY ON CONFLICT功能。为了更好地使用该功能，建议您升级至最新的内核版本，升级内核小版本，请参见版本升级。

使用约束

- 仅支持存储弹性模式实例，暂不支持Serverless版本实例。
- 目标表需为堆表，不支持AO表（AO表不支持唯一索引，所以不支持AO表）。
- 仅V6.3.6.1及以上内核版本支持目标表为分区表。如何升级内核版本，请参见版本升级。
- 目标表不支持Updatable View（可更新视图）。
- COPY ON CONFLICT仅支持COPY FROM，不支持COPY TO。
- 不支持指定约束索引列，COPY ON CONFLICT默认判断所有约束列。若指定约束索引列，则COPY执行失败，报错信息如下：

```
COPY NATION FROM stdin DO ON CONFLICT(n_nationkey) DO UPDATE;
ERROR:  COPY ON CONFLICT does NOT support CONFLICT index params
```

- 不支持指定更新列，COPY ON CONFLICT默认更新所有列。若指定更新列，则COPY执行失败，报错信息如下：

```
COPY NATION FROM stdin DO ON CONFLICT DO UPDATE SET n_nationkey = excluded.n_nationkey;
ERROR:  COPY ON CONFLICT does NOT support UPDATE SET targets
```

语法

```
COPY table [(column [, ...])] FROM {'file' | STDIN}
     [ [WITH]
       [BINARY]
       [OIDS]
       [HEADER]
       [DELIMITER [ AS ] 'delimiter']
       [NULL [ AS ] 'null string']
       [ESCAPE [ AS ] 'escape' | 'OFF']
       [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
       [CSV [QUOTE [ AS ] 'quote']
            [FORCE NOT NULL column [, ...]]
       [FILL MISSING FIELDS]
       [[LOG ERRORS]
       SEGMENT REJECT LIMIT count [ROWS | PERCENT] ]
    [DO ON CONFLICT DO UPDATE | NOTHING]
```

COPY ON CONFLICT提供了DO ON CONFLICT DO UPDATE和DO ON CONFLICT DO NOTHING两个子句：

- DO ON CONFLICT DO UPDATE表示表约束冲突时全列更新。
- DO ON CONFLICT DO NOTHING表示表约束冲突时忽略输入内容。

示例

1. 创建一个表NATION，表中包含4列，其中N_NATIONKEY为主键列，具有主键约束，建表语句如下：

   ```
   CREATE TABLE NATION (
       N_NATIONKEY  INTEGER,
       N_NAME       CHAR(25),
       N_REGIONKEY  INTEGER,
       N_COMMENT    VARCHAR(152),
       PRIMARY KEY (N_NATIONKEY)
   );
   ```

2. 通过COPY导入部分数据，COPY语句如下：

   ```
   COPY NATION FROM stdin;
   ```

   出现>>标志后逐条输入如下内容：

   ```
   0 'ALGERIA' 0 'haggle. carefully final deposits detect slyly agai'
   1 'ARGENTINA' 1 'al foxes promise slyly according to the regular accounts. bold request
   s alon'
   2 'BRAZIL' 1 'y alongside of the pending deposits. carefully special packages are about
   the ironic forges. slyly speci'
   3 'CANADA' 1 'eas hang ironic, silent packages. slyly regular packages are furiously ov
   er the tithes. fluffily bold'
   \.
   ```

   > **说明**
   >
   > 复制以上数据时，请将两列值之间的空格替换为Tab。

3. 查询NATION表，查看已经导入的数据，查询语句如下：

```
SELECT * from NATION;
```

返回信息如下：

```
 n_nationkey |          n_name           | n_regionkey |
n_comment

-------------+---------------------------+-------------+-----------------------------
-------------------------------------------------------------------------------
           2 | 'BRAZIL'                  |           1 | 'y alongside of the pending de
posits. carefully special packages are about the ironic forges. slyly speci'
           3 | 'CANADA'                  |           1 | 'eas hang ironic, silent packa
ges. slyly regular packages are furiously over the tithes. fluffily bold'
           0 | 'ALGERIA'                 |           0 | ' haggle. carefully final depo
sits detect slyly agai'
           1 | 'ARGENTINA'               |           1 | 'al foxes promise slyly accord
ing to the regular accounts. bold requests alon'
(4 rows)
```

4. 使用COPY语句导入一行主键冲突的数据，COPY语句如下：

```
COPY NATION FROM stdin;
```

出现>>标志后逐条输入如下内容：

```
0 'GERMANY' 3 'l platelets. regular accounts x-ray: unusual, regular acco'
\.
```

> **说明**
>
> 复制以上数据时，请将两列值之间的空格替换为Tab。

此时执行会产生报错，报错内容如下：

```
ERROR:  duplicate key value violates unique constraint "nation_pkey"
DETAIL:  Key (n_nationkey)=(0) already exists.
CONTEXT:  COPY nation, line 1
```

5. 使用COPY ON CONFLICT语句，在主键冲突的情况下更新数据，COPY ON CONFLICT语句如下：

```
COPY NATION FROM stdin DO  ON CONFLICT DO UPDATE;
```

出现>>标志后逐条输入如下内容：

```
0 'GERMANY' 3 'l platelets. regular accounts x-ray: unusual, regular acco'
\.
```

> **说明**
>
> 复制以上数据时，请将两列值之间的空格替换为Tab。

此时COPY语句不会产生报错信息，查询NATION表可以看到主键为0的行数据已更新，查询语句如下：

```
SELECT * FROM NATION;
```

返回信息如下：

```
 n_nationkey |          n_name           | n_regionkey |
n_comment

-------------+---------------------------+-------------+-----------------------------
-------------------------------------------------------------------------------
           2 | 'BRAZIL'                  |           1 | 'y alongside of the pending de
posits. carefully special packages are about the ironic forges. slyly speci'
           3 | 'CANADA'                  |           1 | 'eas hang ironic, silent packa
ges. slyly regular packages are furiously over the tithes. fluffily bold'
           1 | 'ARGENTINA'               |           1 | 'al foxes promise slyly accord
ing to the regular accounts. bold requests alon'
           0 | 'GERMANY'                 |           3 | 'l platelets. regular accounts
x-ray: unusual, regular acco'
(4 rows)
```

6. 使用COPY ON CONFLICT功能，在主键冲突的情况下，忽略输入：

```
COPY NATION FROM stdin DO ON CONFLICT DO NOTHING;
```

出现>>标志后逐条输入如下内容：

```
1 'GERMANY' 3 'l platelets. regular accounts x-ray: unusual, regular acco'
\.
```

> **说明**
>
> 复制以上数据时，请将两列值之间的空格替换为Tab。

此时COPY语句不会产生报错信息，查询NATION表可以看到主键为1的行数据没有更新，查询语句如下：

```
SELECT * FROM NATION;
```

返回信息如下：

```
 n_nationkey |          n_name           | n_regionkey |
n_comment

-------------+---------------------------+-------------+-----------------------------
-------------------------------------------------------------------------------
           2 | 'BRAZIL'                  |           1 | 'y alongside of the pending de
posits. carefully special packages are about the ironic forges. slyly speci'
           3 | 'CANADA'                  |           1 | 'eas hang ironic, silent packa
ges. slyly regular packages are furiously over the tithes. fluffily bold'
           1 | 'ARGENTINA'               |           1 | 'al foxes promise slyly accord
ing to the regular accounts. bold requests alon'
           0 | 'GERMANY'                 |           3 | 'l platelets. regular accounts
x-ray: unusual, regular acco'
(4 rows)
```

7.使用ODPS Foreign Table访问MaxCompute数据

ODPS Foreign Table（简称ODPS FDW）是AnalyticDB PostgreSQL版基于PostgreSQL Foreign Data Wrapper是（简称 PG FDW）框架开发的用于访问大数据计算服务MaxCompute的外部数据访问方案。

ODPS FDW模块的加入，填补了当前AnalyticDB PostgreSQL与MaxCompute的数据同步链路的缺失。您通过ODPS FDW可以创建三种类型的ODPS外表。

- ODPS非分区外表：映射MaxCompute非分区表。
- ODPS末级分区外表：映射MaxCompute末级分区表。
- ODPS分区外表：映射MaxCompute分区表。

开始使用 ODPS FDW

1. 在AnalyticDB PostgreSQL数据库中创建ODPS FDW插件。

```
CREATE EXTENSION odps_fdw ;
```

2. 将使用权赋权给所有用户，示例如下：

```
GRANT USAGE ON FOREIGN DATA WRAPPER odps_fdw TO PUBLIC;
```

- 新建实例默认创建ODPS FDW extension，无需执行创建和赋权操作。
- 既存实例，可以使用**初始账号**，连接指定 database手动执行如下命令，创建 ODPS FDW extension。

开始使用 ODPS Foreign Table

使用 ODPS 外表需要以下三要素，缺一不可，包括：

- ODPS Server - 定义 MaxCompute 的访问端点。
- ODPS User Mapping - 定义 MaxCompute 的访问账户。
- ODPS Foreign Table - 定义 MaxCompute 的访问对象。

1. 创建 ODPS Server

1.1 语法示例

```
CREATE SERVER odps_serv                          -- ODPS Server 名称
  FOREIGN DATA WRAPPER odps_fdw
  OPTIONS (
    tunnel_endpoint '<odps tunnel endpoint>'    -- ODPS Tunnel Endpoint
  );
```

1.2 参数选项

在 ADB PG 中定义 ODPS Server 只需要指定 tunnel_endpoint 或 odps_endpoint 即可。其中：

选项	是否必选	备注
tunnel_endpoint	可选，优先推荐设置此选项。	指 ODPS tunnel 服务的 Endpoint。
odps_endpoint	可选	指 MaxCompute 服务的 Endpoint。

> 说明
>
> - 创建时，可以设置其中任意一个，也可以都设置，优先选择使用 Tunnel Endpoint，当缺少 Tunnel Endpoint时，则通过 ODPS Endpoint 路由到对应的 Tunnel Endpoint。
> - 推荐配置阿里云经典网络或VPC网络的 Tunnel Endpoint。
> - 通过配置外网 Tunnel Endpoint 地址访问 MaxCompute 数据，价格为0.1166美元/GB。

关于 ODPS Endpoint，请参见配置Endpoint。

2. 创建 ODPS User Mapping

2.1 语法示例

```
CREATE USER MAPPING FOR { username | USER | CURRENT_USER | PUBLIC }
  SERVER odps_serv                                    -- ODPS Server 名称
  OPTIONS (
    id '<odps access id>',                                -- ODPS Account ID
    key '<odps access key>'                               -- ODPS Account Key
  );
```

> 说明
>
> - **username** The name of an existing user that is mapped to foreign server.
> - **CURRENT_USER** and **USER** match the name of the current user.
> - **PUBLIC** all roles, including those that might be created later.

2.2 参数选项

在 ADB PG 中定义访问 ODPS Server 的账户，需要指定账户类型 TYPE，ID 和 KEY。

选项	是否必选	备注
`id`	必选	指定账户ID

选项	是否必选	备注
key	必选	指定账户KEY

3. 创建 ODPS Foreign Table

3.1 语法示例

```
CREATE FOREIGN TABLE IF NOT EXISTS table_name ( -- ODPS 外表名称
    column_name data_type [, ... ]
)
  SERVER odps_serv                              -- ODPS Server 名称
  OPTIONS (
    project '<odps project>',                   -- ODPS 项目空间
    table '<odps table>'                        -- ODPS 表名称
);
```

3.2 参数选项

定义了 ODPS Server 和 ODPS User Mapping 后，就可以创建 ODPS Foreign Table。参数选项包括：

选项	是否必选	备注
project	必选	即项目/项目空间。项目空间（Project）是 MaxCompute 的基本组织单元，它类似于传统数据库的 Database 或 Schema 的概念，是进行多用户隔离和访问控制的主要边界。详情请参见项目。
table	必选	即 ODPS 表。表是 MaxCompute 的数据存储单元，详情请参见表。
partition	可选	即用于定义 MaxCompute 的末级分区表。分区 partition 是指一张表下，根据分区字段（一个或多个字段的组合）对数据存储进行划分。也就是说，如果表没有分区，数据是直接放在表所在的目录下。如果表有分区，每个分区对应表下的一个目录，数据是分别存储在不同的分区目录下。关于分区的更多介绍请参见分区。

3.3 外表分类

根据 MaxCompute 的表分类，ODPS FDW 支持定义以下三种类型的 ODPS 外表。

- 非分区外表

  非分区 ODPS 外表映射的是 MaxCompute 的非分区表。用户创建外表时，只需要指定有效的 **project** 和 **table** 属性即可，无需指定 **partition** 属性或者指定 **partition** 属性为“空”。例如：

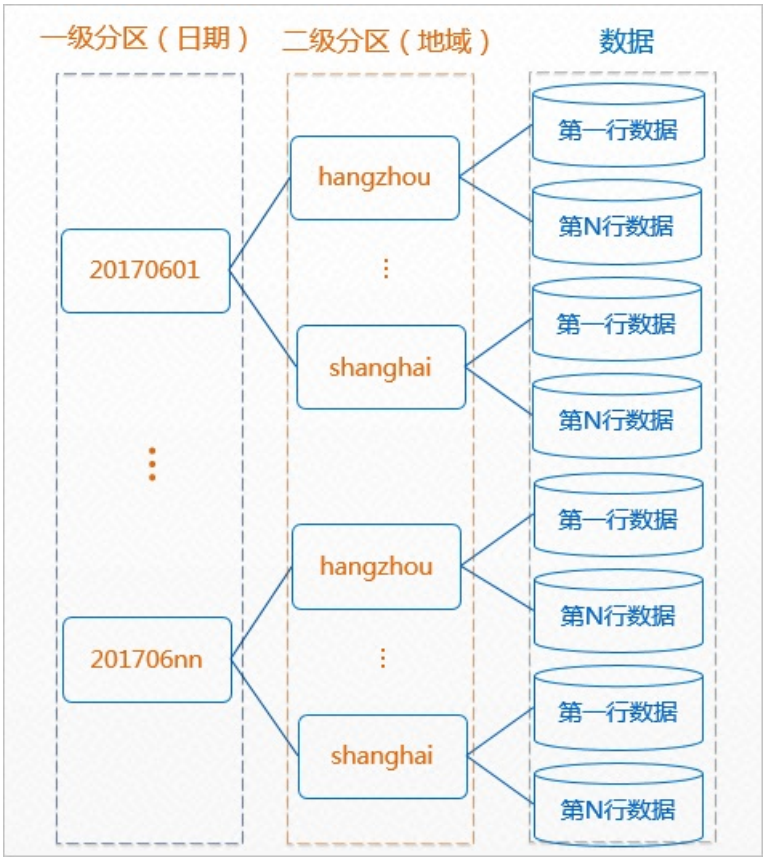
```
CREATE FOREIGN TABLE odps_lineitem (              -- ODPS 外表名称
    l_orderkey      bigint,
    l_partkey       bigint,
    l_suppkey       bigint,
    l_linenumber    bigint,
    l_quantity      double precision,
    l_extendedprice double precision,
    l_discount      double precision,
    l_tax           double precision,
    l_returnflag    char(1),
    l_linestatus    char(1),
    l_shipdate      date,
    l_commitdate    date,
    l_receiptdate   date,
    l_shipinstruct  char(25),
    l_shipmode      char(10),
    l_comment       varchar(44)
) SERVER odps_serv                                -- ODPS Server 名称
OPTIONS (
  project 'odps_fdw',                             -- ODPS 项目空间
  table 'lineitem_big'                            -- ODPS 表名称
);
```

- 末级分区外表

  相对于非分区外表，末级分区外表，映射的是 MaxCompute 的末级分区表，需要设置正确的 **partition** 属性，多级分区时，末级分区外表只支持末级分区表，即 **partition** 属性需要包含多级分区完整路径。

  举例说明：在 MaxCompute 上创建一个二级分区表，如下：

```
--创建一个二级分区表，以日期为一级分区，地域为二级分区
CREATE TABLE src (key string, value bigint) PARTITIONED BY (pt string,region string);
```



当我们需要在 ADB PG 上定义一个末级分区外表，映射 MaxCompute 上一级分区为（20170601）二级分区为（hangzhou）的末级分区表时，需要设置 partition 为“pt=20170601,region=hangzhou”。

```
CREATE FOREIGN TABLE odps_src_20170601_hangzhou (    -- ODPS 外表
  key string,
  value bigint
) SERVER odps_serv                                   -- ODPS Server 名称
OPTIONS (
  project 'odps_fdw',                                -- ODPS 项目空间
  table 'src',                                       -- ODPS 表名称
  partition 'pt=20170601,region=hangzhou'            -- 末级分区完整路径
);
```

说明

i. 分区的 partition specification 需按照 key=value 的方式设置；多级分区时，以逗号分隔，且不能包含额外的空格。

ii. 不支持映射非末级分区的表，如，不支持仅设置一级分区路径：`partition 'pt=20170601'`。

iii. 一定要设置完整的多级分区路径，如，不支持仅设置末级分区路径：`partition 'region=hangzhou'`。

- 分区外表

ODPS 分区外表，映射的是 MaxCompute 的分区表。同样，以上述 MaxCompute 二级分区表 src 为例。我们可以按照如下方法创建对应的分区外表。更多表分区定义。

```
CREATE FOREIGN TABLE odps_src(                    -- ODPS 外表名称
  key text,
  value bigint,
  pt text,                                        -- ODPS 一级分区键
  region text                                     -- ODPS 二级分区键
) SERVER odps_serv
OPTIONS (
  project 'odps_fdw',                             -- ODPS 项目空间
  table 'src'                                     -- ODPS 表名称
)
PARTITION BY LIST (pt)                            -- 一级分区以"pt"字段为分区键
SUBPARTITION BY LIST (region)                     -- 二级分区以"region"字段为分区键
    SUBPARTITION TEMPLATE (                       -- 二级分区模板
       SUBPARTITION hangzhou VALUES ('hangzhou'),
       SUBPARTITION shanghai VALUES ('shanghai')
    )
( PARTITION "20170601" VALUES ('20170601'),
  PARTITION "20170602" VALUES ('20170602'));
```

说明

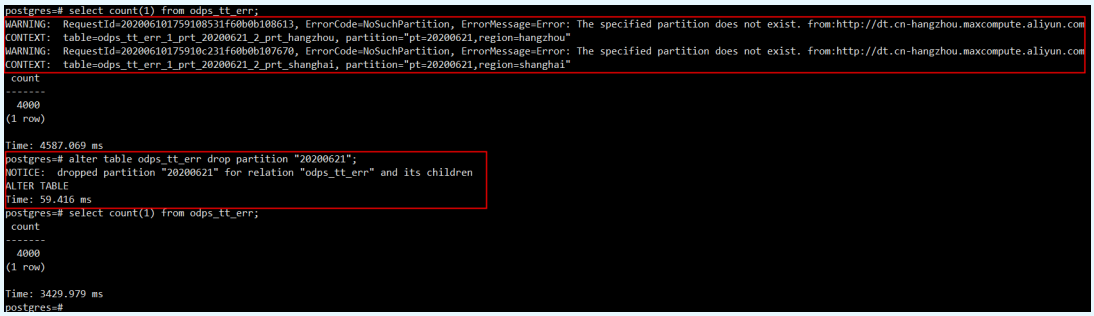
与 MaxCompute 分区表定义不同，ADB PG 的分区外表定义时：

i. 需要将分区键以字段方式定义在其他字段尾部，多级分区时，分区字段定义顺序、分区键层级、MaxCompute 分区表层级三者需保持一致。

ii. 分区表定义需要指定分区键值，请使用 LIST 方式分区。

iii. 分区外表定义时，不需要指定 partition 属性，这是因为 partition 属性标记的是末级分区外表，与分区外表在逻辑上冲突。

iv. 当出现 MaxCompute 上不存在的分区外表，查询外表时，会告警显示，用户可以参考本章的3.5节**如何删除子分区外表**删除相应子分区。

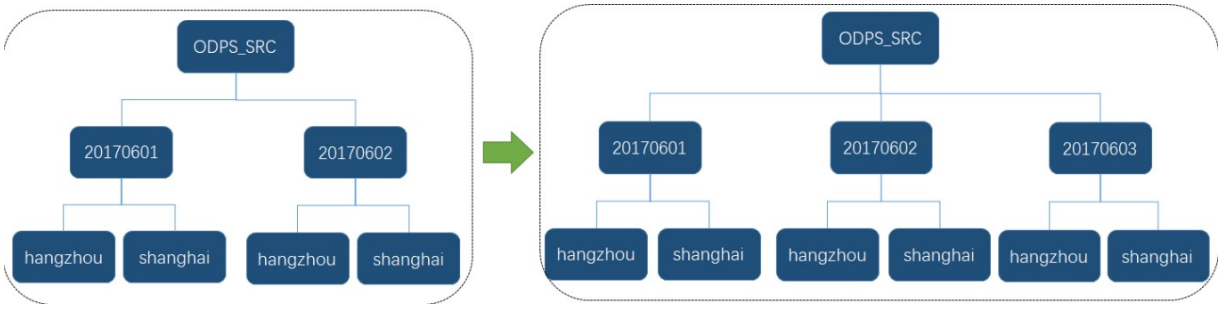


3.4 如何添加子分区外表

以上述 odps_src 分区外表为例。

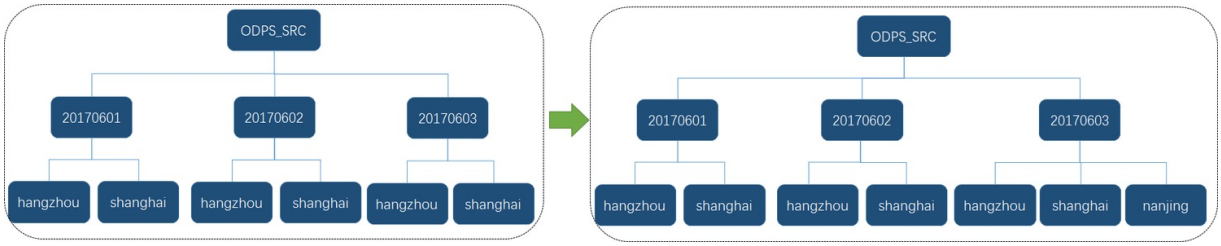
- 添加一级子分区，效果如下图。

```
-- 添加一级子分区（自动创建二级子分区）
alter table odps_src add partition "20170603" values(20170603);
```



- 添加二级子分区，效果如下图。

```
-- 添加二级子分区
alter table odps_src alter partition "20170603" add partition "nanjing" values('nanjing');
```

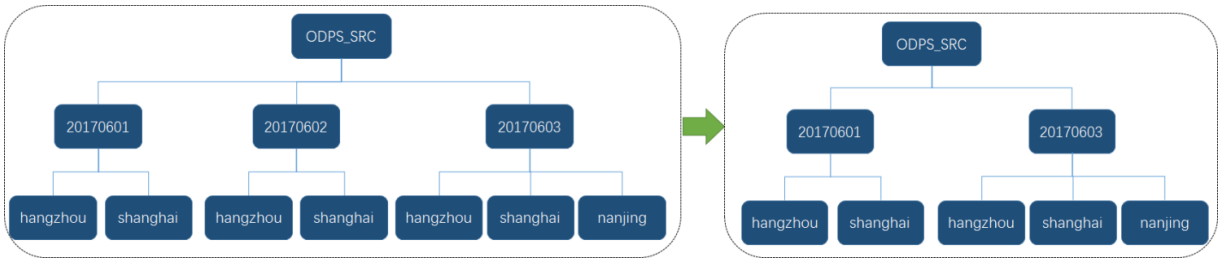


3.5 如何删除子分区外表

以上述 odps_src 分区外表为例。

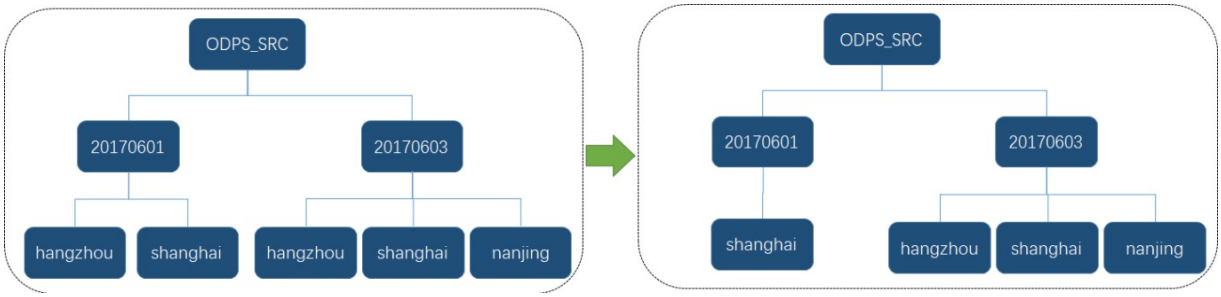
- 删除一级子分区，效果如下图。

```
-- 删除一级子分区（级联删除二级子分区）
alter table odps_src drop partition "20170602";
```



- 删除二级子分区

```
-- 删除二级子分区
alter table odps_src alter partition "20170601" drop partition "hangzhou";
```



ODPS 外表数据类型

目前 MaxCompute 数据类型与 ADB PG 数据类型的对应关系如下，建议按照此类型对照表来定义 ADB PG 外表的字段类型。

> **说明**
>
> 目前暂不支持 MaxCompute 的STRUCT/MAP/ARRAY类型。

ODPS类型	ADB PG类型
BOOLEAN	bool
TINYINT	int2
SMALLINT	int2
INTEGER	int4
BIGINT	int8
FLOAT	float4
DOUBLE	float8
DECIMAL	numeric
BINARY	bytea
VARCHAR(n)	varchar(n)
CHAR(n)	char(n)
STRING	text
DATE	date
DATETIME	timestamp

ODPS类型	ADB PG类型
TIMESTAMP	timestamp

ODPS 外表使用场景

ODPS 外表扫描，实现了 ADB PG 的 Foreign Scan 算子。因此，表查询的使用方法上，对 ODPS 外表的查询与对普通表的查询基本一致。本文以TPC-H Query为例，举例说明常见的使用场景。

ODPS 外表查询分析

TPC-H Query Q1 是典型的单表聚集过滤场景，我们定义 odps_lineitem 为 ODPS 外表，对其执行 Q1 查询。

```
-- 定义 ODPS 外表 odps_lineitem
CREATE FOREIGN TABLE odps_lineitem (
    l_orderkey bigint,
    l_partkey bigint,
    l_suppkey bigint,
    l_linenumber bigint,
    l_quantity double precision,
    l_extendedprice double precision,
    l_discount double precision,
    l_tax double precision,
    l_returnflag CHAR(1),
    l_linestatus CHAR(1),
    l_shipdate DATE,
    l_commitdate DATE,
    l_receiptdate DATE,
    l_shipinstruct CHAR(25),
    l_shipmode CHAR(10),
    l_comment VARCHAR(44)
) server odps_serv
    options (
        project 'odps_fdw', table 'lineitem'
    );

-- TPC-H Q1
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    odps_lineitem
where
    l_shipdate <= date '1998-12-01' - interval '88' day --(3)
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus;
```

ODPS 数据导入本地表

导入数据时，请执行如下步骤：

1. 在 ADB PG 中，创建 ODPS 外表。
2. 执行如下操作，并行导入数据。

```
-- INSERT 方式
INSERT INTO <本地目标表> SELECT * FROM <ODPS 外表>;

-- CREATE TABLE AS 方式
CREATE TABLE <本地目标表> AS SELECT * FROM <ODPS 外表>;
```

- **示例1** - INSERT 方式将 odps_lineitem 数据导入到本地 AOCS 表。

```
-- 创建本地 AOCS 表
CREATE TABLE aocs_lineitem (
    l_orderkey bigint,
    l_partkey bigint,
    l_suppkey bigint,
    l_linenumber bigint,
    l_quantity double precision,
    l_extendedprice double precision,
    l_discount double precision,
    l_tax double precision,
    l_returnflag CHAR(1),
    l_linestatus CHAR(1),
    l_shipdate DATE,
    l_commitdate DATE,
    l_receiptdate DATE,
    l_shipinstruct CHAR(25),
    l_shipmode CHAR(10),
    l_comment VARCHAR(44)
) WITH (APPENDONLY=TRUE, ORIENTATION=COLUMN, COMPRESSTYPE=ZSTD, COMPRESSLEVEL=5)
DISTRIBUTED BY (l_orderkey);

-- 将 odps_lineitem 数据导入到 AOCS 本地表
INSERT INTO aocs_lineitem SELECT * FROM odps_lineitem;
```

- **示例2** - CREATE TABLE AS 方式将 odps_lineitem 导入到本地 heap 表。

```
create table heap_lineitem as select * from odps_lineitem distributed by (l_orderkey);
```

ODPS 外表与本地表关联

以TPC-H Query Q19为例，使用本地列存表aocs_lineitem与ODPS外表odps_part关联查询。

```
-- TPC-H Q19
select
    sum(l_extendedprice* (1 - l_discount)) as revenue
from
    aocs_lineitem,          -- 本地 AOCS 列存表
    odps_part               -- ODPS 外表
where
    (
        p_partkey = l_partkey
        and p_brand = 'Brand#32'
        and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
        and l_quantity >= 8 and l_quantity <= 8 + 10
        and p_size between 1 and 5
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p_partkey = l_partkey
        and p_brand = 'Brand#41'
        and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
        and l_quantity >= 15 and l_quantity <= 15 + 10
        and p_size between 1 and 10
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p_partkey = l_partkey
        and p_brand = 'Brand#44'
        and p_container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
        and l_quantity >= 22 and l_quantity <= 22 + 10
        and p_size between 1 and 15
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    );
```

ODPS 外表使用常见错误

Tunnel常见错误

ODPS 外表使用建议

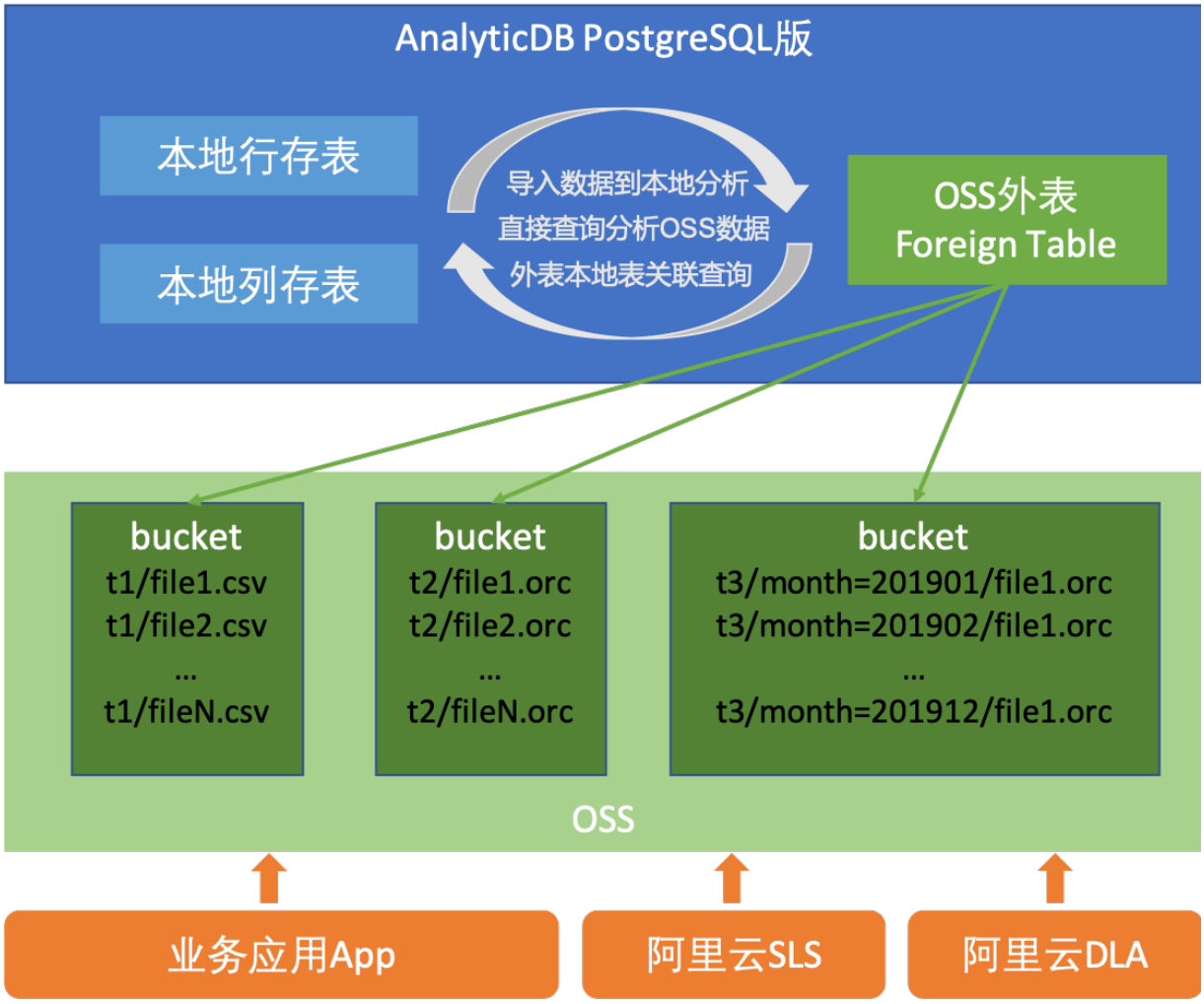
ODPS 外表通过网络访问 MaxCompute，使用瓶颈除了机器自身资源外，还受限于 MaxCompute Tunnel 对外吞吐的网络带宽 。因此，建议用户

1. 建议纯外表使用的并发数不超过5个。
2. 多张ODPS外表关联使用时，建议将大表导入本地后再和小的外表关联，性能更佳。

8.使用OSS Foreign Table访问OSS数据

OSS Foreign Table是基于PostgreSQL Foreign Data Wrapper（简称PG FDW）框架开发，用于访问OSS数据的数据分析方案。

功能介绍



通过OSS Foreign Table，您可以进行如下操作：

- 导入OSS数据到本地表（行存表或列存表）进行分析加速。
- 直接查询分析OSS海量数据。
- OSS外表与本地表关联分析。

目前OSS Foreign Table支持ORC、PARQUET、JSON、JSONLINE、CSV（支持GZIP、标准SNAPPY压缩）文件格式，同时支持按一个或多个字段进行分区，查询特定分区时起到过滤效果。

OSS上的数据，可以来自业务应用App的写入、阿里云SLS的日志归档、阿里云DLA的ETL输出等。

与 OSS External Table 区别

- ADB PG之前已支持OSS External Table用于数据导入导出，但是用于OSS数据分析的话在大数据量场景下性能无法达到预期。

- OSS Foreign Table基于PostgreSQL Foreign Data Wrapper（简称PG FDW）框架开发，支持orc，csv（支持gzip压缩）文件格式，同时支持按一个或多个字段进行分区，支持外表统计信息收集帮助优化器生成最优执行计划。
- Foreign Table整体在性能，功能和稳定性上都优于External Table。同时Greenplum社区本身接下来也计划用Foreign Table代替External Table。

开始使用 OSS Foreign Table

OSS Foreign Table的使用方法可以理解为：以**USER MAPPING**（CREATE USER MAPPING）用户在**OSS FOREIGN SERVER**（CREATE SERVER）上定义**OSS FOREIGN TABLE**（CREATE FOREIGN TABLE）。

例如，您可以通过命令行工具ossutil查看OSS域上TPCH lineitem表，查询命令示例如下。

```
ossutil ls oss://adbpg-tpch/data/tpch_data_10x/lineitem.tbl
```

其中：

- **adbpg-tpch**：OSS bucket名称。
- **data/tpch_data_10x/...**：文件相对于 bucket 的路径。

返回的相关列表如下。

```
LastModifiedTime                   Size(B)  StorageClass   ETAG
ObjectName
2020-03-12 09:29:48 +0800 CST    144144997      Standard   1F426F2FFC70A0262D2D69183BC3A0BD
-57   oss://adbpg-tpch/data/tpch_data_10x/lineitem.tbl.1
2020-03-12 09:29:58 +0800 CST    145177420      Standard   CFE2CFF1C8059547DC9F1711E77F74DD
-57   oss://adbpg-tpch/data/tpch_data_10x/lineitem.tbl.10
2020-03-10 21:23:24 +0800 CST    145355168      Standard   35C6227D1C29F1236A92A4D5D7922625
-57   oss://adbpg-tpch/data/tpch_data_10x/lineitem.tbl.11
... ...
```

以下内容将详细讲解如何创建并使用OSS Foreign Table。

创建OSS Server

创建OSS Server，即定义需要访问的OSS服务端，需要指定 `endpoint` 参数。

语法如下：

```
CREATE SERVER oss_serv              -- OSS server名称
    FOREIGN DATA WRAPPER oss_fdw
    OPTIONS (
        endpoint '<oss endpoint>',  -- OSS server域名
        bucket '<oss bucket>'       -- 指定数据文件所在的OSS bucket
  );
```

OPTIONS的取值请参见下表。了解更多请参见 CREATE SERVER。

选项	类型	是否必选	说明
oss endpoint	字符串	必选	OSS对应区域的域名。 **说明** 如果是从阿里云的主机访问数据库，应该使用内网域名（即带有 `internal` 的域名），避免产生公网流量。
oss bucket	字符串	可选	指定数据文件所在的Bucket，需要通过OSS预先创建。 **说明** OSS Server和OSS Table必须有一个设置该选项且OSS Table的优先级高，会覆盖OSS Server的值。
说明 • 以下为OSS访问时容错相关参数，通常不需要额外设置，使用默认值即可。 • 下列参数如果使用默认值，表示如果连续1500秒的传输速率小于1 KB，就会触发超时。详细信息，请参见错误处理。			
speed_limit	数值	可选	设置能容忍的最小速率。
speed_time	数值	可选	设置能容忍speed_limit的最长时间。
connect_timeout	数值	可选	设置连接超时时间。
dns_cache_timeout	数值	可选	设置DNS超时时间。

创建OSS User Mapping

创建OSS Server后，还需要创建访问OSS Server的用户。通过创建OSS User Mapping，定义AnalyticDB PostgreSQL数据库用户到访问OSS Server用户的映射关系。

语法

- 创建用户映射

```
CREATE USER MAPPING FOR { username | USER | CURRENT_USER | PUBLIC }
    SERVER servername
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

- 删除用户映射

```
DROP USER MAPPING [ IF EXISTS ] FOR { user_name | USER | CURRENT_USER | PUBLIC }
    SERVER server_name
```

参数选项

OPTIONS参数选项请参见下表。更多信息，请参见CREATE USER MAPPING。

选项	是否必选	说明
`id`	必选	OSS账号ID。
`key`	必选	OSS账号KEY。
`username`	可选	映射到Foreign Server的现有名称。
`CURRENT_USER` 或 `USER`	可选	匹配当前用户的名称。
`PUBLIC`	可选	所有角色，包括以后可能创建的角色。

示例

```
CREATE USER MAPPING FOR PUBLIC  -- 为所有用户创建到OSS server用户的映射
    SERVER oss_serv             -- 指定需要访问的OSS server
    OPTIONS (
      id '<oss access id>',     -- 指定OSS账号ID
      key '<oss access key>'    -- 指定OSS账号KEY
    );
```

- **创建OSS Foreign Table**

拥有OSS Server和访问OSS Server的用户之后，就可以定义OSS Foreign Table。目前，OSS Foreign Table支持多种格式的数据文件，适应不同的业务场景。

支持的格式文件如下：

- 支持访问CSV、TEXT、JSON、JSONLINE格式的非压缩文本文件。

- 支持访问CSV、TEXT格式的GZIP压缩、标准SNAPPY压缩文本文件。支持访问JSON、JSONLINE格式的GZIP压缩文本文件。
- 支持访问ORC格式的二进制文件。ORC数据类型到AnalyticDB PostgreSQL数据类型的映射关系请参见 ORC - ADB PG 数据类型对照表。
- 支持访问PARQUET格式的二进制文件。PARQUET数据类型到AnalyticDB PostgreSQL数据类型的映射关系详见 PARQUET - ADB PG 数据类型对照表。

语法

- 创建OSS Foreign Table

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name ( [
    column_name data_type [ OPTIONS ( option 'value' [, ... ] ) ] [ COLLATE collation ] [
column_constraint [ ... ] ]
      [, ... ]
] )
    SERVER server_name
  [ OPTIONS ( option 'value' [, ... ] ) ]
```

- 删除OSS Foreign Table

```
DROP FOREIGN TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

参数选项

OPTIONS参数请参见以下两张表。更多信息，请参见CREATE FOREIGN TABLE。

通用选项：

选项	类型	是否必选	备注
filepath	字符串	必选，三选一。 说明 均为相对于bucket的路径。	只匹配一个文件。
prefix	字符串		无
dir	字符串		无
bucket	字符串	可选	OSS Server 和 OSS Table 必须有一个设置该选项且 Oss Table 的优先级高，会覆盖Oss Server 的值。

选项	类型	是否必选	备注
format	字符串	必选	指定文件格式，取值范围如下： • CSV • TEXT • ORC • PARQUET • JSON，了解JSON的规范，请参见JSON 规范。 • JSONLINE，可以理解为以换行符分隔的JSON，所有能被JSONLINE读取的数据一定可以用JSDON读取，反之则不一定。在可行的情况下，推荐使用JSONLINE。更多JSONLINE规范，请参见JSONLINE。

CSV和TEXT格式选项：

> **说明**
> 以下参数若不特殊说明仅适用CSV和TEXT格式，ORC和PARQUET等格式设置无效。

选项	类型	是否必选	默认值	备注
filetype	字符串	可选	plain	取值范围如下： • plain：按字节二进制读取，不做额外处理。 • gzip：读取原始二进制数据并gzip解压缩。 • snappy：读取原始二进制数据并snappy解压缩。（只支持标准格式的snappy压缩文件，暂不支持 hadoop-snappy压缩文件）。 json/jsonline 中也可以使用这个字段指定输入文件的压缩类型，当前仅支持 plain，gzip。
log_errors	布尔型	可选	false	是否将错误记录到日志文件。 更多内容，请参见容错机制。

选项	类型	是否必选	默认值	备注
segment_reject_limit	数值	可选	无	设置error abort的上限数值。当包含%号时表示错误行百分比，否则表示错误行数。 例如： • segment_reject_limit = '10' 表示当segment上错误的行数超过10行时，error abort； • segment_reject_limit = '10%' 表示当segment上错误的行数超过已处理行数的10%时，error abort。
以下为格式化选项，参考COPY命令。				
header	布尔型	可选	false	源文件是否包含字段名 header 行。仅适用于 csv 格式
delimiter	字符串	可选	• text 格式默认tab键。 • csv 格式默认逗号。	字段分隔符。只允许单字节字符。
quote	字符串	可选	默认双引号。	字段引号 • 仅适用于 csv 格式。 • 只允许单字节字符。
escape	字符串	可选	默认与QUOTE值相同。	声明应该出现在一个匹配 QUOTE 值的数据字符之前的字符 • 只允许单字节的字符。 • 仅适用于 csv 格式。
null	字符串		• text 格式默认\N (backslash-N)。 • csv 格式默认为未被引号引用的空白字符。	指定文件的 NULL 字符串。
encoding	字符串	可选	未指定时，默认为客户端编码。	指定数据文件编码。

选项	类型	是否必选	默认值	备注
force_not_null	布尔型	可选	false	如果为真，则声明字段的值不匹配空字符串。
force_null	布尔型	可选	false	• 如果为真，它声明匹配空字符串的字段的值作为NULL 返回，即使该值加了引号。 • 没有这个选项，只有未加引号的空字符串的字段的值作为 NULL返回。

示例

```
CREATE FOREIGN TABLE x(i int, j int)
SERVER oss_serv OPTIONS (format 'jsonline')
PARTITION BY LIST (j) (
    VALUES('20181218'),
    VALUES('20190101')
);
```

> **说明**
>
> 创建 OSS Foreign Table 后，可以通过如下方式查看，OSS 外表匹配的 OSS 文件列表是否符合预期，借以校正。
>
> - 方式一：`explain verbose select * from OSS外表;`
> - 方式二：`select * from get_oss_table_meta('OSS外表');`

容错机制

创建OSS Foreign Table时，通过设置参数 `log_errors` 和 `segment_reject_limit` ，可以使得OSS外表扫描时不因原始数据错误行而退出，具有一定的容错机制。其中：

- `log_errors` ：表示是否记录错误行信息。
- `segment_reject_limit` ：表示容错比例，即错误行在已解析行数中的占比。

> **说明**
>
> 当前仅CSV/TEXT格式的OSS外表支持容错机制。

操作步骤如下：

1. 创建容错OSS FDW外表。

```
CREATE FOREIGN TABLE oss_error_sales (id int, value float8, x text)
    server oss_serv
    options (log_errors 'true',          -- 记录错误行信息
             segment_reject_limit '10', -- 错误行数不得超过10行，否则会报错退出。
             dir 'error_sales/',         -- 指定外表匹配的 OSS 文件目录
             format 'csv',               -- 指定按 csv 格式解析文件
             encoding 'utf8');           -- 指定文件编码
```

2. 扫描新建的外表。

> **说明**
>
> 为了显示为容错效果，这里在OSS文件中添加了3行错误记录。

查询示例如下：

```
SELECT * FROM oss_error_sales ;
```

返回结果如下：

```
 id |     value     |     x
----+---------------+-----------
  1 |  0.1102213212 | abcdefg
  1 |  0.1102213212 | abcdefg
  2 | 0.92983182312 | mmsmda123
  3 |     0.1123123 | abbs
  1 |  0.1102213212 | abcdefg
  2 | 0.92983182312 | mmsmda123
  3 |     0.1123123 | abbs
  1 |  0.1102213212 | abcdefg
  1 |  0.1102213212 | abcdefg
  2 | 0.92983182312 | mmsmda123
  3 |     0.1123123 | abbs
  3 |     0.1123123 | abdsa
  1 |  0.1102213212 | abcdefg
  2 | 0.92983182312 | mmsmda123
  3 |     0.1123123 | abbs
  1 |  0.1102213212 | abcdefg
  2 | 0.92983182312 | mmsmda123
  3 |     0.1123123 | abbs
(18 rows)
```

3. 查看错误行日志。

查询示例如下：

```
SELECT * FROM gp_read_error_log('oss_error_sales');
```

返回结果如下：

```
           cmdtime            |     relname     |        filename         | linenum | b
ytenum |                        errmsg                             | rawdata | rawbytes
------------------------------+-----------------+-------------------------+---------+--
-------+-----------------------------------------------------------+---------+---------
-
 2020-04-22 19:37:35.21125+08 | oss_error_sales | error_sales/sales.2.csv |       2 |
| invalid byte sequence for encoding "UTF8": 0xed 0xab 0xad |         | \x
 2020-04-22 19:37:35.21125+08 | oss_error_sales | error_sales/sales.2.csv |       3 |
| invalid byte sequence for encoding "UTF8": 0xed 0xab 0xad |         | \x
 2020-04-22 19:37:35.21125+08 | oss_error_sales | error_sales/sales.3.csv |       2 |
| invalid byte sequence for encoding "UTF8": 0xed 0xab 0xad |         | \x
(3 rows)
```

4. 删除错误行日志。

   查询示例如下：

```
SELECT gp_truncate_error_log('oss_error_sales');
```

   返回结果如下：

```
 gp_truncate_error_log
-----------------------
 t
(1 row)
```

使用OSS Foreign Table

1. 导入OSS数据到本地表，具体步骤如下：

   i. 将数据均匀分散存储在多个OSS文件中。

   > 说明
   >
   > - AnalyticDB PostgreSQL的每个数据分区（Segment）将按轮询方式并行对OSS上的数据文件进行读取。
   > - 推荐数据文件的编码和数据库编码保持一致，减少编码转换，提高效率。数据库编码默认UTF-8。
   > - 对于CSV和TEXT格式的文本文件，支持多文件并行读取，默认并行数4。
   > - 文件的数目建议为数据节点数（Segment个数 × 单个Segment核数）的整数倍，从而提升读取效率。
   > - 当文件数过少时，推荐拆分多个源文件，以便于使用多节点并行扫描功能，请参见大文件切分。

   ii. 在AnalyticDB PostgreSQL中，创建OSS Foreign Table。

iii. 执行如下操作，并行导入数据。

- INSERT方式：

```
INSERT INTO <本地目标表> SELECT * FROM <OSS Foreign Table>;
```

- CREATE TABLE AS 方式

```
CREATE TABLE <本地目标表> AS SELECT * FROM <OSS Foreign Table>;
```

示例如下：

- 示例一：使用INSERT将oss_lineitem_orc数据导入到本地AOCS表。

  创建本地aocs_lineitem表，示例如下：

```
CREATE TABLE aocs_lineitem (
    l_orderkey bigint,
    l_partkey bigint,
    l_suppkey bigint,
    l_linenumber bigint,
    l_quantity double precision,
    l_extendedprice double precision,
    l_discount double precision,
    l_tax double precision,
    l_returnflag CHAR(1),
    l_linestatus CHAR(1),
    l_shipdate DATE,
    l_commitdate DATE,
    l_receiptdate DATE,
    l_shipinstruct CHAR(25),
    l_shipmode CHAR(10),
    l_comment VARCHAR(44)
) WITH (APPENDONLY=TRUE, ORIENTATION=COLUMN, COMPRESSTYPE=ZSTD, COMPRESSLEVEL=5);
```

  将oss_lineitem_orc数据导入到aocs_lineitem本地表，示例如下：

```
INSERT INTO aocs_lineitem SELECT * FROM oss_lineitem_orc;
```

- 示例二：使用CREATE TABLE AS将oss_lineitem_orc导入到本地heap表。

```
CREATE TABLE heap_lineitem AS SELECT * FROM oss_lineitem_orc DISTRIBUTED BY (l_orderkey);
```

2. 查询分析OSS数据。

   OSS Foreign Table创建后，可以像本地表一样查询。常见查询常见如下：

- 键值过滤查询

```
SELECT * FROM oss_lineitem_orc WHERE l_orderkey = 14062498;
```

- 聚集查询

```
SELECT count(*) FROM oss_lineitem_orc WHERE l_orderkey = 14062498;
```

- 过滤+分组+LIMIT

```
SELECT l_partkey, sum(l_suppkey)
  FROM oss_lineitem_orc
 GROUP BY l_partkey
 ORDER BY l_partkey
 limit 10;
```

3. OSS 外表与本地表关联分析。

   本地AOCS表aocs_lineitem与OSS外表关联执行TPC-H Q3，示例如下：

```
SELECT l_orderkey,
       sum(l_extendedprice * (1 - l_discount)) as revenue,
       o_orderdate,
       o_shippriority
  FROM oss_customer,                                    -- OSS 外表
       oss_orders,                                      -- OSS 外表
       aocs_lineitem                                    -- 本地 AOCS 表
 WHERE c_mktsegment = 'furniture'
   and c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and o_orderdate < date '1995-03-29'
   and l_shipdate > date '1995-03-29'
 GROUP BY l_orderkey, o_orderdate, o_shippriority
 ORDER BY revenue desc, o_orderdate
 limit 10;
```

使用OSS外表分区表

在OSS外表基础上，ADB PG还支持分区功能，当分区列出现在查询WHERE条件时，可以显著降低从远端拖取的数据量。

ADB PG OSS FDW 分区功能的使用对数据文件组织有一定的要求。具体来说，对于分区外表下一个特定的分区，该分区在 OSS 上的数据文件要位于 `oss://bucket/partcol1=partval1/partcol2=partval2/` 这样的目录下，其中 `partcol1` 、 `partcol2` 为分区列， `partval1` 、 `partval2` 定义了该分区，为该分区对应的分区列值。

语法

- 创建分区外表

  ADB PG的OSS FDW分区语法与定义普通分区表时采用的语法完全一致。普通分区表定义语法，请参见表分区定义。

  > **说明**
  >
  > 当前ADB PG的OSS FDW只支持值（LIST）分区。

- 删除分区外表

  与常规的外表一样，可通过 DROP FOREIGN TABLE 来删除一个分区外表。

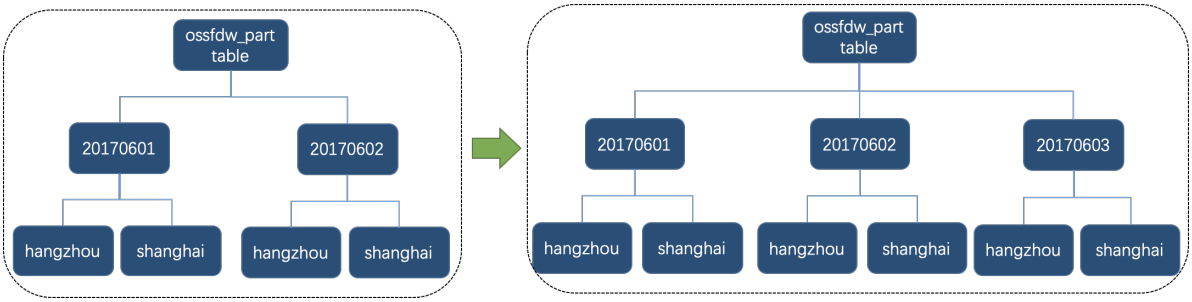
- 调整分区外表结构

  您可通过 ALTER TABLE 命令完成一个已有外表分区的结构调整工作。当前支持：增加分区，删除原有分区。请参见ALTER TABLE文档了解详细的语法定义，本文通过例子来演示如何删除或新增分区：

```
CREATE FOREIGN TABLE ossfdw_parttable(
  key text,
  value bigint,
  pt text,                                          -- 一级分区键
  region text                                       -- 二级分区键
)
SERVER oss_serv
OPTIONS (dir 'PartationDataDirInOss/', format 'jsonline')
PARTITION BY LIST (pt)                              -- 一级分区以"pt"字段为分区键
SUBPARTITION BY LIST (region)                       -- 二级分区以"region"字段为分区键
    SUBPARTITION TEMPLATE (                         -- 二级分区模板
       SUBPARTITION hangzhou VALUES ('hangzhou'),
       SUBPARTITION shanghai VALUES ('shanghai')
    )
( PARTITION "20170601" VALUES ('20170601'),
  PARTITION "20170602" VALUES ('20170602'));
```

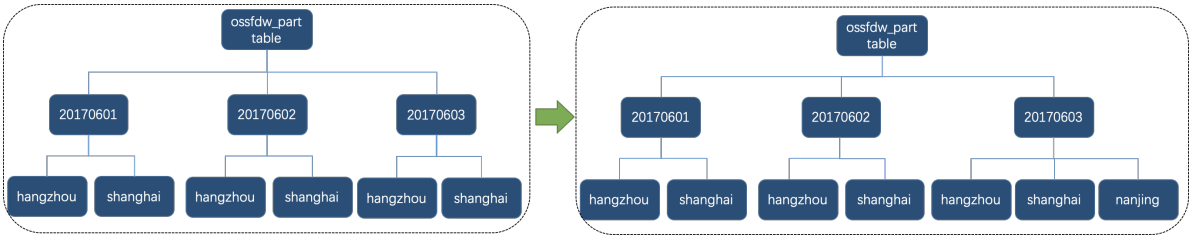
在如上外表分区的基础上运行如下 ALTER TABLE 命令会为 ossfdw_parttable 新增一个一级分区，再结合 ossfdw_parttable 中分区模板的定义，会自动为这个新增一级分区创建对应的子分区。

```
ALTER TABLE ossfdw_parttable ADD PARTITION VALUES ('20170603');
```



在已有一级分区下更新二级子分区：

```
ALTER TABLE ossfdw_parttable ALTER PARTITION FOR ('20170603') ADD PARTITION VALUES('nanji
ng');
```



删除一级分区。

```
ALTER TABLE ossfdw_parttable DROP PARTITION FOR ('20170601');
```

删除二级分区。

```
ALTER TABLE ossfdw_parttable ALTER PARTITION FOR ('20170602') DROP PARTITION FOR ('hangzh
ou');
```

在创建或者新增外表分区时，也可以不使用模板来灵活地定义分区结构，如下所示：

```
CREATE FOREIGN TABLE ossfdw_parttable(
  key text,
  value bigint,
  pt text,                                          -- 一级分区键
  region text                                       -- 二级分区键
)
SERVER oss_serv
OPTIONS (dir 'PartationDataDirInOss/', format 'jsonline')
PARTITION BY LIST (pt)                              -- 一级分区以"pt"字段为分区键
SUBPARTITION BY LIST (region)                       -- 二级分区以"region"字段为分区键
(
    -- 如下两个一级分区下面可以挂着不同的二级分区.
    VALUES('20181218')
    (
        VALUES('hangzhou'),
        VALUES('shanghai')
    ),
    VALUES('20181219')
    (
        VALUES('nantong'),
        VALUES('anhui')
    )
);
```

新增一级分区，此时由于没有模板，所以也需要详细指定一级分区下的二级分区，示例如下：

```
ALTER TABLE ossfdw_parttable ADD PARTITION VALUES ('20181220')
(
    VALUES('hefei'),
    VALUES('guangzhou')
);
```

使用场景

外表分区表的一个典型场景是OSS FDW 访问 SLS 投递数据。业务App写入数据到OSS时也可以使用类似目录组织，然后在定义外表时使用分区表。

示例

如下 SQL 创建的外表 userlogin 具有两个分区，这两个分区在 OSS 文件路径分别为 `oss://bucket/userlogin/month=201812/` 与 `oss://bucket/userlogin/month=201901/`

```
CREATE FOREIGN TABLE userlogin (
        uid integer,
        name character varying,
        source integer,
        logindate timestamp without time zone,
        month int
) SERVER oss_serv OPTIONS (
    dir 'userlogin/',
    format 'text'
)
PARTITION BY LIST (month)
(
        VALUES ('201812'),
        VALUES ('201901')
)
```

使用OSS Foreign Table导出数据

ADB PG支持通过外表向OSS导出数据。目前，OSS外表导出多种格式的数据文件，适用于不同的业务场景。

- 支持导出 csv、text格式的非压缩文本文件。
- 支持导出 csv、text格式的 gzip 压缩文件。
- 支持导出 orc 格式的二进制文件。ORC数据类型与ADBPG数据类型的映射关系请参见ORC - ADB PG 数据类型对照表。

> **说明**
>
> 目前不支持导出分区外表。

导出文件命名规则

导出时，多个segment并行将数据写出到相同的目录下，因此，OSS外表导出文件命名规则如下：

```
{tablename | prefix } _{timestamp}_{random_key}_{seg}{segment_id}_{fileno}.{ext}[.gz]
```

- {tablename | prefix}：导出前缀，`prefix` 时以设置的 `prefix` 为前缀；`dir` option时，以OSS外表名称为默认前缀。
- {timestamp}：导出时的时间戳，格式如 `YYYYMMDDHH24MISS` 。
- {random_key}：随机键值。
- {seg}{segment_id}：“seg” + “segment节点号”。如：“seg1”表明该文件由segment 1导出。
- {fileno}：文件段序号，从0开始。
- {ext}：表示导出的文件格式。如 “.csv”，“.txt”，“.orc” 分别对应 `format` option 设置为 “csv”，“text”，“orc”时导出的文件格式。
- [.gz]：表示导出文件为gzip压缩文件。

- 创建外表，指定格式为CSV，用gzip压缩，并用dir指定路径

```
CREATE FOREIGN TABLE fdw_t_out_1(a int)
SERVER oss_serv
OPTIONS (format 'csv', filetype 'gzip', dir 'test/');
```

- 导出的OSS文件名，示例如下：

```
fdw_t_out_1_20200805110207_1718599661_seg-1_0.csv.g
```

- 创建外表，指定格式为orc，并用prefix指定前缀路径。

```
CREATE FOREIGN TABLE fdw_t_out_2(a int)
SERVER oss_serv
OPTIONS (format 'orc', prefix 'test/my_orc_test');
```

- 导出的OSS文件名，示例如下：

```
my_orc_test_20200924153043_1737154096_seg0_0.orc
```

参数选项

使用ADB PG外表进行数据导出时，外表的option，请参见创建OSS Foreign Table。

除此之外，用外表进行导出时，可以使用特有的option。

通用选项

选项	类型	单位	是否必选	默认值	备注
fragment_size	数值	MB	否	256	指定导出数据到OSS时，切分文件段的大小。当写入文件中的数据量大小超过表定义的fragment_size时，则会切分到新的文件段，向新文件段中继续写入数据。

> **说明**
>
> - fragment指的是导出时的一个文件分片，每个fragment都是完整且独立的文件，可以被ADB PG外表正常解析。同一行记录不会跨fragment存储。
> - fragment文件大小不会严格按照fragment_size设置切分，一般略大于fragment_size设定。
> - **若无特殊需求，直接使用默认值即可，无需修改fragment_size。**

fragment_size使用示例如下：

1. 创建一个外表，格式为CSV，导出目录为*test/lineitem/*，指定framgment_size为512MB。

```
CREATE FOREIGN TABLE oss_lineitem (
    l_orderkey bigint,
    l_partkey bigint,
    l_suppkey bigint,
    l_linenumber bigint,
    l_quantity double precision,
    l_extendedprice double precision,
    l_discount double precision,
    l_tax double precision,
    l_returnflag CHAR(1),
    l_linestatus CHAR(1),
    l_shipdate DATE,
    l_commitdate DATE,
    l_receiptdate DATE,
    l_shipinstruct CHAR(25),
    l_shipmode CHAR(10),
    l_comment VARCHAR(44)
) server oss_serv
    options (
        dir 'test/lineitem/',
        format 'csv',
        fragment_size '512' -- 每当超过512MB时，切分新的文件段
    );
```

2. 从本地表数据写入到外表中。

```
INSERT INTO oss_lineitem SELECT * FROM lineitem;
```

3. 导出结束后，查看OSS上的文件，可以看到大部分文件分片大小略大于512MB。

   命令如下：

```
ossutil -e endpoint -i id -k key ls oss://bucket/test/lineitem
```

   返回示例如下：

```
LastModifiedTime                    Size(B)  StorageClass   ETAG
ObjectName
2020-09-24 14:12:01 +0800 CST    536875660      Standard   ED6C68093E738D09B1386C5F0000
0000      oss://adbpg-tpch/test/lineitem/oss_lineitem2_20200924140843_1702924182_seg15_
7.csv
2020-09-24 14:12:27 +0800 CST    536875604      Standard   FD25FA7C7109ABCDCB386C5F0000
0000      oss://adbpg-tpch/test/lineitem/oss_lineitem2_20200924140843_1702924182_seg15_
8.csv
2020-09-24 14:12:53 +0800 CST    536875486      Standard   7C3EDE6AFE354190E5386C5F0000
0000      oss://adbpg-tpch/test/lineitem/oss_lineitem2_20200924140843_1702924182_seg15_
9.csv
2020-09-24 14:09:07 +0800 CST    536875626      Standard   48B38E65A5BB8B5B03386C5F0000
0000      oss://adbpg-tpch/test/lineitem/oss_lineitem2_20200924140843_1702924182_seg1_0
.csv
2020-09-24 14:09:32 +0800 CST    536875858      Standard   AF5525D81166F02D1C386C5F0000
0000      oss://adbpg-tpch/test/lineitem/oss_lineitem2_20200924140843_1702924182_seg1_1
.csv
2020-09-24 14:13:08 +0800 CST    235457368      Standard   BF1FC0B81376AE14F4386C5F0000
0000      oss://adbpg-tpch/test/lineitem/oss_lineitem2_20200924140843_1702924182_seg1_1
0.csv
2020-09-24 14:09:56 +0800 CST    536875899      Standard   20C824EBCAE2C5DB34386C5F0000
0000      oss://adbpg-tpch/test/lineitem/oss_lineitem2_20200924140843_1702924182_seg1_2
.csv
...
```

命令如下：

```
ossutil -e endpoint -i id -k key du oss://adbpg-tpch/test/lineitem/
```

返回示例如下：

```
storage class   object count            sum size(byte)
------------------------------------------------------------
Standard       176                  89686183118
------------------------------------------------------------
total object count: 176                  total object sum size: 89686183118
total part count:   0                          total part sum size:   0

total du size(byte):89686183118

0.051899(s) elapsed
```

CSV和TEXT选项

> **注意**
>
> 以下选项若不特殊说明仅适用csv/text格式导出时使用，orc格式设置无效。

选项	类型	单位	是否必选	默认值	备注
gzip_level	数值	无	否	1	指定导出CSV/TEXT数据到OSS时，使用的gzip压缩级别，最高压缩级别为9。
force_quote_all	布尔型	无	否	false	指定导出CSV数据时，是否将所有字段都强制引用。 force_quote_all只对CSV格式有效。

> **说明**
>
> - gzip_level 仅在filetype设置为 `gzip` 时生效。
> - gzip_level 压缩级别越高，导出的数据文件越小，但导出时间越长。
> - 根据测试结果，gzip_level设置越大，文件大小并没有明显下降，但导出时间明显增加，因此若无特殊需求，推荐gzip_level使用默认值1。

gzip_level使用示例如下：

1. 创建一个外表，格式为CSV，导出目录为*test/lineitem2/*，指定gzip_level为9。

```
CREATE FOREIGN TABLE oss_lineitem3 (
    l_orderkey bigint,
    l_partkey bigint,
    l_suppkey bigint,
    l_linenumber bigint,
    l_quantity double precision,
    l_extendedprice double precision,
    l_discount double precision,
    l_tax double precision,
    l_returnflag CHAR(1),
    l_linestatus CHAR(1),
    l_shipdate DATE,
    l_commitdate DATE,
    l_receiptdate DATE,
    l_shipinstruct CHAR(25),
    l_shipmode CHAR(10),
    l_comment VARCHAR(44)
) server oss_serv
    options (
        dir 'test/lineitem2/',
        format 'csv', filetype 'gzip' ,gzip_level '9'
    );
```

2. 从本地表数据写入到外表中。

```
INSERT INTO oss_lineitem SELECT * FROM lineitem;
```

3. 导出结束后，查看OSS上的文件，可以看到文件总大小为23GB (25141481880Byte)左右，远小于fragment_size使用示例中的83GB(89686183118 Byte)。

   命令如下：

   ```
   ossutil -e endpoint -i id -k key ls oss://bucket/test/lineitem2
   ```

   返回示例如下：

   ```
   2020-09-24 15:07:49 +0800 CST    270338060      Standard   9B1F7D7CB0748391C5456C5F0000
   0000      oss://adbpg-tpch/test/lineitem2/oss_lineitem3_20200924145900_1220405754_seg8_
   3.csv.gz
   2020-09-24 15:10:08 +0800 CST    270467652      Standard   17DE92CAE57F64F550466C5F0000
   0000      oss://adbpg-tpch/test/lineitem2/oss_lineitem3_20200924145900_1220405754_seg8_
   4.csv.gz
   2020-09-24 15:12:00 +0800 CST    219497966      Standard   FCFE4E457C0F6942C0466C5F0000
   0000      oss://adbpg-tpch/test/lineitem2/oss_lineitem3_20200924145900_1220405754_seg8_
   5.csv.gz
   2020-09-24 15:01:10 +0800 CST    270617343      Standard   13AD24EF528ECDF836446C5F0000
   0000      oss://adbpg-tpch/test/lineitem2/oss_lineitem3_20200924145900_1220405754_seg9_
   0.csv.gz
   2020-09-24 15:03:18 +0800 CST    270377032      Standard   759DBF9B999F8609B6446C5F0000
   0000      oss://adbpg-tpch/test/lineitem2/oss_lineitem3_20200924145900_1220405754_seg9_
   1.csv.gz
   2020-09-24 15:05:28 +0800 CST    270284091      Standard   F9896A5CFF554F2838456C5F0000
   0000      oss://adbpg-tpch/test/lineitem2/oss_lineitem3_20200924145900_1220405754_seg9_
   2.csv.gz
   2020-09-24 15:07:43 +0800 CST    270350284      Standard   C120BE98B47DAD5EBF456C5F0000
   0000      oss://adbpg-tpch/test/lineitem2/oss_lineitem3_20200924145900_1220405754_seg9_
   3.csv.gz
   2020-09-24 15:10:04 +0800 CST    270477777      Standard   69B9B1E854B626364C466C5F0000
   0000      oss://adbpg-tpch/test/lineitem2/oss_lineitem3_20200924145900_1220405754_seg9_
   4.csv.gz
   2020-09-24 15:12:00 +0800 CST    219358236      Standard   A4EB5DFFBD67AF6BC0466C5F0000
   0000      oss://adbpg-tpch/test/lineitem2/oss_lineitem3_20200924145900_1220405754_seg9_
   5.csv.gz
   ....

   Object Number is: 96
   ```

   命令如下：

   ```
   ossutil -e endpoint -i id -k key du oss://adbpg-tpch/test/lineitem/
   ```

   返回示例如下：

```
storage class    object count            sum size(byte)
-----------------------------------------------------------
Standard        96                   25141481880
-----------------------------------------------------------
total object count: 96                   total object sum size: 25141481880
total part count:   0                           total part sum size:   0

total du size(byte):25141481880

0.037620(s) elapsed
```

force_quote_all使用示例如下：

1. 创建一个外表，格式为CSV，导出目录为*test/lineitem/*，指定force_quote_all为true。

```
CREATE FOREIGN TABLE foreign_x (
  a int, b text
) server oss_serv
    options (
        dir 'test/x/',
        format 'csv', force_quote_all 'true'
    );
```

2. 从本地表数据写入到外表中。

```
INSERT INTO foreign_x values(1, 'a'), (2, 'b');
```

3. 导出完成后，查看OSS上的文件内容，可以看到每一列的值都被双引号引用。

   命令如下：

```
cat foreign_x_20200923173618_447789894_seg4_0.csv
```

   返回信息如下：

```
"1","a"
```

   命令如下：

```
cat foreign_x_20200923173618_447789894_seg5_0.csv
```

   返回信息如下：

```
"2","b"
```

ORC选项

> **注意**
>
> 以下选项若不特殊说明仅适用orc格式导出时使用，csv/text格式设置无效。

选项	类型	单位	是否必选	默认值	备注
orc_stripe_size	数值	MB	否	64	指定导出ORC格式数据时，单个ORC文件中每个Stripe的大小。

> 说明
>
> - orc_stripe_size越小，同等数据量导出生成的ORC文件越大，且导出时间越长。
> - 如无特殊需求，一般不需要更改orc_stripe_size，直接使用默认值64MB即可。

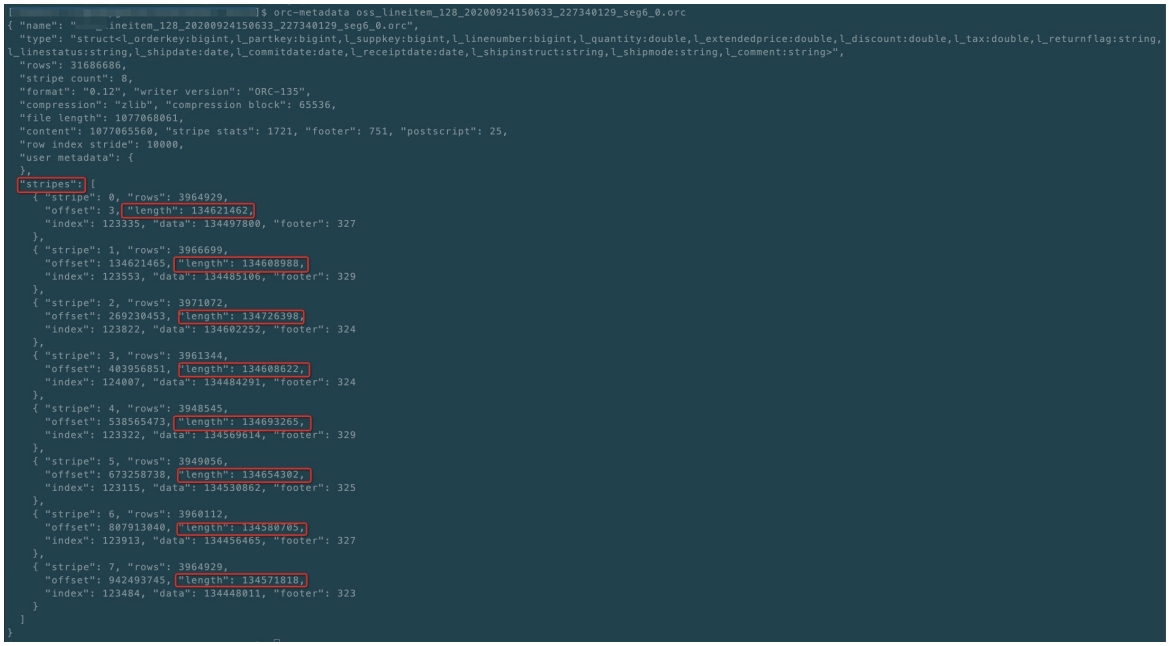
orc_stripe_size使用示例如下：

1. 创建一个外表，格式为ORC，导出目录为test/lineitem/, orc_stripe_size为128 MB。

```
CREATE FOREIGN TABLE oss_lineitem (
    l_orderkey bigint,
    l_partkey bigint,
    l_suppkey bigint,
    l_linenumber bigint,
    l_quantity double precision,
    l_extendedprice double precision,
    l_discount double precision,
    l_tax double precision,
    l_returnflag CHAR(1),
    l_linestatus CHAR(1),
    l_shipdate DATE,
    l_commitdate DATE,
    l_receiptdate DATE,
    l_shipinstruct CHAR(25),
    l_shipmode CHAR(10),
    l_comment VARCHAR(44)
) server oss_serv
options ( dir 'test/lineitem/', format 'orc', orc_stripe_size '128');
```

2. 从本地表数据写入到外表中。

```
INSERT INTO oss_lineitem SELECT * FROM lineitem;
```



3. 导出完成后，查看导出的某个ORC文件的meta信息，可以看到ORC文件中的Stripe大致为128 MB左右。

示例

1. 创建外表

   创建一个外表，格式为CSV，导出目录为tt_csv。

   ```
   CREATE FOREIGN TABLE foreign_x (i int, j int)
   SERVER oss_serv
   OPTIONS (format 'csv', dir 'tt_csv/');
   ```

2. 导出数据

   向外表写入数据和向普通表写入数据一样，可以使用INSERT INTO语句进行写入。

   - 批量导出数据到OSS外表（推荐）

     ```
     INSERT INTO foreign_x SELECT * FROM local_x;
     ```

   - 插入值列表（不推荐）

     ```
     INSERT INTO foreign_x VALUES (1,1), (2,2), (3,3);x;
     ```

大文件切分

OSS FDW支持多节点并行扫描。因此，当文件数过少时，推荐拆分多个源文件，以便于使用多节点并行扫描功能。

例如：Linux平台下，可以使用 **split** 命令对 `text` 或 `csv` 格式的文件按行拆分成多个小文件。

> **说明**
>
> 切分后的小文件需要保证行完整性，即同一行记录只允许存在同一个文件内，不允许跨文件存放。因此，只能通过行切分。

- 获取当前文件行数。

```
wc -l csv_file
```

- 按照指定行数切分成多个小文件，其中，N表示切分后小文件的行数。

```
split -l N csv_file
```

外表统计信息收集

OSS 外表实际数据存储在OSS上，默认不进行数据的统计信息收集。因此针对复杂场景下的查询SQL（比如多张表关联操作）如果统计信息不存在或者信息过时，优化器可能会生成低效的查询计划。使用ANALYZE语句可以更新统计信息。执行方式如下：

- 使用EXPLAIN查看ANALYZE前的执行计划

```
EXPLAIN 表名;
```

- 使用ANALYZE触发收集统计信息

```
ANALYZE 表名;
```

- 使用EXPLAIN查看ANALYZE后的执行计划

```
EXPLAIN 表名;
```

查看执行计划

执行如下操作，查看OSS Foreign表的执行计划。

```
EXPLAIN SELECT COUNT(*) FROM oss_lineitem_orc WHERE l_orderkey > 14062498;
```

> **说明**
>
> `EXPLAIN VERBOSE` 可查看更多信息。

查看 OSS 文件信息

执行如下操作，获取指定OSS Foreign表的文件信息。

```
SELECT * FROM get_oss_table_meta('<OSS FOREIGN TABLE>');
```

ORC：ADB PG 数据类型对照表

ORC：ADB PG数据类型对照如下：

ADB PG数据类型	ORC数据类型
bool	BOOLEAN

ADB PG数据类型	ORC数据类型
int2	SHORT
int4	INT
int8	LONG
float4	FLOAT
float8	DOUBLE
numeric	DECIMAL
char	CHAR
text	STRING
bytea	BINARY
timestamp	TIMESTAMP
date	DATE
int2[]	LIST(SHORT)
int4[]	LIST(INT)
int8[]	LIST(LONG)
float4[]	LIST(FLOAT)
float8[]	LIST(DOUBLE)

> **说明**
>
> 对于ORC的List类型，目前只支持转换为ADB PG 的一维数组。

PARQUET：ADB PG 数据类型对照表

PARQUET格式文件数据类型包含primitive types和logical types，针对这两种情况，建表时推荐使用下面列举的ADB PG类型对应到PARQUET类型，如果不对应会尝试转换。

> **说明**
>
> 不支持PARQUET中的嵌套类型：ARRAY、MAP。

在PARQUET文件中没有提供logical types情况下，数据类型对照如下：

ADB PG数据类型	PARQUER数据类型
bool	BOOLEAN
integer	INT32
bigint	INT64
timestamp	INT96
float4	FLOAT
float8	DOUBLE
bytea	BYTE_ARRAY
bytea	FIXED_LEN_BYTE_ARRAY

在PARQUET文件中提供了logical types情况下，数据类型对照如下：

ADB PG类型	PARQUET类型
text	STRING
date	DATE

ADB PG类型	PARQUET类型
timestamp	TIMESTAMP
time	TIME
interval	INTERVAL
numeric	DECIMAL
smallint	INT(8)/INT(16)
integer	INT(32)
bigint	INT(64)
bigint	UINT(8/16/32/64)
json	JSON
jsonb	BSON
uuid	UUID
text	ENUM

OSS FDW 访问 SLS 投递数据

ADB PG OSS FDW分区表更常见的一种使用场景是搭配 SLS 一起使用。SLS 是阿里巴巴集团经历大量大数据场景锤炼而成，针对日志类数据的一站式服务，关于 SLS 的更多介绍，请参见什么是日志服务。

1. 如果想基于 SLS 投递到 OSS 上的数据建立 OSS 分区外表，那么只需要在 SLS 投递 OSS 相关配置中按照 SLS 文档设置好“分区格式”这一配置项即可，如下图所示。关于配置项的语义信息，请参见将日志服务数据投递到OSS。

图中配置会将归属于同一月的日志都放在 OSS 同一目录下。该配置生成的 OSS 目录布局如下所示：

```
oss://oss-fdw-test/adbpgossfdw
├── date=202002
│   ├── userlogin_1585617629106546791_647504382.csv
│   └── userlogin_1585617849232201154_647507440.csv
└── date=202003
    └── userlogin_1585617944247047796_647508762.csv
```

2. 之后再结合 SLS 投递上来的文件中包含的字段建立如下 OSS 分区外表：

```
CREATE FOREIGN TABLE userlogin (
        uid integer,
        name character varying,
        source integer,
        logindate timestamp without time zone,
        "date" int
) SERVER oss_serv OPTIONS (
    dir 'adbpgossfdw/',
    format 'text'
)
PARTITION BY LIST ("date")
(
        VALUES ('202002'),
        VALUES ('202003')
)
```

3. 最后便可以在此基础之上做一下业务所需的分析逻辑，如：分析2020年2月份，所有用户的登录次数。

请求示例如下：

```
EXPLAIN SELECT uid, count(uid) FROM userlogin WHERE "date" = 202002 GROUP BY uid;
```

返回信息如下：

```
                                                                                QUERY PLAN

------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------
 Gather Motion 3:1  (slice2; segments: 3)  (cost=5135.10..5145.10 rows=1000 width=12)
   ->  HashAggregate  (cost=5135.10..5145.10 rows=334 width=12)
         Group Key: userlogin_1_prt_1.uid
         ->  Redistribute Motion 3:3  (slice1; segments: 3)  (cost=5100.10..5120.10 row
s=334 width=12)
               Hash Key: userlogin_1_prt_1.uid
               -> HashAggregate  (cost=5100.10..5100.10 rows=334 width=12)
                     Group Key: userlogin_1_prt_1.uid
                     ->t;  Append  (cost=0.00..100.10 rows=333334 width=4)
                           ->  Foreign Scan on userlogin_1_prt_1  (cost=0.00..100.10 ro
ws=333334 width=4)
                                 Filter: (date = 202002)
                                 Oss Url: endpoint=oss-cn-hangzhou-zmf-internal.aliyunc
s.com bucket=adbpg-regress dir=adbpgossfdw/date=202002/ filetype=plain|text
                                 Oss Parallel (Max 4) Get: total 0 file(s) with 0 bytes
byte(s).
 Optimizer: Postgres query optimizer
(13 rows)
```

可以看到这里 OSS FDW 只会从 OSS 处取出 date=202002 对应的数据，而不会下载 date=202003 月份的数据，需要拉取数据量的减小可以大幅提升查询的执行效率。

常见错误处理

Oss Foreign Table扫描过程中，当出现形如下错误提示时。

```
“oss server returned error: StatusCode=..., ErrorCode=..., ErrorMessage="...", RequestId=...
”
```

请参见OSS错误响应中的文档了解和处理各类错误。

其中：

- StatusCode：出错请求的HTTP状态码。
- ErrorCode：OSS的错误码。
- ErrorMessage：OSS的错误信息。
- RequestId：标识该次请求的UUID。当您无法解决问题时，可以提供RequestId寻求OSS开发工程师帮助。

参考文档

- 开始使用阿里云OSS
- OSS访问域名使用规则
- 错误处理
- OSS错误响应

9.使用COPY/UNLOAD导入/导出数据到OSS

AnalyticDB PostgreSQL版支持COPY/UNLOAD命令，COPY表示从外表导入数据到本地表，UNLOAD表示从本地表导出数据到外表。

COPY和UNLOAD都是基于OSS Foreign Table来完成数据导入导出的，OSS Foreign Table的详细内容请参见使用 OSS Foreign Table 访问 OSS 数据 。

注意事项

在使用COPY/UNLOAD导出为CSV文件时，可能存在某些option会因关键字冲突而发生语法错误，需要将option选项用双引号引用，并写成小写形式。需要进行特殊处理的options：`delimiter`、`quote`、`null`、`header`、`escape`、`encoding`。示例如下：

```
UNLOAD ('select * from table')
TO 'path'
ACCESS_KEY_ID 'id'
SECRET_ACCESS_KEY 'key'
FORMAT csv
"delimiter" '|'
"quote" '"'
"null" ''
"header" 'true'
"escape" 'E'
"encoding" 'utf-8'
FDW 'oss_fdw'
ENDPOINT 'endpoint';
```

COPY

语法

```
COPY table-name
[ column-list ]
FROM data_source
ACCESS_KEY_ID '<access-key-id>'
SECRET_ACCESS_KEY '<secret-access-key>'
[ [ FORMAT ] [ AS ] data_format ]
[ MANIFEST ]
[ option 'value' [ ... ] ]
```

- table_name：需要导入数据的本地表名，必须是已经存在的本地表。
- column-list：可选，表示需要写入的目标列列表。若不使用，则默认写入所有列。
- data_source：表示OSS路径的字符串，如*oss://bucket_name/path_prefix*。
- <access-key-id>：表示OSS账号ID。
- <secret-access-key>：表示OSS账号secret key。
- [FORMAT] [AS] data_format： 可选。若不使用，则默认FORMAT AS CSV。data_format可以是BINARY，CSV，JSON，JSONLINE，ORC，PARQUET，TEXT（FORMAT与AS可缺省，即FORMAT AS CSV与FORMAT CSV和CSV等价）。

- [MANIFEST]：表示data_source是manifest清单文件。manifest清单文件必须是JSON格式，由以下元素构成：
  - entries：数组，表示清单内具体的OSS文件列表，可以位于不同的bucket或拥有不同的路径前缀，但需要能够使用相同的ACCESS_KEY_ID或SECRET_ACCESS_KEY访问。示例如下：

```
{   "entries": [
      {"url": "oss://adbpg-regress/local_t/_seg2_0.csv", "mandatory":true},
      {"url": "oss://adbpg-regress/local_t/_seg1_0.csv", "mandatory":true},
      {"url": "oss://adbpg-regress/local_t/_seg0_0.csv", "mandatory":true},
      {"url": "oss://adbpg-regress-2/local_t/_seg1_0.csv", "mandatory":true},
      {"url": "oss://adbpg-regress-2/local_t/_seg2_0.csv", "mandatory":true},
      {"url": "oss://adbpg-regress-2/local_t/_seg0_0.csv", "mandatory":true}
    ]
}
```

  - url：一个OSS文件的完整路径。
  - mandatory：OSS文件不存在时是否报错。
- [option [value] [...]]：选项列表，以key value的形式输入。选项说明见下表：

选项	类型	是否必选	备注
endpoint	字符串	是	指定OSS的endpoint。
fdw	字符串	是	指定oss fdw插件名字。COPY命令在创建临时Server时需要用到。
其他所有创建外表时用到的option，如format，filetype，delimiter，escape等	不涉及	不涉及	创建临时外表时用到的option，详细内容请参见使用 OSS Foreign Table 访问 OSS 数据。

示例1

1. 创建本地表。

```
CREATE TABLE local_t2 (a int, b float8, c text);
```

2. 使用COPY命令导入数据，只写入a和c两列，b列全部写入NULL。

```
COPY local_t2 (a, c)
FROM 'oss://adbpg-regress/local_t/'
ACCESS_KEY_ID 'id'
SECRET_ACCESS_KEY 'key'
FORMAT AS CSV
ENDPOINT '<endpoint>'
FDW 'oss_fdw';
```

查询表中的数据：

```
SELECT * FROM local_t2 LIMIT 10;
```

返回信息如下：

```
 a  | b |                 c
----+---+----------------------------------
 12 |   | a24cba6ebdc5e0c485cd88ef60b72fea
 15 |   | c4d3028f5205fab98e5f43c7945db4ba
 20 |   | 769884311db01f400e21a903a3f1cb50
 26 |   | 7d12c981d262e0067ea1a04368f32f2a
 30 |   | 4e64bda52d54d263d16f42771b1d0225
 35 |   | b70c976d4c04568bd497b42a7d2e451d
 40 |   | d07ce2948b8618b47c351b6e222182f6
 46 |   | c2234393f878f5557776b7e778299564
 47 |   | cde904b2331fa274cd8d9266aa858342
 50 |   | 1235b900fb644bb36440a274314e4b6b
(10 rows)
```

3. 可以看到从外表导入的a和c列，与源数据表local_t的a和c列数据相同。

```
SELECT sum(hashtext(t.a::text)) AS col_a_hash, sum(hashtext(t.c::text)) AS col_c_
hash FROM local_t2 t;
```

返回信息如下：

```
 col_a_hash  | col_c_hash
-------------+-------------
 23725368368 | 13447976580
(1 row)
```

```
SELECT sum(hashtext(t.a::text)) AS col_a_hash, sum(hashtext(t.c::text)) AS col_c_
hash FROM local_t t;
```

返回信息如下：

```
 col_a_hash  | col_c_hash
-------------+-------------
 23725368368 | 13447976580
(1 row)
```

4. 导入其他格式示例。
   - 导入ORC格式数据：

```
COPY tt
FROM 'oss://adbpg-regress/q_oss_orc_list/'
ACCESS_KEY_ID 'id'
SECRET_ACCESS_KEY 'key'
FORMAT AS ORC
ENDPOINT '<endpoint>'
FDW 'oss_fdw';
```

   - 导入PARQUET格式数据：

```
COPY tp
FROM  'oss://adbpg-regress/test_parquet/'
ACCESS_KEY_ID 'id'
SECRET_ACCESS_KEY 'key'
FORMAT AS PARQUET
ENDPOINT '<endpoint>'
FDW 'oss_fdw';
```

示例2

1. 创建本地表。

```
CREATE TABLE local_manifest (a int, c text);
```

2. 创建manifest文件，其中OSS文件列表可位于不同的bucket。

```
{
   "entries": [
      {"url": "oss://adbpg-regress/local_t/_20210114103840_83f407434beccbd4eb2a0ce4
5ef39568_1450404435_seg2_0.csv", "mandatory":true},
      {"url": "oss://adbpg-regress/local_t/_20210114103840_83f407434beccbd4eb2a0ce4
5ef39568_1856683967_seg1_0.csv", "mandatory":true},
      {"url": "oss://adbpg-regress/local_t/_20210114103840_83f407434beccbd4eb2a0ce4
5ef39568_1880804901_seg0_0.csv", "mandatory":true},
      {"url": "oss://adbpg-regress-2/local_t/_20210114103849_67100080728ef95228e662
bc02cb99d1_1008521914_seg1_0.csv", "mandatory":true},
      {"url": "oss://adbpg-regress-2/local_t/_20210114103849_67100080728ef95228e662
bc02cb99d1_1234881553_seg2_0.csv", "mandatory":true},
      {"url": "oss://adbpg-regress-2/local_t/_20210114103849_67100080728ef95228e662
bc02cb99d1_1711667760_seg0_0.csv", "mandatory":true}
   ]
}
```

3. 使用COPY命令从manifest清单中导入本地表。

```
COPY local_manifest
FROM 'oss://adbpg-regress-2/unload_manifest/t_manifest'
ACCESS_KEY_ID '<id>'
SECRET_ACCESS_KEY '<key>'
FORMAT AS CSV
MANIFEST                          -- 表示从manifest文件中导入
ENDPOINT '<endpoint>'
FDW 'oss_fdw';
```

示例3

COPY导入OSS数据时，可能会存在异常的数据行（无法正常COPY导入）。当遇到这种情况时，可以通过额外的option选项设置实现容错。

- log_errors：表示是否记录错误行信息。
- segment_reject_limit：`segment_reject_limit '10'` 表示最多容忍10行，大于等于10行时报错退出；`segment_reject_limit '10%'` 表示当前的错误总行数/当前总共已处理的行 >= 10% 时，报错退出。

1. 创建本地表。

```
CREATE TABLE sales(id integer, value float8, x text) DISTRIBUTED BY (id);
```

2. 使用COPY导入OSS数据文件，文件中有3行数据存在编码问题。

```
COPY sales
FROM 'oss://adbpg-const/error_sales/'
ACCESS_KEY_ID '<id>'
SECRET_ACCESS_KEY '<key>'
FORMAT AS csv
log_errors 'true'                -- 表示需要记录错误行的信息
segment_reject_limit '10'        -- 表示最多容忍10行错误记录，超过10行时报错退出
endpoint '<endpoint>'
FDW 'oss_fdw';
NOTICE:  found 3 data formatting errors (3 or more input rows), rejected related in
put data
COPY FOREIGN TABLE
```

3. 查看具体的错误行信息。

```
SELECT * FROM gp_read_error_log('<COPY的目标表名>');
```

例如，查看sales表的错误行信息：

```
SELECT * FROM gp_read_error_log('sales');
```

返回信息如下：

```
            cmdtime            |                    relname                     |
filename          | linenum | bytenum |                            errmsg
| rawdata | rawbytes
-------------------------------+------------------------------------------------+--
----------------------+---------+---------+----------------------------------------
--------------------+---------+----------
 2021-02-08 14:24:04.225238+08 | adbpgforeigntabletmp_20210208142403_1936866966 | e
rror_sales/sales.2.csv |       2 |         | invalid byte sequence for encoding "UT
F8": 0xed 0xab 0xad |         | \x
 2021-02-08 14:24:04.225238+08 | adbpgforeigntabletmp_20210208142403_1936866966 | e
rror_sales/sales.2.csv |       3 |         | invalid byte sequence for encoding "UT
F8": 0xed 0xab 0xad |         | \x
 2021-02-08 14:24:04.225269+08 | adbpgforeigntabletmp_20210208142403_1936866966 | e
rror_sales/sales.3.csv |       2 |         | invalid byte sequence for encoding "UT
F8": 0xed 0xab 0xad |         | \x
(3 rows)
```

> **说明** 追加保存的错误行日志，需要占用一定的存储空间。删除错误行日志语法为：`select gp_truncate_error_log('<表名>')`。

UNLOAD

语法

```
UNLOAD ('select-statement')
TO destination_url
ACCESS_KEY_ID '<access-key-id>'
SECRET_ACCESS_KEY '<secret-access-key>'
[ [ FORMAT ] [ AS ] data_format ]
[ MANIFEST [ '<manifest_url>' ] ]
[ PARALLEL [ { ON | TRUE } | { OFF | FALSE } ] ]
[ option 'value' [ ... ] ]
```

参数说明：

- select-statement：一个select查询语句，查询的结果数据会被写到OSS。
- destination_url：OSS路径的字符串，如*oss://bucket-name/path-prefix*。
- <access-key-id>： OSS账号ID。
- <secret-access-key>： OSS账号KEY。
- [FORMAT] [AS] data_format： 可选。若不使用，则默认FORMAT AS CSV。data_format可以是CSV，ORC，TEXT（FORMAT与AS可缺省，即FORMAT AS CSV与FORMAT CSV和CSV等价）。
- MANIFEST：表示导出时需要生成导出清单文件。<manifest_url>可以指定与数据文件不同bucket的OSS完整路径，且路径需要以manifest为后缀。不指定<manifest_url>时，表示清单文件与导出的数据文件有相同的路径前缀。

  > **说明** 如果<manifest_url>指定的文件已经存在，则需要在[option [value] [...]]选项列表中指定allowoverwrite选项为true， 表示覆盖原manifest文件。

- PARALLEL：表示是否多segment并行导出。默认多节点并行导出，每个节点生成独立的导出文件。值设置为OFF/FALSE时，表示关闭并行，导出数据大小不超过8GB时，仅导出一个文件。
- [option [value] [...]]：选项列表，以key value的形式输入。选项说明见下表：

选项	类型	是否必选	备注
endpoint	字符串	是	指定OSS的endpoint。
fdw	字符串	是	指定oss fdw插件名字。UNLOAD命令在创建临时Server时需要用到。
allowoverwrite	布尔类型	否	指定是否覆写已存在的manifest文件。 **说明** 仅覆写manifest文件，不会覆写数据文件。
其他所有创建外表时用到的option，如format，filetype，delimiter，escape等	不涉及	不涉及	创建临时外表时用到的option，详细内容请参见使用 OSS Foreign Table 访问 OSS 数据。

示例1

1. 创建本地表并写入测试数据。

```
CREATE TABLE local_t (a int, b float8, c text);
INSERT INTO local_t SELECT r, random() * 1000, md5(random()::text) FROM generate_se
ries(1,1000)r;
```

查询本地表：

```
SELECT * FROM local_t LIMIT 5;
```

返回信息如下：

```
 a  |        b         |                c
----+------------------+----------------------------------
  5 |  550.81393988803 | 8009fa725372e996786849213a695ce0
  6 | 95.8335199393332 | ce7952c6728cdffdee06cc5b502d6457
  9 | 421.379795763642 | d3260ccbf6b9c03f3658d96bb7678b4d
 10 | 362.347379792482 | 2bbbf89d23a2f83b089b589f55b5c4fc
 11 | 800.203878898174 | a52994c5573e6b36d8a1c357bf800ce5
(5 rows)
```

2. 使用UNLOAD，导出指定列数据到OSS。

```
UNLOAD ('select a, c from local_t') TO 'oss://adbpg-regress/local_t/'
ACCESS_KEY_ID '<id>'
SECRET_ACCESS_KEY '<key>'
FORMAT AS CSV
ENDPOINT '<endpoint>'
FDW 'oss_fdw';
NOTICE:  OSS output prefix: "local_t/adbpgforeigntabletmp_20200907164801_1354519958
_20200907164801_652261618".
UNLOAD
```

3. 查看OSS对应路径下已经写入的csv文件。

```
$ ossutil --config hangzhou-zmf.config ls oss://adbpg-regress/local_t/
```

返回信息如下

```
LastModifiedTime                   Size(B)  StorageClass   ETAG
ObjectName
2020-09-07 16:48:01 +0800 CST        12023      Standard   9F38B5407142C044C1F3555F
00000000      oss://adbpg-regress/local_t/adbpgforeigntabletmp_20200907164801_13545
19958_20200907164801_652261618_seg0_0.csv
2020-09-07 16:48:01 +0800 CST        12469      Standard   807BA680A0DED49BC1F3555F
00000000      oss://adbpg-regress/local_t/adbpgforeigntabletmp_20200907164801_13545
19958_20200907164801_652261618_seg1_0.csv
2020-09-07 16:48:01 +0800 CST        12401      Standard   3524F68F628CEB64C1F3555F
00000000      oss://adbpg-regress/local_t/adbpgforeigntabletmp_20200907164801_13545
19958_20200907164801_652261618_seg2_0.csv
Object Number is: 3
0.153414(s) elapsed
```

查看文件数据，只写出了a和c两列的数据。

```
$ head -n 10 adbpgforeigntabletmp_20200907164801_1354519958_20200907164801_652261618_seg2_0.csv
```

返回信息如下：

```
7,1225341d0d367a69b1b345536b21ef73
19,424a7a5c36066842f4de8c8a8341fc89
27,c214432e9928e4a6f7bef7bd815424c0
29,ade5d636e2b5d2a606a02e79255da4bd
37,85660e60ede47b68493f6295620db568
77,e1be448ba2b08f0a2ca05b7ed812abfd
80,5e85d597a3b0f2f9736a728724a0f9e0
92,dc23f76f0b1446504b8f1c2274521d2f
94,50304822488d55a500e3a71bcf40890f
97,e970fde8cd0df9c6b610925a488f6042
```

示例2

1. 使用UNLOAD导出时自动生成manifest文件（manifest文件与数据文件有相同的路径前缀）。

```
UNLOAD ('select * from local_t') TO 'oss://adbpg-regress/local_t/'
ACCESS_KEY_ID '<id>'
SECRET_ACCESS_KEY '<key>'
FORMAT AS CSV
MANIFEST    -- 表示UNLOAD导出时生成导出清单，清单文件与数据文件有相同的路径前缀
ENDPOINT '<endpoint>'
FDW 'oss_fdw';
NOTICE:  OSS output prefix: "local_t/_20210114100329_3e9b07726306d88b3193dc95c10a5c5c".
UNLOAD
```

查看导出的文件列表：

```
ossutil ls -s oss://adbpg-regress/local_t/
```

除了数据文件外，还有一个清单文件，返回信息如下：

```
oss://adbpg-regress/local_t/_20210114100329_3e9b07726306d88b3193dc95c10a5c5c_162488956_seg1_0.csv
oss://adbpg-regress/local_t/_20210114100329_3e9b07726306d88b3193dc95c10a5c5c_163756258_seg0_0.csv
oss://adbpg-regress/local_t/_20210114100329_3e9b07726306d88b3193dc95c10a5c5c_1741120517_seg2_0.csv
oss://adbpg-regress/local_t/_20210114100329_3e9b07726306d88b3193dc95c10a5c5c_manifest
Object Number is: 4
0.136180(s) elapsed
```

查看清单文件内容：

```
ossutil cat oss://adbpg-regress/local_t/_20210114100329_3e9b07726306d88b3193dc95c10a5c5c_manifest
```

返回信息如下：

```
{
   "entries": [
      {"url": "oss://adbpg-regress/local_t/_20210114100329_3e9b07726306d88b3193dc95c10a5c5c_162488956_seg1_0.csv"},
      {"url": "oss://adbpg-regress/local_t/_20210114100329_3e9b07726306d88b3193dc95c10a5c5c_163756258_seg0_0.csv"},
      {"url": "oss://adbpg-regress/local_t/_20210114100329_3e9b07726306d88b3193dc95c10a5c5c_1741120517_seg2_0.csv"}
   ]
}
```

2. 使用UNLOAD导出时，生成指定manifest文件（manifest路径可与数据文件路径不同）。

> **说明** ALLOWOVERWRITE为true时，会覆写已存在的manifest文件，但不会覆写数据文件，数据文件由客户按需自行删除。

```
UNLOAD ('select * from local_t') TO 'oss://adbpg-regress/local_t/'
ACCESS_KEY_ID '<id>'
SECRET_ACCESS_KEY '<key>'
FORMAT AS CSV
MANIFEST 'oss://adbpg-regress-2/unload_manifest/t_manifest' -- 表示UNLOAD导出时生成指定路径的导出清单
ALLOWOVERWRITE 'true' -- 覆写已存在的manifest文件
ENDPOINT '<endpoint>'
FDW 'oss_fdw';
NOTICE:  OSS output prefix: "local_t/_20210114100329_3e9b07726306d88b3193dc95c10a5c5c".
UNLOAD
```

查看导出的文件列表：

```
ossutil ls -s oss://adbpg-regress/local_t/
```

导出路径下只有数据文件，返回信息如下：

```
oss://adbpg-regress/local_t/_20210114100956_4d3395a9501f6e22da724a2b6df1b6d3_1736161168_seg0_0.csv
oss://adbpg-regress/local_t/_20210114100956_4d3395a9501f6e22da724a2b6df1b6d3_1925769064_seg2_0.csv
oss://adbpg-regress/local_t/_20210114100956_4d3395a9501f6e22da724a2b6df1b6d3_644328153_seg1_0.csv
Object Number is: 3
0.118540(s) elapsed
```

查看清单文件内容，清单文件位于另一个bucket下：

```
ossutil cat oss://adbpg-regress-2/unload_manifest/t_manifest
```

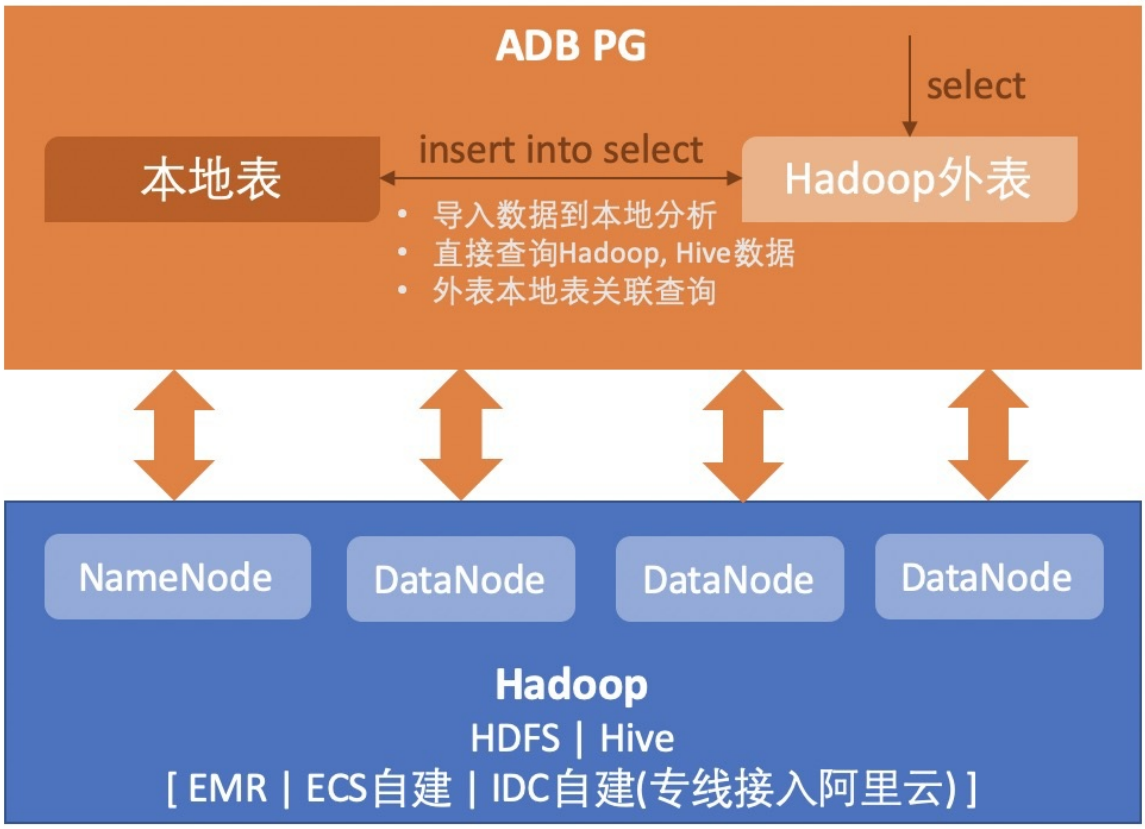
返回信息如下：

```
{
   "entries": [
      {"url": "oss://adbpg-regress/local_t/_20210114100956_4d3395a9501f6e22da724a2b6df1b6d3_1736161168_seg0_0.csv"},
      {"url": "oss://adbpg-regress/local_t/_20210114100956_4d3395a9501f6e22da724a2b6df1b6d3_1925769064_seg2_0.csv"},
      {"url": "oss://adbpg-regress/local_t/_20210114100956_4d3395a9501f6e22da724a2b6df1b6d3_644328153_seg1_0.csv"}
   ]
}
```

10.Hadoop生态外表联邦分析

云原生数据仓库 AnalyticDB PostgreSQL（简称 ADB PG）支持访问 Hadoop 生态的外部数据源。

> 说明
> - 本特性只支持存储弹性模式实例，且需要ADB PG实例和目标访问的外部数据源处于同一个VPC网络。
> - 2020年9月6日前申请的存量存储弹性模式实例，由于网络架构不同，无法与外部 Hadoop 生态的数据源网络打通，无法使用该特性。如需使用，请联系后台技术人员，重新申请实例，迁移数据。



前提条件：配置SERVER端

由于不同用户的配置需求不尽相同，如果您需要访问 Hadoop 生态的外部数据源进行联邦分析，请提交工单由ADB PG后台技术人员进行配置。以下为提交工单时需要提交的对应文件。

连接对象	提交工单内容

连接对象	提交工单内容
Hadoop(HDFS, HIVE, HBase)	core-site.xml 、hdfs-site.xml 、mapred-site.xml 、yarn-site.xml、hive-site.xml **说明** Kerberos认证时还需提供 keytab、krb5.conf等配置文件

基本语法

创建扩展

```
CREATE extension pxf;
```

创建外表

```
CREATE EXTERNAL TABLE <table_name>
        ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION('pxf://<path-to-data>?PROFILE[&<custom-option>=<value>[...]]&[SERVER=value]')
FORMAT '[TEXT|CSV|CUSTOM]' (<formatting-properties>);
```

创建 EXTERNAL TABLE 语法请参见CREATE EXTERNAL TABLE。

参数	说明
path-to-data	外表访问的目录，文件名，表名例如 *//data/pxf_examples/pxf_hdfs_simple.txt*
PROFILE [&<custom-option>=<value>[...]]	PXF访问外部数据的配置。 支持格式包括：Jdbc | hdfs:text | hdfs:text:multi | hdfs:avro | hdfs:json | hdfs:parquet | hdfs:AvroSequenceFile | hdfs:SequenceFile | HiveText | HiveRC | HiveORC | HiveVectorizedORC | HBase
FORMAT '[TEXT|CSV|CUSTOM]'	读取数据的格式。取值范围：TEXT、CSV、CUSTOM。

参数	说明

formatting-properties	与特定文件数据对应的格式化选项： formatter 或者 delimiter（分割符）
	• 与CUSTOM搭配
	  ◦ formatter='pxfwritable_import'
	  ◦ formatter='pxfwritable_export'
	• 与TEXT|CSV搭配
	  ◦ delimiter=E'\t'
	  ◦ delimiter ':'
	说明 escape 时需要加上 E。

SERVER	配置服务端文件的位置，该部分由后台技术人员操作后反馈给用户。 • postgresql • hdp3

访问HDFS数据

支持格式：

数据格式	PROFILE
text	hdfs:text
csv	hdfs:text:multi 、hdfs:text
Avro	hdfs:avro
JSON	hdfs:json
Parquet	hdfs:parquet
AvroSequenceFile	hdfs:AvroSequenceFile
SequenceFile	hdfs:SequenceFile

`FORMAT` 与 `formatting-properties` 请参见部分 。

示例 HDFS访问文件

HDFS构建测试数据文件 `pxf_hdfs_simple.txt` 。

```
echo 'Prague,Jan,101,4875.33
Rome,Mar,87,1557.39
Bangalore,May,317,8936.99
Beijing,Jul,411,11600.67' > /tmp/pxf_hdfs_simple.txt
# 构建目录
hdfs dfs -mkdir -p /data/pxf_examples
# 推文件至hdfs
hdfs dfs -put /tmp/pxf_hdfs_simple.txt /data/pxf_examples/
# 查看文件
hdfs dfs -cat /data/pxf_examples/pxf_hdfs_simple.txt
```

ADB PG实例创建外表并访问

```
postgres=# CREATE EXTERNAL TABLE pxf_hdfs_textsimple(location text, month text, num_ord
ers int, total_sales float8)
            LOCATION ('pxf://data/pxf_examples/pxf_hdfs_simple.txt?PROFILE=hdfs:text&SE
RVER=hdp3')
          FORMAT 'TEXT' (delimiter=E',');
postgres=# select * from pxf_hdfs_textsimple;
 location  | month | num_orders |    total_sales
-----------+-------+------------+--------------------
 Prague    | Jan   |        101 | 4875.3299999999999
 Rome      | Mar   |         87 | 1557.3900000000001
 Bangalore | May   |        317 | 8936.9899999999998
 Beijing   | Jul   |        411 |           11600.67
(4 rows)
```

LOCATION各字段含义说明：

- pxf:// ：pxf 协议，固定值。
- data/pxf_examples/pxf_hdfs_simple.txt：代表HDFS中的 *pxf_hdfs_simple.txt* 文件。
- PROFILE=hdfs:text：代表使用 *PROFILE=hdfs:text* 访问HDFS。
- SERVER=hdp3：后台技术人员会提供该选项。代表使用*PXF_SERVER/hdp3/*下的配置文件来支持PXF访问 HDFS。
- FORMAT 'TEXT' (delimiter=E',')：外部数据源格式配置项，代表以 `,` 分割符，格式为 `TEXT` 。

详细字段说明请参见基本语法。

示例：向HDFS写入 (TEXT,CSV)

前提条件：在HDFS上构建数据目录*/data/pxf_examples/pxfwritable_hdfs_textsimple1*。

```
hdfs dfs -mkdir -p /data/pxf_examples/pxfwritable_hdfs_textsimple1
```

> **说明** 在ADB PG执行insert的用户需要有HDFS该目录的写入权限。

1. ADB PG实例上创建可写外表。

```
CREATE WRITABLE EXTERNAL TABLE pxf_hdfs_writabletbl_1(location text, month text, num_orders int, total_sales float8)
            LOCATION ('pxf://data/pxf_examples/pxfwritable_hdfs_textsimple1?PROFILE=hdfs:text&SERVER=hdp3')
          FORMAT 'TEXT' (delimiter=',');
INSERT INTO pxf_hdfs_writabletbl_1 VALUES ( 'Frankfurt', 'Mar', 777, 3956.98 );
INSERT INTO pxf_hdfs_writabletbl_1 VALUES ( 'Cleveland', 'Oct', 3812, 96645.37 );
```

2. HDFS查看。

```
#查看文件
hdfs dfs -ls /data/pxf_examples/pxfwritable_hdfs_textsimple1
#查看数据
hdfs dfs -cat /data/pxf_examples/pxfwritable_hdfs_textsimple1/*
Frankfurt,Mar,777,3956.98
Cleveland,Oct,3812,96645.37
```

访问Hive数据

数据格式	PROFILE
TextFile	Hive, HiveText
SequenceFile	Hive
RCFile	Hive, HiveRC
ORC	Hive, HiveORC, HiveVectorizedORC
Parquet	Hive

FORMAT与 <formatting-properties>请参见部分。

示例 Hive

1. 产生数据。

```
echo 'Prague,Jan,101,4875.33
Rome,Mar,87,1557.39
Bangalore,May,317,8936.99
Beijing,Jul,411,11600.67
San Francisco,Sept,156,6846.34
Paris,Nov,159,7134.56
San Francisco,Jan,113,5397.89
Prague,Dec,333,9894.77
Bangalore,Jul,271,8320.55
Beijing,Dec,100,4248.41' > /tmp/pxf_hive_datafile.txt
```

2. Hive 创建table。

```
hive> CREATE TABLE sales_info (location string, month string,
        number_of_orders int, total_sales double)
        ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
        STORED AS textfile;
hive> LOAD DATA LOCAL INPATH '/tmp/pxf_hive_datafile.txt'
        INTO TABLE sales_info;
hive> SELECT * FROM sales_info;
```

3. ADB PG实例访问数据

```
postgres=# create extension pxf;
postgres=# CREATE EXTERNAL TABLE salesinfo_hiveprofile(location text, month text, n
um_orders int, total_sales float8)
LOCATION ('pxf://default.sales_info?PROFILE=Hive&SERVER=hdp3')
FORMAT 'custom' (formatter='pxfwritable_import');
postgres=# SELECT * FROM salesinfo_hiveprofile;
   location    | month | num_orders | total_sales
---------------+-------+------------+-------------
 Prague        | Jan   |        101 |     4875.33
 Rome          | Mar   |         87 |     1557.39
 Bangalore     | May   |        317 |     8936.99
 Beijing       | Jul   |        411 |    11600.67
 San Francisco | Sept  |        156 |     6846.34
 Paris         | Nov   |        159 |     7134.56
......
```

LOCATION各字段含义说明：

- pxf://：pxf 协议，固定值。
- default.sales_info：代表 Hive中default数据库下的 *sales_info* 表。
- PROFILE=Hive：代表使用*PROFILE=Hive* 访问 Hive。
- SERVER=hdp3：后台技术人员会提供该选项,代表使用*PXF_SERVER/hdp3/*下的配置文件来支持PXF访问 Hive。
- FORMAT 'custom' (formatter='pxfwritable_import')：外部数据源格式配置项，读取Hive中sales_info表时采用 *custom*，并与 *formatter='pxfwritable_import'* 搭配。

详细字段说明请参见基本语法。

示例 HiveText

```
postgres=# CREATE EXTERNAL TABLE salesinfo_hivetextprofile(location text, month text, n
um_orders int, total_sales float8)
             LOCATION ('pxf://default.sales_info?PROFILE=HiveText&SERVER=hdp3')
           FORMAT 'TEXT' (delimiter=E',');
postgres=# select * from salesinfo_hivetextprofile;
   location    | month | num_orders | total_sales
---------------+-------+------------+-------------
 Prague        | Jan   |        101 |     4875.33
 Rome          | Mar   |         87 |     1557.39
 Bangalore     | May   |        317 |     8936.99
 ......
```

示例 HiveRC

1. Hive创建table。

```
hive> CREATE TABLE sales_info_rcfile (location string, month string,
             number_of_orders int, total_sales double)
           ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
           STORED AS rcfile;
OK
##导入数据
hive> INSERT INTO TABLE sales_info_rcfile SELECT * FROM sales_info;
##查看
hive> SELECT * FROM sales_info_rcfile;
```

2. ADB PG 实例访问数据。

```
postgres=# CREATE EXTERNAL TABLE salesinfo_hivercprofile(location text, month text,
num_orders int, total_sales float8)
             LOCATION ('pxf://default.sales_info_rcfile?PROFILE=HiveRC&SERVER=hdp3'
)
           FORMAT 'TEXT' (delimiter=E',');
postgres=# SELECT location, total_sales FROM salesinfo_hivercprofile;
   location    | total_sales
---------------+-------------
 Prague        |     4875.33
 Rome          |     1557.39
 Bangalore     |     8936.99
 ......
```

示例 HiveORC

ORC可以有两种配置 HiveORC 和 HiveVectorizedORC 。

- 一次读取一行数据。
- 支持列投影。
- 支持复杂类型, 可以访问由数组、映射、结构和联合数据类型组成的Hive表。

1. Hive创建table。

```
hive> CREATE TABLE sales_info_ORC (location string, month string,
        number_of_orders int, total_sales double)
      STORED AS ORC;
hive> INSERT INTO TABLE sales_info_ORC SELECT * FROM sales_info;
hive> SELECT * FROM sales_info_ORC;
```

2. ADB PG 实例访问数据。

```
postgres=# CREATE EXTERNAL TABLE salesinfo_hiveORCprofile(location text, month text
, num_orders int, total_sales float8)
            LOCATION ('pxf://default.sales_info_ORC?PROFILE=HiveORC&SERVER=hdp3')
            FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
postgres=# SELECT * FROM salesinfo_hiveORCprofile;
......
 Prague        | Dec   |        333 |     9894.77
 Bangalore     | Jul   |        271 |     8320.55
 Beijing       | Dec   |        100 |     4248.41
(60 rows)
Time: 420.920 ms
```

示例 HiveVectorizedORC

- 一次最多读取1024行数据。
- 不支持列投影。
- 不支持复杂类型或时间戳数据类型。

ADB PG 实例访问数据

```
CREATE EXTERNAL TABLE salesinfo_hiveVectORC(location text, month text, num_orders int,
total_sales float8)
            LOCATION ('pxf://default.sales_info_ORC?PROFILE=HiveVectorizedORC&SERVER=h
dp3')
            FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
select * from salesinfo_hiveVectORC;
   location    | month | num_orders | total_sales
---------------+-------+------------+-------------
 Prague        | Jan   |        101 |     4875.33
 Rome          | Mar   |         87 |     1557.39
 Bangalore     | May   |        317 |     8936.99
 Beijing       | Jul   |        411 |    11600.67
 San Francisco | Sept  |        156 |     6846.34
 ......
```

示例 Parquet

1. Hive创建table。

```
hive> CREATE TABLE hive_parquet_table (location string, month string,
            number_of_orders int, total_sales double)
        STORED AS parquet;
INSERT INTO TABLE hive_parquet_table SELECT * FROM sales_info;
select * from hive_parquet_table;
```

2. ADB PG 实例访问数据。

```
postgres=# CREATE EXTERNAL TABLE pxf_parquet_table (location text, month text, numb
er_of_orders int, total_sales double precision)
    LOCATION ('pxf://default.hive_parquet_table?profile=Hive&SERVER=hdp3')
    FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
postgres=# SELECT month, number_of_orders FROM pxf_parquet_table;
 month | number_of_orders
-------+------------------
 Jan   |              101
 Mar   |               87
 May   |              317
 Jul   |              411
 Sept  |              156
 Nov   |              159
 ......
```

11.Database外表联邦分析

云原生数据仓库AnalyticDB PostgreSQL（简称ADB PG）支持通过JDBC的方式访问Oracle、PostgreSQL、MySQL外部数据源。

> **说明**
> - 本特性只支持存储弹性模式实例，且需要ADB PG实例和目标访问的外部数据源处于同一个VPC网络。
> - 2020年9月6日前申请的存量存储弹性模式实例，由于网络架构不同，无法与外部数据库网络打通，无法使用该特性。如需使用，请联系后台技术人员，重新申请实例，迁移数据。

前提条件：配置SEVER端

由于不同用户的需求配置不尽相同。如果您需要通过JDBC的方式访问外部数据源进行联邦分析，请提交工单由ADB PG后台技术人员进行配置。以下为提交工单时需要提交的对应文件。

连接对象	提交工单内容
SQL Database(PostgreSQL, Mysql, Oracle)	JDBC的连接串、目标访问外部数据源的用户名和密码。

使用Database外表联邦分析

创建扩展

```
CREATE extension pxf;
```

创建外表

```
CREATE EXTERNAL TABLE <table_name>
        ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION('pxf://<path-to-data>?PROFILE[&<custom-option>=<value>[...]]&[SERVER=value]')
FORMAT '[TEXT|CSV|CUSTOM]' (<formatting-properties>);
```

创建EXTERNAL TABLE语法请参见CREATE EXTERNAL TABLE。

参数	说明
path-to-data	外表访问表名，例如 *public.test_a*。
PROFILE [&<custom-option>=<value>[...]]	访问外部数据的配置。 使用JDBC方式访问外部数据库时使用PROFILE=Jdbc
FORMAT '[TEXT|CSV|CUSTOM]'	读取文件的格式。

参数	说明

formatting-properties	与特定文件数据对应的格式化选项：formatter或者delimiter（分割符）
	• 与CUSTOM搭配
	  ○ formatter='pxfwritable_import'
	  ○ formatter='pxfwritable_export'
	• 与TEXT|CSV搭配
	  ○ delimiter=E'\t'
	  ○ delimiter ':'
	说明 escape时需要加上E。

SERVER	配置服务端文件的位置，该部分由后台技术人员操作后反馈给用户。

示例：访问postgresql数据库

ADB PG后台技术人员配置完成后，您可以在ADB PG数据库中采用以下SQL语句创建外表并查询。

```
postgres=# CREATE EXTERNAL TABLE pxf_ext_pg(a int, b int)
  LOCATION ('pxf://public.t?PROFILE=Jdbc&SERVER=postgresql')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import')
ENCODING 'UTF8';
postgres=# select * from pxf_ext_pg;
   a   |   b
-------+-------
     1 |     2
     2 |     4
     3 |     6
     4 |     8
     5 |    10
     6 |    12
     7 |    14
--more--
```

LOCATION各字段含义说明：

- pxf://：pxf 协议，固定值。
- public.t：代表数据库public下的表名为t的表。
- PROFILE=Jdbc: 代表使用JDBC访问外部数据源
- SERVER=postgresql：ADB PG的后台技术人员将提供该选项。后台人员会根据您提交的工单要求进行相对应配置。此例中 SERVER=postgresql 代表使用*PXF_SERVER/postgresql/*下的配置文件来支持连接。
- FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import')：外部数据源格式配置项。查询外部数据源table时使用*CUSTOM*，并与*FORMATTER='pxfwritable_import*'搭配。

```
CREATE EXTERNAL TABLE pxf_ext_test_a( id int,name varchar)
  LOCATION ('pxf://public.test_a?PROFILE=Jdbc&server=postgresql')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import')
ENCODING 'UTF8';
```

支持的数据类型

- INTEGER, BIGINT, SMALLINT
- REAL, FLOAT8
- NUMERIC
- BOOLEAN
- VARCHAR, BPCHAR, TEXT
- DATE
- TIMESTAMP
- BYTEA

12.事务管理

AnalyticDB PostgreSQL版支持标准数据库事务ACID属性，提供了三种隔离级别。AnalyticDB PostgreSQL版为分布式MPP架构，支持节点水平扩展，同时保证节点间事务的强一致属性。本文介绍AnalyticDB PostgreSQL版的事务隔离级别及事务的相关操作。

隔离级别

AnalyticDB PostgreSQL版支持以下三种事务隔离级别，默认为读已提交（READ COMMITTED）。

- 读未提交（READ UNCOMMITTED）：SQL标准中的脏读（READ UNCOMMITED ）语法定义，但实际按读已提交（READ COMMITED）执行。
- 读已提交（READ COMMITTED）：SQL标准中的读已提交（READ COMMITTED）语法定义，按读已提交（READ COMMITTED）级别执行。
- 可序列化（SERIALIZABLE）：SQL标准中的可序列化（SERIALIZABLE）语法定义，但实际按可重复读（REPEATABLE READ）级别执行。

> **说明** AnalyticDB PostgreSQL版Serverless版本目前仅支持读已提交级别。

示例：

使用可序列化（SERIALIZABLE）隔离级别开始事务块：

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

事务管理

AnalyticDB PostgreSQL版提供了下列事务管理相关的SQL命令：

- BEGIN或者START TRANSACTION 开始一个事务块。
- END或者COMMIT提交一个事务的结果。
- ROLLBACK放弃一个事务而不做任何更改。
- SAVEPOINT在一个事务中标记一个位置并且允许做部分回滚。用户可以回滚在一个保存点之后执行的命令但保留该保存点之前执行的命令。
- ROLLBACK TO SAVEPOINT 回滚一个事务到一个保存点。
- RELEASE SAVEPOINT 销毁一个事务内的保存点。

示例：

在事务中建立一个保存点，后来撤销在它建立之后执行的所有命令的效果：

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

上面的事务将插入值1和3，但不会插入2。

要建立并且稍后销毁一个保存点：

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上面的事务将插入值3和4。

13.JSON & JSONB 数据类型操作

JSON 类型几乎已成为互联网及物联网（IoT）的基础数据类型，AnalyticDB PostgreSQL数据库对JSON数据类型做了完善的支持。并且AnalyticDB PostgreSQL 6.0版支持JSONB类型。

这部分介绍对JSON & JSONB数据类型的操作，包括：

- JSON & JSONB的异同
- JSON输入输出语法
- JSON操作符
- JSONB操作符
- JSON创建函数
- JSON处理函数
- JSONB创建索引

具体协议请参见 JSON 官网。

JSON & JSONB的异同

JSON和JSONB类型在使用上几乎完全一致，两者的区别主要在存储上，json数据类型直接存储输入文本的完全的拷贝，JSONB数据类型以二进制格式进行存储。同时JSONB相较于JSON更高效，处理速度提升非常大，且支持索引，一般情况下，AnalyticDB PostgreSQL 6.0版都建议使用JSONB类型替代JSON类型

JSON输入输出语法

AnalyticDB PostgreSQL支持的JSON数据类型的输入和输出语法详见RFC 7159。

一个JSON数值可以是一个简单值（数字、字符串、true/null/false），数组，对象。下列都是合法的JSON表达式：

```
-- 简单标量/简单值
-- 简单值可以是数字、带引号的字符串、true、false或者null
SELECT '5'::json;
-- 零个或者更多个元素的数组（元素类型可以不同）
SELECT '[1, 2, "foo", null]'::json;
-- 含有键/值对的对象
-- 注意对象的键必须总是带引号的字符串
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;
-- 数组和对象可以任意嵌套
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

以上的JSON类型都可以写成JSONB类型的表达式，例如：

```
-- 简单标量/简单值,转化为jsonb类型
SELECT '5'::jsonb;
```

JSON操作符

下表说明了可以用于JSON & JSONB数据类型的操作符。

操作符	右操作数类型	描述	例子	结果
->	int	获得JSON数组元素（索引从零开始）。	`'[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->2`	`{"c":"baz"}`
->	text	根据键获得JSON对象的域。	`'{"a": {"b":"foo"}}'::json->'a'`	`{"b":"foo"}`
->>	int	获得JSON数组元素的文本形式。	`'[1,2,3]'::json->>2`	`3`
->>	text	获得JSON对象域的文本形式。	`'{"a":1,"b":2}'::json->>'b'`	`2`
#>	text[]	获得在指定路径上的JSON对象。	`'{"a": {"b": {"c": "foo"}}}'::json#>'{a,b}'`	`{"c": "foo"}`
#>>	text[]	获得在指定路径上的JSON对象的文本形式。	`'{"a": [1,2,3], "b": [4,5,6]}'::json#>>'{a,2}'`	`3`

下表说明了可以用于JSONB数据类型的操作符。

JSONB操作符

下表说明了可以用于JSONB数据类型的操作符。

操作符	右操作数类型	描述	例子
=	jsonb	两个JSON对象的内容是否相等	`'[1,2]'::jsonb='[1,2]'::jsonb`
@>	jsonb	左边的JSON对象是否包含右边的JSON对象	`'{"a":1, "b":2}'::jsonb @>'{"b":2}'::jsonb`
<@	jsonb	左边的JSON对象是否包含于右边的JSON对象	`'{"b":2}'::jsonb <@'{"a":1, "b":2}'::jsonb`
?	text	指定的字符串是否存在于JSON对象中的key或者字符串类型的元素中	`'{"a":1, "b":2}'::jsonb ?'b'`
?|	text[]	右值字符串数组是否存在任一元素在JSON对象字符串类型的key或者元素中	`'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c']`

操作符	右操作数类型	描述	例子
?&	text[]	右值字符串数组是否所有元素在JSON对象字符串类型的key或者元素中	`'["a", "b"]'::jsonb ?& array['a', 'b']`

JSON创建函数

下表说明了用于创建JSON值的函数。

函数	描述	例子	结果
`to_json(anyelement)`	返回该值作为一个合法的JSON对象。数组和组合会被递归处理并且转换成数组和对象。如果输入包含一个从该类型到JSON的造型，会使用该cast函数来执行转换，否则将会产生一个JSON标量值。对于任何非数字、布尔值或空值的标量类型，会使用其文本表示，并且加上适当的引号和转义让它变成一个合法的JSON字符串。	`to_json('Fred said "Hi."'::text)`	`"Fred said \"Hi.\""`
`array_to_json(anyarray [, pretty_bool])`	返回该数组为一个JSON数组。一个多维数组会变成一个JSON数组的数组。 **说明** 如果pretty_bool为true，在第一维元素之间会增加换行。	`array_to_json('{{1,5},{99,100}}'::int[])`	`[[1,5],[99,100]]`
`row_to_json(record [, pretty_bool])`	返回该行为一个JSON对象。 **说明** 如果pretty_bool为true，在第一级别元素之间会增加换行。	`row_to_json(row(1,'foo'))`	`{"f1":1,"f2":"foo"}`

JSON处理函数

下表说明了处理JSON值的函数。

函数	返回类型	描述	例子	例子结果
`json_each(json)`	`set of key text, value json set of key text, value jsonb`	把最外层的JSON对象展开成键/值对的集合。	`select * from json_each('{"a":"foo", "b":"bar"}')`	`key | value -----+------ a | "foo" b | "bar"`

函数	返回类型	描述	例子	例子结果

`json_each_text(json)`	`set of key text, value text`	把最外层的JSON对象展开成键/值对的集合。返回值的类型是text。	`select * from json_each_text('{"a":"foo", "b":"bar"}')`	`key | value`
				`-----+-------`
				`a | foo`
				`b | bar`

`json_extract_path(from_json json, VARIADIC path_elems text[])`	`json`	返回path_elems指定的JSON值。等效于#>操作符。	`json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4')`	`{"f5":99,"f6":"foo"}`
`json_extract_path_text(from_json json, VARIADIC path_elems text[])`	`text`	返回path_elems指定的JSON值为文本。等效于#>>操作符。	`json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4', 'f6')`	`foo`

`json_object_keys(json)`	`setof text`	返回最外层JSON对象中的键集合。	`json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}')`	`json_object_keys`
				`------------`
				`f1`
				`f2`

`json_populate_record(base anyelement, from_json json)`	`anyelement`	把Expands the object in from_json中的对象展开成一行，其中的列匹配由base定义的记录类型。	`select * from json_populate_record(null::myrowtype, '{"a":1,"b":2}')`	`a | b`
				`---+---`
				`1 | 2`

`json_populate_recordset(base anyelement, from_json json)`	`set of anyelement`	将from_json中最外层的对象数组展开成一个行集合，其中的列匹配由base定义的记录类型。	`select * from json_populate_recordset(null::myrowtype, '[{"a":1,"b":2},{"a":3,"b":4}]')`	`a | b`
				`---+---`
				`1 | 2`
				`3 | 4`

函数	返回类型	描述	例子	例子结果

`json_array_elements(json)`	`set of json`	将一个JSON数组展开成JSON值的一个集合。	`select * from json_array_elements('[1,true, [2,false]]')`	`value`
				`-----------`
				`1`
				`true`
				`[2,false]`

JSONB创建索引

JSONB类型支持GIN, BTree索引。一般情况下，我们会在JSONB类型字段上建GIN索引，语法如下：

```
CREATE INDEX idx_name ON table_name USING gin (idx_col);
CREATE INDEX idx_name ON table_name USING gin (idx_col jsonb_path_ops);
```

> **说明** 在JSONB上创建GIN索引的方式有两种：使用默认的jsonb_ops操作符创建和使用jsonb_path_ops操作符创建。两者的区别在jsonb_ops的GIN索引中，JSONB数据中的每个key和value都是作为一个单独的索引项的，而jsonb_path_ops则只为每个value创建一个索引项。

JSON操作举例

创建表

```
create table tj(id serial, ary int[], obj json, num integer);
=> insert into tj(ary, obj, num) values('{1,5}'::int[], '{"obj":1}', 5);
INSERT 0 1
=> select row_to_json(q) from (select id, ary, obj, num from tj) as q;
                row_to_json
-------------------------------------------
 {"f1":1,"f2":[1,5],"f3":{"obj":1},"f4":5}
(1 row)
=> insert into tj(ary, obj, num) values('{2,5}'::int[], '{"obj":2}', 5);
INSERT 0 1
=> select row_to_json(q) from (select id, ary, obj, num from tj) as q;
                row_to_json
-------------------------------------------
 {"f1":1,"f2":[1,5],"f3":{"obj":1},"f4":5}
 {"f1":2,"f2":[2,5],"f3":{"obj":2},"f4":5}
(2 rows)
```

> **说明** JSON 类型不能支持作为分布键来使用；也不支持 JSON 聚合函数。

多表JOIN

```
create table tj2(id serial, ary int[], obj json, num integer);
=> insert into tj2(ary, obj, num) values('{2,5}'::int[], '{"obj":2}', 5);
INSERT 0 1
=> select * from tj, tj2 where tj.obj->>'obj' = tj2.obj->>'obj';
 id |  ary  |    obj    | num | id |  ary  |    obj    | num
----+-------+-----------+-----+----+-------+-----------+-----
  2 | {2,5} | {"obj":2} |   5 |  1 | {2,5} | {"obj":2} |   5
(1 row)
=> select * from tj, tj2 where json_object_field_text(tj.obj, 'obj') = json_object_field_te
xt(tj2.obj, 'obj');
 id |  ary  |    obj    | num | id |  ary  |    obj    | num
----+-------+-----------+-----+----+-------+-----------+-----
  2 | {2,5} | {"obj":2} |   5 |  1 | {2,5} | {"obj":2} |   5
(1 row)
```

JSON 函数索引

```
CREATE TEMP TABLE test_json (
       json_type text,
       obj json
);
=> insert into test_json values('aa', '{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}');
INSERT 0 1
=> insert into test_json values('cc', '{"f7":{"f3":1},"f8":{"f5":99,"f6":"foo"}}');
INSERT 0 1
=> select obj->'f2' from test_json where json_type = 'aa';
 ?column?
----------
 {"f3":1}
(1 row)
=> create index i on test_json (json_extract_path_text(obj, '{f4}'));
CREATE INDEX
=> select * from test_json where json_extract_path_text(obj, '{f4}') = '{"f5":99,"f6":"foo"
}';
 json_type |                 obj
-----------+--------------------------------------
 aa        | {"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}
(1 row)
```

JSONB创建索引

```
-- 创建测试表并生成数据
CREATE TABLE jtest1 (
    id int,
    jdoc json
);
CREATE OR REPLACE FUNCTION random_string(INTEGER)
RETURNS TEXT AS
$BODY$
SELECT array_to_string(
    ARRAY (
        SELECT substring(
            '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
```

```
            FROM (ceil(random()*62))::int FOR 1
        )
        FROM generate_series(1, $1)
    ),
    ''
)
$BODY$
LANGUAGE sql VOLATILE;
insert into jtest1 select t.seq, ('{"a":{"a1":"a1a1", "a2":"a2a2"},
"name":"'||random_string(10)||'","b":"bbbbb"}')::json from
generate_series(1, 10000000) as t(seq);
CREATE TABLE jtest2 (
    id int,
    jdoc jsonb
);
CREATE TABLE jtest3 (
    id int,
    jdoc jsonb
);
insert into jtest2 select id, jdoc::jsonb from jtest1;
insert into jtest3 select id, jdoc::jsonb from jtest1;
-- 建立索引
CREATE INDEX idx_jtest2 ON jtest2 USING gin(jdoc);
CREATE INDEX idx_jtest3 ON jtest3 USING gin(jdoc jsonb_path_ops);
-- 未建索引执行
EXPLAIN ANALYZE SELECT * FROM jtest1 where jdoc @> '{"name":"N9WP5txmVu"}';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
-------------------------------------------
 Gather Motion 2:1  (slice1; segments: 2)  (cost=0.00..162065.73 rows=10100 width=88) (actu
al time=1343.248..1777.605 rows=1 loops=1)
   ->  Seq Scan on jtest2  (cost=0.00..162065.73 rows=5050 width=88) (actual time=0.042..13
42.426 rows=1 loops=1)
         Filter: (jdoc @> '{"name": "N9WP5txmVu"}'::jsonb)
 Planning time: 0.172 ms
   (slice0)    Executor memory: 59K bytes.
   (slice1)    Executor memory: 91K bytes avg x 2 workers, 91K bytes max (seg0).
 Memory used:  2047000kB
 Optimizer: Postgres query optimizer
 Execution time: 1778.234 ms
(9 rows)
-- 使用jsonb_ops操作符创建索引执行
EXPLAIN ANALYZE SELECT * FROM jtest2 where jdoc @> '{"name":"N9WP5txmVu"}';
                                                              QUERY PLAN
-----------------------------------------------------------------------------------------------
-------------------------------------
 Gather Motion 2:1  (slice1; segments: 2)  (cost=88.27..13517.81 rows=10100 width=88) (actu
al time=0.655..0.659 rows=1 loops=1)
   ->  Bitmap Heap Scan on jtest2  (cost=88.27..13517.81 rows=5050 width=88) (actual time=0
.171..0.172 rows=1 loops=1)
         Recheck Cond: (jdoc @> '{"name": "N9WP5txmVu"}'::jsonb)
         ->  Bitmap Index Scan on idx_jtest2  (cost=0.00..85.75 rows=5050 width=0) (actual
time=0.217..0.217 rows=1 loops=1)
               Index Cond: (jdoc @> '{"name": "N9WP5txmVu"}'::jsonb)
```

```
 Planning time: 0.151 ms
   (slice0)    Executor memory: 69K bytes.
   (slice1)    Executor memory: 628K bytes avg x 2 workers, 632K bytes max (seg1).  Work_mem: 9K bytes max.
 Memory used:  2047000kB
 Optimizer: Postgres query optimizer
 Execution time: 1.266 ms
(11 rows)
-- 使用jsonb_path_ops操作符创建索引执行
EXPLAIN ANALYZE SELECT * FROM jtest3 where jdoc @> '{"name":"N9WP5txmVu"}';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Gather Motion 2:1  (slice1; segments: 2)  (cost=84.28..13513.81 rows=10101 width=88) (actual time=0.710..0.711 rows=1 loops=1)
   ->  Bitmap Heap Scan on jtest3  (cost=84.28..13513.81 rows=5051 width=88) (actual time=0.179..0.181 rows=1 loops=1)
         Recheck Cond: (jdoc @> '{"name": "N9WP5txmVu"}'::jsonb)
         ->  Bitmap Index Scan on idx_jtest3  (cost=0.00..81.75 rows=5051 width=0) (actual time=0.106..0.106 rows=1 loops=1)
               Index Cond: (jdoc @> '{"name": "N9WP5txmVu"}'::jsonb)
 Planning time: 0.144 ms
   (slice0)    Executor memory: 69K bytes.
   (slice1)    Executor memory: 305K bytes avg x 2 workers, 309K bytes max (seg1).  Work_mem: 9K bytes max.
 Memory used:  2047000kB
 Optimizer: Postgres query optimizer
 Execution time: 1.291 ms
(11 rows)
```

下面是Python访问的一个例子：

```
#! /bin/env python
import time
import json
import psycopg2
def gpquery(sql):
    conn = None
    try:
        conn = psycopg2.connect("dbname=sanity1x2")
        conn.autocommit = True
        cur = conn.cursor()
        cur.execute(sql)
        return cur.fetchall()
    except Exception as e:
        if conn:
            try:
                conn.close()
            except:
                pass
            time.sleep(10)
        print e
    return None
def main():
    sql = "select obj from tj;"
    #rows = Connection(host, port, user, pwd, dbname).query(sql)
    rows = gpquery(sql)
    for row in rows:
        print json.loads(row[0])
if __name__ == "__main__":
    main()
```

14.列存表使用排序键和粗糙集索引加速查询

本文介绍如何在列存表中使用排序键结合粗糙集索引，从而提高查询性能。

> **注意** 本文适用于：
> - 存储预留模式：数据库内核版本为20200826版本之后的新建实例。
> - 存储弹性模式：数据库内核版本为20200906版本之后的新建实例。

背景信息

当您创建表的时候，可以定义一个或者多个列为排序键（SORTKEY）。数据写入到表中之后，您可以对该表按照排序键进行排序重组。

表排序后可以加速范围限定查询，数据库会对每固定行记录每一列的min、max值。如果在查询时使用范围限定条件，ADBPG的查询引擎可以根据min、max值在对表进行扫描（SCAN）时快速跳过不满足限定条件的数据块（Block）。

例如，假设一张表存储了7年的数据，并且这张表的数据是按照时间字段排序存储的，如果我们需要查询一个月的数据，那么只需要扫描 1/(7*12) 的数据，也就是说有98.8%的数据块在扫描（SCAN）时可以被过滤掉。但是如果数据没有按照时间排序的话，可能所有的磁盘上的数据块都要被扫描到。

ADBPG支持两种排序方式：

- 组合排序：适用于限定条件是查询的前缀子集或者完全包含排序键，更适合于查询包含首列限定条件的情况。
- 多维排序：给每一个排序键分配相同的权重，更适合于查询条件包含任意限定条件子集的场景。

更多详情请参见组合排序和多维排序的性能对比。

性能对比

本节以组合排序给粗糙集索引带来的性能提升为例，展示粗糙集索引相比全表扫描的性能提升。

以TPCH Lineitem表为例，表中存储了7年的数据，我们比较数据未按照l_shipdate字段排序和用l_shipdate字段作为排序键并进行排序的限定条件查询的性能。

> **说明** 本文的TPC的实现基于官方TPC的基准测试，并不能与已发布的TPC基准测试结果相比较，本文中的测试并不符合TPC基准测试的所有要求。

测试步骤：

1. 创建一个32节点的实例。
2. 对Lineitem写入130亿行记录。
3. 查询1997-09-01到1997-09-30的数据。
   - 数据未按照l_shipdate排序。

```
adbpgadmin=# select count(*) from lineitem where l_shipdate between '1997-09-01' and '1997-09-30';
   count
-----------
 149619638
(1 row)

Time: 34651.793 ms
```

- 数据按照l_shipdate排序。

```
adbpgadmin=# select count(*) from lineitem where l_shipdate between '1997-09-01' and '1997-09-30';
   count
-----------
 149619638
(1 row)

Time: 1661.046 ms
```

创建表时定义排序键

样例

```
create table test(date text, time text, open float, high float, low float, volume int) with
(APPENDONLY=true,ORIENTATION=column) ORDER BY (volume);
```

语法

```
CREATE [[GLOBAL | LOCAL] {TEMPORARY | TEMP}] TABLE table_name (
[ { column_name data_type  ...} ]
)
[ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
[ ORDER BY (column, [ ... ] )]
```

数据库内核版本20210326之前，指定排序键语法为SORT KEY (column, [...])

对表进行排序

对数据进行组合排序

```
SORT [tablename]
```

数据库内核版本20210326之前可以使用以下语法的语句：

```
VACUUM SORT ONLY [tablename]
```

对数据进行多维排序

```
MULTISORT [tablename]
```

数据库内核版本20210326之前可以使用以下语法的语句：

```
VACUUM REINDEX [tablename]
```

当您对一张表执行过 `SORT` 或者 `MULTISORT` 之后，当前的数据会组织为按照排序键全表有序，但随着表中不断写入新数据，未排序的部分就会不断增加，这将有可能影响粗糙集过滤的性能。因此您需要周期性地执行 `SORT` 或者 `VACUUM REINDEX` MULTISORT操作来对表进行重排序，从而保证粗糙集过滤的性能。

修改排序键

您可以根据业务的变化修改已经创建的列存表的排序键，命令语法如下：

```
ALTER [[GLOBAL | LOCAL] {TEMPORARY | TEMP}] TABLE table_name SET ORDER BY (column, [ ... ]
)
```

这个命令只会修改catalog，不会对数据立即排序，需要重新执行 `SORT table_name` 命令排序。

样例

```
ALTER TABLE test SET ORDER BY(high,low);
```

数据库内核版本20210326之前可以使用以下语法的语句：

```
ALTER TABLE test SET SORTKEY(high,low);
```

如何选择排序键和排序方式

当您的查询SQL经常包含某一个列或者某几个列的等值或者范围限定条件查询时，比如时间列等，可以考虑使用这些列作为排序键，从而利用数据排序并结合粗糙索引，加速这类SQL的查询速度。

一般情况下建议使用组合排序，因为多维排序在排序过程中还需要做一些额外的数据组织工作，多维排序 `VACUUM REINDEX` 的时间会长于组合排序 `VACUUM SORT ONLY` 的时间。

如果您的查询SQL包含的限定条件经常不是总是包含某些列的，可以使用多维排序来加速查询。多维排序最多支持8列。

组合排序和多维排序的性能对比

我们会对同一张表分别做组合排序和多维排序，从而比较两种排序方式在不同的场景下，对不同查询的性能影响。

在这个场景中，我们创建一张表test，其包含4列（id, num1, num2, value）。使用（id,num1,num2）作为排序键。这张表一共包含一千万条记录。对于ADBPG来说并不算是一张特别大的表，但是其可以显示出组合排序和多维排序的性能差异，在更大的数据集中，两者的性能差异也会更明显。

测试步骤：

1. 创建测试表并设置表的排序键。
2. 写入测试数据。
3. 分别对这张表做组合排序和多维排序。
4. 对比同样的SQL场景，组合排序和多维排序的点查性能。
5. 对比同样的SQL场景，组合排序和多维排序的范围查询性能。

创建测试表并设置表的排序键

```
CREATE TABLE test(id int, num1 int, num2 int, value varchar)
with(APPENDONLY=TRUE, ORIENTATION=column)
DISTRIBUTED BY(id)
ORDER BY(id, num1, num2);
CREATE TABLE test_multi(id int, num1 int, num2 int, value varchar)
with(APPENDONLY=TRUE, ORIENTATION=column)
DISTRIBUTED BY(id)
ORDER BY(id, num1, num2);
```

写入一千万行数据

```
INSERT INTO test(id, num1, num2, value) select g,
(random()*10000000)::int,
(random()*10000000)::int,
(array['foo', 'bar', 'baz', 'quux', 'boy', 'girl', 'mouse', 'chlid', 'phone'])[floor(ra
ndom() * 10 +1)]
FROM generate_series(1, 10000000) as g;
INSERT INTO test_multi SELECT * FROM test;
adbpgadmin=# SELECT count(*) FROM test;
  count
----------
 10000000
(1 row)
adbpgadmin=# SELECT count(*) FROM test_multi;
  count
----------
 10000000
(1 row)
```

对两张表分别进行组合排序和多维排序

```
SORT test;
MULTISORT test_multi;
```

点查询比较性能

- 包含首列排序键限定条件。

  ```
  -- Q1 包含首列限定条件
  select * from test where id = 100000;
  select * from test_multi where id = 100000;
  ```

- 包含第二列限定条件。

  ```
  -- Q2 包含第二列限定条件
  select * from test where num1 = 8766963;
  select * from test_multi where num1 = 8766963;
  ```

- 包含二三列限定条件。

  ```
  -- Q3 包含二三列限定条件
  select * from test where num1 = 100000 and num2=2904114;
  select * from test_multi where num1 = 100000 and num2=2904114;
  ```

  性能对比结果

排序方式	Q1	Q2	Q3
组合排序	0.026s	3.95s	4.21s
多维排序	0.55s	0.42s	0.071s

范围查询比较性能

- 包含首列排序键限定条件。

```
-- Q1 包含首列限定条件
select count(*) from test where id>5000 and id < 100000;
select count(*) from test_multi where id>5000 and id < 100000;
```

- 包含第二列限定条件。

```
-- Q2 包含第二列限定条件
select count(*) from test where num1 >5000 and num1 <100000;
select count(*) from test_multi where num1 >5000 and num1 <100000;
```

- 包含二三列限定条件。

```
-- Q3 包含二三列限定条件
select count(*) from test where num1 >5000 and num1 <100000; and num2 < 100000;
select count(*) from test_multi where num1 >5000 and num1 <100000 and num2 < 100000;
```

性能对比结果

排序方式	Q1	Q2	Q3
组合排序	0.07s	3.35s	3.64s
多维排序	0.44s	0.28s	0.047s

结论

- 对于Q1场景，由于包含排序键的首列，所以组合排序的效果非常好，而多维排序则会相对性能弱一些。
- 对于Q2场景，由于不包含排序键的首列，组合排序基本上失效了，而多维排序依然能维持比较稳定的性能提升。
- 对于Q3场景，由于不包含排序键的首列，组合排序依然起不到很好的效果，并且由于比较条件的增加，需要额外的比较开销，时间更长，而多维排序表现出更好的性能，这是因为在查询时，限定条件包含的多维排序键越多，性能越好。

15.使用ANALYZE收集统计信息

正确的统计信息是查询优化器选择高效的查询计划的必要条件，如果统计信息不存在或者信息过时，优化器可能会生成低效的查询计划。使用ANALYZE语句可以更新统计信息。

ANALYZE使用命令如下：

```
ANALYZE [VERBOSE] [ROOTPARTITION [ALL] ] [table [ (column [, ...] ) ]]
```

AUTO ANALYZE

AUTO ANALYZE可以自动执行ANALYZE命令。AUTO ANALYZE将检查具有大量插入、更新或删除的表，并在需要的时候主动对表执行ANALYZE来收集更新表的统计信息。当前默认情况下，当表改动行数超过10%时，AUTO ANALYZE会自动对表触发一次ANALYZE操作。

对于MULTI MASTER实例，当前暂时只能追踪主MASTER上发生的改动行为，辅助MASTER发生的改动行为将不会触发AUTO ANALYZE。

> **说明** 云原生数据仓库AnalyticDB PostgreSQL版仅20210527及以后版本支持AUTO ANALYZE功能，如何升级小版本，请参见版本升级。

选择生成统计信息

不带参数运行ANALYZE会为数据库中所有的表更新统计信息，运行时间可能会很长。用户可以指定表名或者列名来指定收集单个表或者某些列的统计信息。当数据被改变时，应该有选择地ANALYZE表。

在大表上运行ANALYZE可能需要很长时间，用户可以只使用 `ANALYZE table(column, ...)` 为选择的列生成统计信息，例如收集用在连接、WHERE子句、SORT子句、GROUP BY子句或者HAVING子句中的那些列。对于一个分区表，用户可以在pg_partitions系统目录中寻找分区表的名字，只在更改过的分区上运行ANALYZE：

```
SELECT partitiontablename from pg_partitions WHERE tablename='parent_table';
```

何时运行ANALYZE

在以下时候运行ANALYZE：

- 导入数据之后。
- CREATE INDEX之后。
- 在做大量的INSERT、UPDATE以及DELETE操作之后。

ANALYZE会在表上要求一个读锁，因此它可以与其他数据库活动并行运行，但不要在执行导入、INSERT、UPDATE、DELETE以及CREATE INDEX操作期间运行ANALYZE。

16.使用EXPLAIN阅读查询计划

查询优化器使用数据库的数据统计信息来选择具有最小总代价的查询计划，查询代价通过磁盘I/O取得的磁盘页面数作为单位来度量。 可以使用EXPLAIN和EXPLAIN ANALYZE语句发现和改进查询计划。

EXPLAIN语法

EXPLAIN的语法如下：

```
EXPLAIN [ANALYZE] [VERBOSE] statement
```

EXPLAIN展示查询优化器对该查询计划估计的代价。例如：

```
EXPLAIN SELECT * FROM names WHERE id=22;
```

EXPLAIN ANALYZE不仅会显示查询计划，还会实际运行语句，显示实际执行的行数、时间等额外信息。例如：

```
EXPLAIN ANALYZE SELECT * FROM names WHERE id=22;
```

阅读EXPLAIN输出

查询计划类似于一棵有节点的树，执行和阅读的顺序是自底而上。计划中的每个节点表示一个操作，例如表扫描、表连接、聚集或者排序。阅读的顺序是从底向上：每个节点会把结果输出给直接在它上面的节点。一个计划中的底层节点通常是表扫描操作：顺序扫描表、通过索引或者位图索引扫描表等。如果该查询要求那些行上的连接、聚集、排序或者其他操作，就会有额外的节点在扫描节点上面负责执行这些操作。最顶层的计划节点通常是数据库的移动（MOTION）节点：重分布（REDISTRIBUTE)、广播（BROADCAST）或者收集（GATHER）节点。这些操作在查询处理时在实例节点之间移动数据。

EXPLAIN的输出对于查询计划中的每个节点都显示为一行并显示该节点类型和下面的执行的代价估计：

- cost：以磁盘页面获取为单位度量。1.0等于一次顺序磁盘页面读取。第一个估计是得到第一行的启动代价，第二个估计是得到所有行的总代价。
- rows：这个计划节点输出的总行数。这个数字根据条件的过滤因子会小于被该计划节点处理或者扫描的行数。最顶层节点的是估算的返回、更新或者删除的行数。
- width：这个计划节点输出的所有行的总字节数。

需要注意以下两点：

- 一个节点的代价包括其子节点的代价。最顶层计划节点有对于该计划估计的总执行代价。这是优化器估算出来的最小的数字。
- 代价只反映了在数据库中执行的时间，并没有计算在数据库执行之外的时间，例如将结果行传送到客户端花费的时间。

EXPLAIN示例

下面的例子描述了如何阅读一个查询的EXPLAIN查询代价：

```
EXPLAIN SELECT * FROM names WHERE name = 'Joelle';
                     QUERY PLAN
-------------------------------------------------------------
Gather Motion 4:1 (slice1) (cost=0.00..20.88 rows=1 width=13)
   -> Seq Scan on 'names' (cost=0.00..20.88 rows=1 width=13)
         Filter: name::text ~~ 'Joelle'::text
```

查询优化器会顺序扫描names表，对每一行检查WHERE语句中的filter条件，只输出满足该条件的行。 扫描操作的结果被传递给一个Gather Motion操作。Gather Motion是Segment把所有行发送给Master节点。在这个例子中，有4个Segment节点会并行执行，并向Master节点发送数据。这个计划估计的启动代价是00.00（没有代价）而总代价是20.88次磁盘页面获取。优化器估计这个查询将返回一行数据。

EXPLAIN ANALYZE

EXPLAIN ANALYZE除了显示执行计划还会运行语句。EXPLAIN ANALYZE计划会把实际执行代价和优化器的估计一起显示，同时显示额外的下列信息：

- 查询执行的总运行时间（以毫秒为单位）。
- 查询计划每个Slice使用的内存，以及为整个查询语句保留的内存。
- 计划节点操作中涉及的Segment节点数量，其中只会统计返回行的Segment。
- 操作产生最多行的Segment节点返回的行最大数量。如果多个Segment节点产生了相等的行数，EXPLAIN ANALYZE会显示那个用了最长结束时间的Segment节点。
- 为一个操作产生最多行的Segment节点的ID。
- 相关操作使用的内存量（work_mem）。如果work_mem不足以在内存中执行该操作，计划会显示溢出到磁盘的数据量最少的Segment的溢出数据量。例如：

  ```
  Work_mem used: 64K bytes avg, 64K bytes max (seg0).
  Work_mem wanted: 90K bytes avg, 90K byes max (seg0) to lessen
  workfile I/O affecting 2 workers.
  ```

- 产生最多行的Segment节点检索到第一行的时间（以毫秒为单位）以及该Segment节点检索到所有行花掉的时间。

下面的例子用同一个查询描述了如何阅读一个EXPLAIN ANALYZE查询计划。这个计划中粗体部分展示了每一个计划节点的实际计时和返回行，以及整个查询的内存和时间统计信息。

```
EXPLAIN ANALYZE SELECT * FROM names WHERE name = 'Joelle';
                     QUERY PLAN
-------------------------------------------------------------
Gather Motion 2:1 (slice1; segments: 2) (cost=0.00..20.88 rows=1 width=13)
Rows out: 1 rows at destination with 0.305 ms to first row, 0.537 ms to end, start offset b
y 0.289 ms.
        -> Seq Scan on names (cost=0.00..20.88 rows=1 width=13)
Rows out: Avg 1 rows x 2 workers. Max 1 rows (seg0) with 0.255 ms to first row, 0.486 ms to
end, start offset by 0.968 ms.
                 Filter: name = 'Joelle'::text
 Slice statistics:
      (slice0) Executor memory: 135K bytes.
    (slice1) Executor memory: 151K bytes avg x 2 workers, 151K bytes max (seg0).
Statement statistics: Memory used: 128000K bytes Total runtime: 22.548 ms
```

运行这个查询花掉的总时间是22.548毫秒。Sequential scan操作只有Segment（seg0）节点返回了1行，用

了0.255毫秒找到第一行且用了0.486毫秒来扫描所有的行。Segment向Master发送数据的Gather Motion接收到1行。这个操作的总消耗时间是0.537毫秒。

常见查询算子

表扫描算子

表扫描操作算子（SCAN）扫描表中的行以寻找一个行的集合，包括：

- Seq Scan： 顺序扫描表中的所有行。
- Append-only Scan： 扫描行存追加优化表。
- Append-only Columnar Scan： 扫描列存追加优化表中的行。
- Index Scan： 遍历一个B树索引以从表中取得行。
- Bitmap Append-only Row-oriented Scan： 从索引中收集仅追加表中行的指针并且按照磁盘上的位置进行排序。
- Dynamic Table Scan — 使用一个分区选择函数来选择分区。Function Scan节点包含分区选择函数的名称，可以是下列之一：
  - gp_partition_expansion： 选择表中的所有分区。
  - gp_partition_selection： 基于一个等值表达式选择一个分区。
  - gp_partition_inversion： 基于一个范围表达式选择分区。

  Function Scan节点将动态选择的分区列表传递给Result节点，该节点又会被传递给Sequence节点。

表连接

表连接操作算子（JOIN）包括：

- Hash Join： 从较小的表构建一个哈希表，用连接列作为哈希键扫描较大的表，为连接列计算哈希键并寻找具有相同哈希键的行。哈希连接通常是数据库中最快的连接。计划中的Hash Cond标识要被连接的列。
- Nested Loop Join： 选择在较大的表作为外表，迭代扫描较小的表中的行。Nested Loop Join要求广播其中的一个小表，这样一个表中的所有行才能与其他表中的所有行进行连接操作。Nested Loop Join在较小的表或者通过使用索引约束的表上执行得不错，但在使用Nested Loop连接大型表时可能会有性能影响。设置配置参数enable_nestloop为OFF（默认）能够让优化器更偏向选择Hash Join。
- Merge Join： 排序两个表并且将它们连接起来。Merge Join对于预排序好的数据很快。为了使查询优化器偏向选择Merge Join，可将参数enable_mergejoin设置为ON。

移动

移动操作算子（MOTION）在Segment节点之间移动数据，包括：

- Broadcast motion： 每一个Segment节点将自己的行发送给所有其他Segment节点，这样每一个Segment节点都有表的一份完整的本地拷贝。优化器通常只为小型表选择Broadcast motion。对大型表来说，Broadcast motion是会比较慢的。在连接操作没有按照连接键分布的情况下，可能会将把一个表中所需的行动态重分布到别的Segment节点上。
- Redistribute motion： 每一个Segment节点重新哈希数据并且把行发送到对应于哈希键的Segment节点上。
- Gather motion： 来自所有Segment的结果数据被合并在一起发送到节点上（通常是Master节点）。对大部分查询计划来说这是最后的操作。

其他

查询计划中出现的其他操作算子包括：

- Materialize： 优化器将一个子查询结果进行物化。
- InitPlan： 预查询，被用在动态分区消除中，当执行时还不知道优化器需要用来标识要扫描分区的值

时，会执行这个预查询。

- Sort： 为另操作（例如Aggregation或者Merge Join）进行所有数据排序。
- Group By： 通过一个或者更多列对行进行分组。
- Group/Hash Aggregate： 使用哈希对行进行聚集操作。
- Append： 串接数据集，例如在整合从分区表中各分区扫描的行时会用到。
- Filter： 使用来自于一个WHERE子句的条件选择行。
- Limit： 限制返回的行数。

查询优化器的选择

您可以通过查看EXPLAIN输出来判断计划是由ORCA还是传统查询优化器生成。这一信息出现在EXPLAIN输出的末尾。Settings行显示配置参数OPTIMIZER的设置。Optimizer status行显示该解释计划是由ORCA还是传统查询优化器生成。

使用GPORCA优化器的例子：

```
                       QUERY PLAN
------------------------------------------------------------------------------------
 Aggregate  (cost=0.00..296.14 rows=1 width=8)
   ->  Gather Motion 2:1  (slice1; segments: 2)  (cost=0.00..295.10 rows=1 width=8)
         ->  Aggregate  (cost=0.00..294.10 rows=1 width=8)
               ->  Table Scan on part  (cost=0.00..97.69 rows=100040 width=1)
 Settings:  optimizer=on
 Optimizer status: PQO version 1.609
(5 rows)
explain select count(*) from part;
```

使用传统优化器的例子：

```
                       QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=3519.05..3519.06 rows=1 width=8)
   ->  Gather Motion 2:1  (slice1; segments: 2)  (cost=3518.99..3519.03 rows=1 width=8)
         ->  Aggregate  (cost=3518.99..3519.00 rows=1 width=8)
               ->  Seq Scan on part  (cost=0.00..3018.79 rows=100040 width=1)
 Settings:  optimizer=off
 Optimizer status: legacy query optimizer
(5 rows)
```

17.使用Resource Queue（资源队列）进行负载管理

使用Resource Queue可以对云原生数据仓库PostgreSQL版的系统资源进行管理或隔离，本文将介绍如何在云原生数据仓库PostgreSQL版中创建和使用Resource Queue。

Resource Queue介绍

一个数据库实例的CPU资源和内存资源是有限的，这些资源影响着数据库的查询性能，当数据库负载达到一定程度，各个查询会竞争CPU资源和内存资源，造成整体的查询性能低下。对于那些延迟敏感的业务，是不可忍受的情况。

云原生数据仓库PostgreSQL版提供了系统资源负载管理工具——Resource Queue。您可以根据自身业务的情况，指定数据库可运行的并发查询数、每个查询可以使用的内存大小、以及可使用的CPU资源。这样可以保证执行查询时有预期的系统资源，从而得到符合预期的查询性能。

创建Resource Queue

您可以使用如下语法创建Resource Queue：

```
CREATE RESOURCE QUEUE name WITH (queue_attribute=value [, ... ])
其中queue_attribute为Resource Queue的属性，可以取如下值：
    ACTIVE_STATEMENTS=integer
        [ MAX_COST=float [COST_OVERCOMMIT={TRUE|FALSE}] ]
        [ MIN_COST=float ]
        [ PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX} ]
        [ MEMORY_LIMIT='memory_units' ]
    MAX_COST=float [ COST_OVERCOMMIT={TRUE|FALSE} ]
        [ ACTIVE_STATEMENTS=integer ]
        [ MIN_COST=float ]
        [ PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX} ]
        [ MEMORY_LIMIT='memory_units' ]
```

参数	描述
ACTIVE_STATEMENTS	用于指定Resource Queue在某个时间点最多的活跃查询（正在执行的查询）数量。
MEMORY_LIMIT	用于指定在单个计算节点（segment）上Resource Queue所有查询最多可以使用的内存。 • 单位可以为KB、MB、GB。 • 默认值为-1，表示没有限制。
MAX_COST	用于指定Resource Queue查询代价的最大值，默认值为-1，表示没有限制。 **说明** 这里的查询代价是指云原生数据仓库PostgreSQL版优化器估算出来的查询代价。

参数	描述
COST_OVERCOMMIT	该参数需要设置MAX_COST参数。 • 当COST_OVERCOMMIT为true，查询代价大于MAX_COST的查询可以在系统空闲的时候执行。 • 当COST_OVERCOMMIT为false，查询代价大于MAX_COST的查询将会被拒绝执行。
MIN_COST	用于指定Resource Queue最小的查询代价，当查询代价小于MIN_COST的查询，将不会排队等待而是会立即被执行。
PRIORITY	用于指定Resource Queue的优先级，优先级高的查询将会被分配更多的CPU资源用于执行。 • MIN • LOW • MEDIUM • HIGH • MAX 默认值为MEDIUM。

说明 创建Resource Queue时，`ACTIVE_STATEMENTS` 和 `MAX_COST` 两个属性中必须指定一个，否则创建无法成功。

```
postgres=>  CREATE RESOURCE QUEUE adhoc2 WITH (MEMORY_LIMIT='2000MB');
ERROR:  at least one threshold ("ACTIVE_STATEMENTS", "MAX_COST") must be specified
```

- 指定并发查询数量

  在创建Resource Queue时，使用如下语法指定Resource Queue里用户可以并发执行的查询数量：

  ```
  CREATE RESOURCE QUEUE adhoc WITH (ACTIVE_STATEMENTS=3);
  ```

  这里创建了一个名为 `adhoc` 的Resource Queue，这个Resource Queue的用户在指定时间点，最多执行3个查询操作。如果此时队列中执行的查询数量为3个，同时有新的查询进入，那么新的查询将会处于等待状态，直到前面有查询执行完毕。

- 指定使用的内存上限

  在创建Resource Queue时，使用如下语法指定Resource Queue里查询使用的内存上限：

  ```
  CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20, MEMORY_LIMIT='2000MB');
  ```

  这里创建了一个名为 `myqueue` 的Resource Queue，在这个Resource Queue中，所有查询最多能用2000MB的内存。执行查询时，会在计算节点（segment）占用MEMORY_LIMIT/ACTIVE_STATEMENTS的内存，对于 `myqueue` 来说每个查询会占用2000MB/20=100MB的内存。如果查询需要单独可用内存，可以使用系统参数statement_mem来设置，但是需要保证不超过Resource Queue设定的MEMORY_LIMIT和系统参数max_statement_mem。示例如下：

  ```
  SET statement_mem='1GB';
  SELECT * FROM test_adbpg WHERE col='adb4pg' ORDER BY id;
  RESET statement_mem;
  ```

- 设置队列优先级

  给不同的Resource Queue设置优先级可以控制Resource Queue中的查询对CPU资源的使用。例如，系统中有大规模并发查询时，高优先级Resource Queue中的查询会比低优先级Resource Queue中的查询使用更多的CPU资源，从而保证高优先级的查询有更充分的CPU资源来执行。

  Resource Queue优先级的设置可以在创建时设置，示例如下：

  ```
  CREATE RESOURCE QUEUE executive WITH (ACTIVE_STATEMENTS=3, PRIORITY=MAX);
  ```

  Resource Queue优先级也可以在创建完成之后进行修改，具体操作请参见修改Resource Queue的配置。

  > **说明** 优先级与ACTIVE_STATEMENTS/MEMORY_LIMIT的区别：
  > - ACTIVE_STATEMENTS/MEMORY_LIMIT会在查询执行之前判断其是否允许被执行。
  > - 优先级机制是在一个查询开始执行后，根据系统的运动状态以及其所在队列的优先级，动态地给这个对应的查询分配可用的CPU资源。
  >
  > 例如，云原生数据仓库PostgreSQL版在执行一些低优先级的查询，然后一个高优先级的查询进入并准备执行。那么云原生数据仓库PostgreSQL版会为这个查询分配更多的CPU资源，并减少低优先级查询的CPU资源。CPU资源分配的规则如下：
  > - 相同优先级的查询所能被分配到的CPU资源一致。
  > - 当系统中同时有高、中、低三个优先级别的查询的时候，高优先级的查询会分配得到系统90%的CPU资源，剩下的10%会留给中和低优先级的查询去分配，在这10%的系统CPU资源中，中优先级的查询会得到其中90%的CPU资源，低优先级查询则得到剩余的10%。

Resource Queue被创建完毕之后，可以使用 `gp_toolkit.gp_resqueue_status` 系统视图查看限制设置和当前资源队列的状态，如下所示：

```
postgres=> SELECT * from gp_toolkit.gp_resqueue_status WHERE
postgres->   rsqname='adhoc';
 queueid | rsqname | rsqcountlimit | rsqcountvalue | rsqcostlimit | rsqcostvalue | rsqmemor
ylimit | rsqmemoryvalue | rsqwaiters | rsqholders
---------+---------+---------------+---------------+--------------+--------------+---------
-------+----------------+------------+------------
   19283 | adhoc   |             3 |             0 |           -1 |            0 |
-1 |              0 |          0 |          0
(1 行记录)
```

Resource Queue的创建不能在事务块内进行，如下所示：

```
postgres=> begin;
BEGIN
postgres=> CREATE RESOURCE QUEUE test_q WITH (ACTIVE_STATEMENTS=3, PRIORITY=MAX);
ERROR:  CREATE RESOURCE QUEUE cannot run inside a transaction block
```

> **说明** 并非所有的SQL都会受到Resource Queue的限制：
>
> - 当resource_select_only为on，SELECT、 SELECT INTO、CREATE TABLE AS SELECT、DECLARE CURSOR会受到约束。
> - 当resource_select_only为off，INSERT、UPDATE、DELETE也会被资源队列管理起来。
> - 在云原生数据仓库PostgreSQL版中，resource_select_only默认为off。

分配用户到Resource Queue

创建Resource Queue完成后，需要将一个或多个用户分配给这个Resource Queue，完成分配后Resource Queue就会对队列中用户的查询进行资源管理。

> **说明**
>
> - 如果某个用户没有分配到Resource Queue，云原生数据仓库PostgreSQL版会将该用户分配给pg_default资源队列。
> - pg_default可以同时运行500个活跃的查询语句，没有cost limit的限制，优先级为MEDIUM。

```
postgres=> SELECT * from gp_toolkit.gp_resqueue_status WHERE rsqname='pg_default';
 queueid |  rsqname   | rsqcountlimit | rsqcountvalue | rsqcostlimit | rsqcostvalue | r
sqmemorylimit | rsqmemoryvalue | rsqwaiters | rsqholders
---------+------------+---------------+---------------+--------------+--------------+--
--------------+----------------+------------+------------
    6055 | pg_default |           500 |             1 |           -1 |          126 |
-1 |   2.096128e+09 |          0 |          1
(1 行记录)
```

将用户分配给指定Resource Queue，语法如下：

```
ALTER ROLE name RESOURCE QUEUE queue_name;
CREATE ROLE name WITH LOGIN RESOURCE QUEUE queue_name;
```

您可以在用户创建之后修改其所属的Resource Queue，也可以在用户创建之时为其指定Resource Queue。

> **说明** 在任意时刻一个用户只能归属于一个Resource Queue。

删除Resource Queue中的用户

如果您需要把某个role移除出指定的Resource Queue，可以使用如下指令：

```
ALTER ROLE role_name RESOURCE QUEUE none;
```

修改Resource Queue的配置

您可以使用如下语句对Resource Queue的资源配置进行修改：

- 修改活跃的查询数：

```
ALTER RESOURCE QUEUE adhoc WITH (ACTIVE_STATEMENTS=5);
```

- 修改Resource Queue内存和查询代价约束：

```
ALTER RESOURCE QUEUE adhoc WITH (MAX_COST=-1.0, MEMORY_LIMIT='2GB');
```

- 修改Resource Queue优先级：

```
ALTER RESOURCE QUEUE adhoc WITH (PRIORITY=LOW);
ALTER RESOURCE QUEUE reporting WITH (PRIORITY=HIGH);
```

删除Resource Queue

您可以使用如下语句删除Resource Queue：

```
DROP RESOURCE QUEUE name;
```

> **说明** 删除Resource Queue前请检查如下项目，否则会导致删除失败：
>
> - Resource Queue没有被分配的用户。
> - Resource Queue中没有任何查询处于waiting状态。

18.列存表MetaScan加速查询性能

AnalyticDB PostgreSQL版支持列存储格式，具有较高的数据压缩能力，以及查询性能，但是当针对有较高过滤率的查询条件时，依然要做整列数据读取，或者建B-Tree索引，但是索引也有其的问题：一是列存表的索引无压缩，数据膨胀比较严重；二是结果集大的时候，索引代价比顺序扫描高，索引失效等问题。为此AnalyticDB PostgreSQL版针对此问题开发了MetaScan功能，具有很好的过滤性能，并且占用的存储空间也基本可以忽略不计。

> 注意
> - 该特性目前仅支持新建的存储预留模式实例，且数据库内核版本为20200826及以后。
> - 如果您的实例是存储弹性模式实例，请暂时忽略这个特性。

开启MetaScan功能

MetaScan分为2部分：meta信息收集和MetaScan，由2个系统参数控制开启和关闭：

- RDS_ENABLE_CS_ENHANCEMENT

  控制MetaScan的meta信息收集功能，on表示开启meta信息收集，off表示关闭meta信息收集。系统默认为on，在列存表数据发生变化时，自动收集meta信息。该系统参数是实例级别的参数，如需要修改，可以提工单让技术支持人员修改。

- RDS_ENABLE_COLUMN_META_SCAN

  控制查询是否使用MetaScan，on表示使用MetaScan，off表示不使用MetaScan。系统默认为off。该系统参数为session级别，可以通过下面SQL查看、修改：

  - 查看MetaScan是否开启

    ```
    SHOW RDS_ENABLE_COLUMN_META_SCAN;
    ```

  - 关闭MetaScan

    ```
    SET RDS_ENABLE_COLUMN_META_SCAN = OFF;
    ```

  - 开启MetaScan

    ```
    SET RDS_ENABLE_COLUMN_META_SCAN = ON;
    ```

> 注意 RDS_ENABLE_CS_ENHANCEMENT关闭后，所有列存表都将停止收集meta，如果需要开启RDS_ENABLE_CS_ENHANCEMENT的话，则在开启后，要想使用MetaScan功能，必须重新收集表的meta信息，可以使用如下SQL:
>
> ```
> ALTER TABLE table_name SET WITH (REORGANIZE=TRUE);
> ```

如何查看query是否使用了MetaScan

可以使用EXPLAIN查看SELECT是否使用了MetaScan，如下图所示：

```
postgres=# EXPLAIN SELECT * from lineitem where l_shipdate = '2019-01-01 00:00:00';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..99175.19 rows=2370 width=129)
   -> Append-only Columnar Meta Scan on lineitem  (cost=0.00..99175.19 rows=790 width=129)
         Filter: l_shipdate = '2019-01-01'::date
 Optimizer status: legacy query optimizer
(4 rows)
```

Explain 输出中的“Append-only Columnar Meta Scan”节点即是MetaScan节点，如果EXPLAIN中有该节点，则表示查询时使用了MetaScan。

MetaScan支持的类型和操作符

目前版本，MetaScan支持如下数据类型：

- INT2/INT4/INT8
- FLOAT4/FLOAT8
- TIME/TIMETZ/TIMESTAMP/TIMESTAMPTZ
- VARCHAR/TEXT/BPCHAR
- CASH

支持如下操作符：

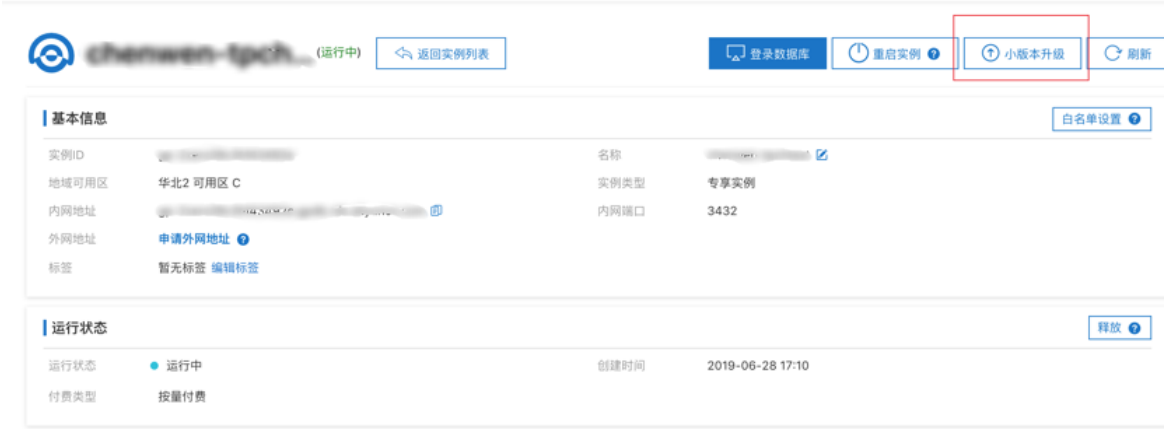
- =、<、<=、>、>=
- 逻辑运算符：and

已有实例升级

已有实例要使用MetaScan特性，需要做如下步骤升级AnalyticDB PostgreSQL版版本和表的meta信息：

1. 升级AnalyticDB PostgreSQL版版本

   从控制台上点击**小版本升级**：



2. 升级表的meta信息

   小版本升级完成后，需要升级表的meta信息到最新版。 如果RDS_ENABLE_CS_ENHANCEMENT是关闭状态，请提工单联系技术支持来升级meta信息，否则可以按如下步骤升级表的meta版本：

i. 使用管理员账号创建升级函数。

```
CREATE OR REPLACE FUNCTION UPGRADE_AOCS_TABLE_META(tname TEXT) RETURNS BOOL AS $$
DECLARE
    tcount INT := 0;
BEGIN
    -- CHECK TABLE NAME
    EXECUTE 'SELECT COUNT(1) FROM PG_AOCSMETA WHERE RELID = ''' || tname || '''::RE
GCLASS' INTO tcount;
    IF tcount IS NOT NULL THEN
        IF tcount > 1 THEN
            RAISE EXCEPTION 'found more than one table of name %', tname;
        ELSEIF tcount = 0 THEN
            RAISE EXCEPTION 'not found target table in pg_aocsmeta, table name:%',
tname;
        END IF;
    END IF;
    EXECUTE 'ALTER TABLE ' || tname || ' SET WITH(REORGANIZE=TRUE)';
    RETURN TRUE;
END;
$$  LANGUAGE PLPGSQL;
```

ii. 使用管理员账号或者表的owner升级需要的表，执行如下SQL:

```
SELECT UPGRADE_AOCS_TABLE_META(table_name);
```

iii. 检查表的meta version, SQL如下：

```
SELECT version = 4 AS is_latest_version  FROM pg_appendonly WHERE relid = 'test'::R
EGCLASS
```

使用SortKey提升MetaScan性能

SortKey是AnalyticDB PostgreSQL版的另一个特性，可以让表按指定的列排序。把MetaScan与SortKey结合使用，可以有效的提高MetaScan执行的性能。 列存表是通过block为单位来存储数据的，MetaScan是通过meta信息判断block是否满足条件，跳过不满足条件的block，从而减少I/O，提升扫描性能的。如果过滤列的数值分布的非常散，例如每个block都有，这样的话，即使过滤率很好也需要扫描所有的block，扫描性能低下。如果在表上以过滤列创建SortKey，则可以把列上相同的值集中到连续的block内，这样MetaScan就可以快速过滤掉不需要的block，从而提升扫描性能。

创建SortKey，请参见 列存表使用排序键和粗糙集索引加速查询。

MetaScan的限制

目前版本MetaScan与ORCA优化器不兼容，在开启ORCA优化器时，无法使用MetaScan。查看当前优化器的方式是：

```
SHOW OPTIMIZER;
```

on表示是ORCA优化器。优化器相关信息，请参见两种优化器的选择。

19.排序加速计算

本文为您介绍云原生数据仓库AnalyticDB PostgreSQL版如何通过底层的数据顺序加速查询速度。

当您执行 `SORT <tablename>` 后，系统会对表数据进行排序，当数据完成排序后，AnalyticDB PostgreSQL版即可利用数据的物理顺序，将SORT算子下推到存储层进行计算加速。如果您的SQL可以利用底层的数据顺序，则会从中获得加速收益，该特性可以基于SORT KEY加速SORT、AGG、JOIN算子。

> **说明**
> - 排序加速计算功能需要数据完全有序，当您写入数据后需要重新执行 `SORT <tablename>` 对数据进行排序。
> - 排序加速计算功能默认开启。

示例

以下示例将在测试表far中执行同样的查询语句，对比排序加速前与排序加速后查询时间的差距。

1. 创建测试表far，语句如下：

```
CREATE TABLE far(a int,  b int)
WITH (APPENDONLY=TRUE, COMPRESSTYPE=ZSTD, COMPRESSLEVEL=5)
DISTRIBUTED BY (a)  --分布键
ORDER BY (a);       --排序键
```

2. 写入1000000行数据，语句如下：

```
INSERT INTO far VALUES(generate_series(0, 1000000), 1);
```

3. 数据导入完成后，对数据进行排序，语句如下：

```
SORT far;
```

查询性能对比如下：

> **说明** 当前示例的查询时间仅供参考。查询时间受到数据量、计算资源、网络状况等多个因素影响，请以实际为准。

- ORDER BY加速
  - 排序加速前（未排序）

```
postgres=# select * from far order by a limit 1;
 a | b
---+---
 0 | 1
(1 row)

Time: 323.980 ms
```

- 排序加速后

```
postgres=# select * from far order by a limit 1;
 a | b
---+---
 0 | 1
(1 row)

Time: 6.971 ms
```

- GROUP BY加速
  - 排序加速前（未排序）

```
postgres=# select a,count(*) from far  group by a limit 1;
   a    | count
--------+-------
 579229 |     1
(1 row)

Time: 779.368 ms
```

  - 排序加速后

```
postgres=# select a, count(*) from far group by a limit 1;
 a | count
---+-------
 0 |     1
(1 row)

Time: 6.859 ms
```

- JOIN加速
  - 排序加速前（未排序）

```
postgres=# select * from far t1, far t2 where t1.a = t2.a limit 1;
 a | b | a | b
---+---+---+---
 2 | 1 | 2 | 1
(1 row)

Time: 289.075 ms
```

- 排序加速后

  > **说明** JOIN排序加速需要关闭ORCA功能，打开mergejoin功能，语句如下：

  ```
  SET enable_mergejoin TO on;
  SET optimizer TO off;
  ```

  ```
  postgres=# select * from far t1, far t2 where t1.a = t2.a limit 1;
   a | b | a | b
  ---+---+---+---
   2 | 1 | 2 | 1
  (1 row)

  Time: 12.315 ms
  ```

-	ORDER BY	GROUP BY	JOIN
加速前	323.980 ms	779.368 ms	289.075 ms
加速后	6.971 ms	6.859 ms	12.315 ms

20.自动增量排序

AnalyticDB PostgreSQL版提供了自动增量排序功能，本文为您介绍如何使用自动增量排序功能。

功能简介

自动增量排序（AutoMerge）是AnalyticDB PostgreSQL版在后台运行的数据自动排序进程。自动增量排序会定期巡检表的数据状态，对新增无序数据进行排序，并与已有有序数据进行增量归并。

自动增量排序功能默认全局开启，无需使用自动增量排序的表可以关闭该功能，具体操作方法，请参见开启或关闭表级别自动增量排序。

如果您需要全局关闭自动增量排序功能，请提交工单联系技术支持进行关闭。

> **说明** 自动增量排序有助于查询的性能提升，但是增量排序生成新的有序文件需要占用一定资源，尤其是I/O资源。

使用限制

- 自动增量排序功能仅支持设置了排序键的AO表（AO行存表和AOCS列存表）。您可以通过如下命令查看是否为AO表：

  ```
  SELECT reloptions FROM pg_class WHERE relname = 'table_name';
  ```

  返回示例如下，其中 `appendonly=true` 表示该表为AO表：

  ```
                               reloptions
  ------------------------------------------------------------------------
   {appendonly=true,orientation=column,compresstype=lz4,compresslevel=9}
  (1 row)
  ```

- 自动增量排序功能依赖Autovacuum功能，如需使用自动增量排序请确保Autovacuum功能已开启。关于Autovacuum功能，请参见维护定期回收空间任务。
- 自动增量排序功能需要AnalyticDB PostgreSQL版为V6.3.5.0及以上版本，如何查看和升级内核版本，请参见查看内核小版本和版本升级。

定义排序键

如果您希望表具备自动排序的能力，则需要在DDL语句中定义排序键。

- 新建表时添加排序键，示例如下：

  ```
  CREATE TABLE table_name(
  col_name type,
  ...
  )
  WITH(appendonly = true, orientation = row/column)
  DISTRIBUTED BY(distributed_keys)
  ORDER BY(sort_keys)
  ;
  ```

- 对已创建的表使用ALTER命令添加或修改排序键，示例如下：

  ```
  ALTER TABLE table_name SET ORDER BY(sort_keys);
  ```

开启或关闭表级别自动增量排序

AnalyticDB PostgreSQL版提供了表级别的参数设置，可以灵活开启或关闭自动增量排序功能。

- 开启表级别自动增量排序功能：

```
ALTER TABLE table_name SET (automerge = on);
```

- 关闭表级别自动增量排序功能：

```
ALTER TABLE table_name SET (automerge = off);
```

- 查询表的自动增量排序功能是否开启，示例SQL如下：

```
SELECT relname, reloptions FROM pg_class WHERE relname = 'table_name';
```

返回示例如下：

```
  relname   |                    reloptions
------------+-------------------------------------------------
 table_name | {appendonly=true,orientation=column,automerge=on}
```

`automerge=on` 表示已开启自动增量排序功能，如果没有返回该内容，则表示表级别自动增量排序功能与系统保持一致，此时您可以通过以下SQL进行确认：

```
SHOW automerge;
```

自动增量排序的性能收益测试

以下内容以过滤计算为例，展示自动增量排序对于查询性能带来的收益。

1. 确认AutoVacuum和AutoMerge功能已开启，命令如下：

```
SHOW autovacuum;
SHOW automerge;
```

返回值为on则表示已开启。

2. 关闭Laser引擎功能，命令如下：

```
SET laser.enable = off;
```

> **说明** 关闭Laser引擎功能仅用于对比自动增量排序前后的加速效果，实际使用时建议开启。更多关于Laser引擎功能介绍，请参见Laser计算引擎的使用。

3. 新建一张表，并导入随机数据。

```
CREATE TABLE test_automerge(a bigint, b bigint)
WITH(appendonly = true, orientation = column, compresstype = lz4, compresslevel = 9)
DISTRIBUTED BY(a)
ORDER BY(b);
```

4. 调整合入排序（Merge）数据的阈值，用于体现排序对性能的影响。

```
ALTER TABLE test_automerge set (automerge_unsorted_row_num_threshold = 10000000);
INSERT INTO test_automerge SELECT i, round(random() * 100) FROM generate_series(1, 9000
000) as i;
```

5. 执行过滤查询。

```
SELECT count(*) FROM test_automerge WHERE b = 5;
```

返回结果如下：

```
 count
-------
 89713
(1 row)
Time: 204.918 ms
```

6. 继续导入数据，触发自动增量排序。

```
INSERT INTO test_automerge SELECT i, round(random() * 100) FROM generate_series(1, 1000000) as i;
```

7. 等待增量数据合入排序完成后，再次进行查询。

```
SELECT count(*) FROM test_automerge WHERE b = 5;
```

返回结果如下：

```
 count
-------
 99683
(1 row)
Time: 15.289 ms
```

> **说明** 自动增量排序的任务进程可以在 `pg_stat_activity` 视图中查询。

通过以上示例可以看出，排序对过滤场景的加速较为显著。排序对于计算性能的提升不仅仅局限于过滤查询场景，在Aggregate、Join、OrderBy等场景中均有非常出色的表现，因此合理的选择排序键非常关键。如何选择排序键可以参考以下建议：

- 优先考虑过滤场景，选择选择率低的过滤列作为排序键。
- 若表的Aggregate、Join、OrderBy计算场景占比很高，则选择GroupKey、JoinKey、OrderKey作为排序键，并且此种场景最好保证分布键和排序键一致。

相关文档

排序加速计算

21.并行查询

云原生数据仓库AnalyticDB PostgreSQL版提供了单表查询的并行查询功能，本文为您介绍并行查询功能。

功能说明

如果您的实例Segment节点是4核及以上规格，单表查询将自动开启并行查询，提升多核并发能力、降低查询时间。系统会通过当前并发数、Segment配置和SQL信息，自动选择并行度，在检测到当前系统负载较大时会关闭并行查询。在低并发场景下，对大数据量单表聚合查询，能够减少约50%查询时间。

> **说明**
>
> - 请确保您的实例内核小版本为6.3.4.0及以上版本，如何升级小版本，请参见版本升级。
> - 并行查询适用于4核及以上规格的实例。

测试示例

用于测试的AnalyticDB PostgreSQL实例规格如下：

- Segment节点规格：4C32G
- Segment节点数量：4

在数据库中导入10 GB测试数据，并行加速前后的单表查询时间耗时如下：

- 并行加速前

```
postgres=# select
l_returnflag,
l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from
lineitem
where
l_shipdate <= date '1998-12-01' - interval '90' day --(3)
group by
l_returnflag,
l_linestatus
order by
l_returnflag,
l_linestatus;
 l_returnflag | l_linestatus |  sum_qty   |  sum_base_price  |   sum_disc_price   |      sum_charge      |       avg_qty       |     avg_price      |        avg_disc
  | count_order
--------------+--------------+------------+------------------+--------------------+----------------------+---------------------+--------------------+---------------------
--+-------------
 A            | F            | 377518399 |  566065727797.25 |  537759104278.0656 |  559276670892.116819 | 25.5009751030070973 | 38237.151008958546 | 0.0500065745402432046
3 |    14804077
 N            | F            |   9851614 |   14767438399.17 |   14028805792.2114 |   14590490998.366737 | 25.5224483028409474 | 38257.810660081140 | 0.0499733677376566718
0 |      385998
 N            | O            | 743124873 | 1114302286901.88 | 1058580922144.9638 | 1100937000170.591854 | 25.4980758706893147 | 38233.902923481810 | 0.0500008118211313060
3 |    29144351
 R            | F            | 377732830 |  566431054976.00 |  538110922664.7677 |  559634780885.086257 | 25.5083847896801383 | 38251.219273559761 | 0.0499967923140874204
5 |    14808183
(4 rows)

Time: 17456.066 ms
```

- 并行加速后

```
postgres=# select
l_returnflag,
l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from
lineitem
where
l_shipdate <= date '1998-12-01' - interval '90' day --(3)
group by
l_returnflag,
l_linestatus
order by
l_returnflag,
l_linestatus;
 l_returnflag | l_linestatus |  sum_qty  |  sum_base_price  |   sum_disc_price   |      sum_charge      |       avg_qty       |     avg_price      |        avg_disc
   | count_order
--------------+--------------+-----------+------------------+--------------------+----------------------+---------------------+--------------------+---------------------
--+-------------
 A            | F            | 377518399 |  566065727797.25 |  537759104278.0656 |  559276670892.116819 | 25.5009751030070973 | 38237.151008958546 | 0.0500065745402432046
3 |    14804077
 N            | F            |   9851614 |   14767438399.17 |   14028805792.2114 |   14590490998.366737 | 25.5224483028409474 | 38257.810660081140 | 0.0499733677376566718
0 |      385998
 N            | O            | 743124873 | 1114302286901.88 | 1058580922144.9638 | 1100937000170.591854 | 25.4980758706893147 | 38233.902923481810 | 0.0500008118211313060
3 |    29144351
 R            | F            | 377732830 |  566431054976.00 |  538110922664.7677 |  559634780885.086257 | 25.5083847896801383 | 38251.219273559761 | 0.0499967923140874204
5 |    14808183
(4 rows)

Time: 9407.291 ms
```

并行加速前	并行加速后
17456.066ms	9407.291ms

22.Query Cache

AnalyticDB PostgreSQL版V6.3.7.0版本引入了Query Cache特性，该特性可以缓存查询结果，对于重复的查询可以非常迅速地返回查询结果，因此对于读多写少，尤其是重复查询多的场景，可以有效提升数据库的查询性能。

注意事项

AnalyticDB PostgreSQL版内核小版本需要为V6.3.7.0及以上版本。查询和升级内核小版本，请参见查看内核小版本和版本升级。

目前AnalyticDB PostgreSQL版在使用Query Cache功能时存在以下限制：

- 事务隔离级别必须为读已提交（READ-COMMITTED）才支持Query Cache。
- GUC参数rds_uppercase_colname和gp_select_invisible必须设置为off才支持Query Cache。
- 查询中涉及的表必须都开启Query Cache，才能查询缓存的数据。
- libpq前端协议版本小于3.0不支持Query Cache。
- 如果事务块内的查询之前有过修改操作，查询结果无法存储在Query Cache中。
- 临时表、视图、物化视图、系统表、Unlogged Table、外表、Volatile/Immutable Function不支持Query Cache。
- 直接访问分区子表时不支持Query Cache。
- 开启多Master特性时，所有查询都不支持Query Cache。
- 结果集大小超过7.5 KB不支持Query Cache。
- 单个查询涉及的表超过32个不支持Query Cache。
- 拓展查询（Extend Query）中使用了游标（Cursor）时不支持Query Cache。

全局开启Query Cache

由于Query Cache只有在查询时间局部性比较高的情况下才能发挥作用，因此Query Cache功能默认关闭，如需全局开启Query Cache功能，请提交工单联系技术支持进行开启。

会话级别开启或关闭Query Cache

您可以使用rds_session_use_query_cache参数在会话级别开启或关闭Query Cache功能。

会话级别开启Query Cache功能：

```
SET rds_session_use_query_cache = on;
```

会话级别关闭Query Cache功能：

```
SET rds_session_use_query_cache = off;
```

表级开启或关闭Query Cache

您可以使用querycache_enabled参数在表级别开启或关闭Query Cache功能。

对于新建表，可以使用如下语句开启Query Cache：

```
CREATE TABLE table_name (c1 int, c2 int) WITH (querycache_enabled=on);
```

对于建表时未开启Query Cache的表，可以使用如下语句开启Query Cache：

```
ALTER TABLE table_name SET (querycache_enabled=on);
```

对于不再需要使用Query Cache的表，可以使用如下语句关闭Query Cache：

```
ALTER TABLE table_name SET (querycache_enabled=off);
```

调整Query Cache失效时间

为了保证Query Cache查询结果的正确性，当系统中有DDL、DML操作时，Query Cache中表的缓存结果会失效，以免查询到过期的结果。但是由于AnalyticDB PostgreSQL版使用了MVCC机制，而Query Cache只会存储最新的查询结果，导致某些情况（例如读写事务并发场景）会返回一个过期的结果。

Query Cache的缓存结果有效生命周期默认为10分钟。如果缓存结果的时长超过了有效生命周期，那么执行相同的查询时，系统会重新执行该查询而不是返回缓存结果。

为了避免返回过期结果，您可以提交工单联系技术支持修改Query Cache有效生命周期。

性能评估

以下内容将针对OLTP和OLAP两种负载场景对Query Cache的效果进行评估。

> **说明** 本文的TPC-H和TPC-DS的实现基于TPC-H和TPC-DS的基准测试，并不能与已发布的TPC-H和TPC-DS基准测试结果相比较，本文中的测试并不符合TPC-H和TPC-DS基准测试的所有要求。

OLTP

以下内容将使用带索引的点查进行评估，测试结果如下：

场景	不使用Query Cache	使用Query Cache
0%缓存命中率 包含1条点查句	1718 TPS	• 测试过程中无Cache替换：1399 TPS • 测试过程中有Cache替换：915 TPS
50%缓存命中率 包含2条点查句	807 TPS	• 测试过程中无Cache替换：1367 TPS • 测试过程中有Cache替换：877 TPS
100%缓存命中率 包含1条点查句	1718 TPS	11219 TPS

在OLTP负载场景，一条正常的查询时延在10ms左右。在查询100%缓存命中的情况下，性能约有6.5倍提升。即使Query Cache无任何缓存命中，相比较不使用Query Cache性能差距也较小。而且Query Cache引入的开销绝对时间相对固定，在产生Cache替换时，大部分情况下开销不超过20ms。

OLAP

以下内容将分别使用10 GB数据量的TPC-H和TPC-DS进行评估，测试结果如下：

场景	不使用Query Cache	使用Query Cache
10 GB TPC-H	1255秒	522秒
10 GB TPC-DS	2813秒	1956秒

在OLAP负载场景，TPC-H和TPC-DS查询性能均有较为明显的提升，其中TPC-H查询命中缓存时，Q1查询约有1000多倍的提升。由于TPC-H和TPC-DS部分查询结果过大，超过缓存查询的最大值（7.5 KB），导致这部分查询结果没有被缓存，因此以上测试结果整体性能提升不高。

23.Dynamic Join Filter

Dynamic Join Filter可以有效提升AnalyticDB PostgreSQL版Hash Join性能。本文介绍Dynamic Join Filter的使用说明。

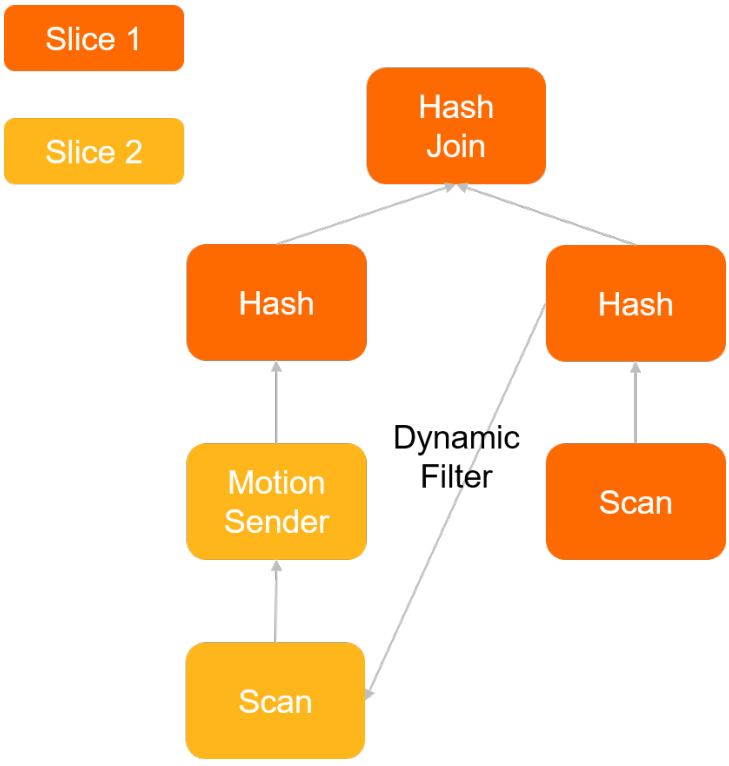
注意事项

- 内核版本为V6.3.8.0及以上版本。如何升级内核版本，请参见版本升级。
- 仅支持Legacy优化器，不支持ORCA优化器。关于优化器的详细信息，请参见查询性能优化指导。
- 仅支持Laser计算引擎。关于Laser计算引擎的详细信息，请参见Laser计算引擎的使用。
- 不支持执行计划中包含IntiPlan的SQL。
- 支持UDP网络模式，不支持TCP网络模式。

功能介绍

Hash Join是AnalyticDB PostgreSQL版中常用的算子，95%以上的关联查询是通过Hash Join实现的，由于Hash Join涉及到磁盘读写、网络交互和大量的计算，通常会成为查询中的瓶颈。

AnalyticDB PostgreSQL版于V6.3.8.0版本支持Dynamic Join Filter功能，通过动态收集Hash Join的右表Join键信息，在Join计算前过滤左表中无法Join匹配的数据，减少不必要的磁盘读取、网络和CPU计算开销，从而提升Hash Join的性能。Dynamic Join Filter原理示意图如下：



关闭或开启Dynamic Join Filter功能

Dynamic Join Filter功能默认为开启状态，您可以通过修改adbpg_enable_dynamic_join_filter参数关闭或开启Dynamic Join Filter功能，操作方法如下：

- 会话级别关闭Dynamic Join Filter功能：

```
SET adbpg_enable_dynamic_join_filter TO off;
```

- 会话级别开启Dynamic Join Filter功能：

```
SET adbpg_enable_dynamic_join_filter TO on;
```

上述方法仅支持会话级别关闭或开启Dynamic Join Filter功能，如需实例级别关闭或开启Dynamic Join Filter功能，请提交工单联系技术支持进行修改。

使用示例

以下示例以TPC-H测试集中的lineitem和part表作为测试数据表，使用TPC-H中的Q17语句为例对比开启或关闭Dynamic Join Filter功能后的查询耗时。

> **说明** 本文的TPC-H的实现基于TPC-H的基准测试，并不能与已发布的TPC-H基准测试结果相比较，本文中的测试并不符合TPC-H基准测试的所有要求。

1. 使用TPC-H在本地生成1 GB测试数据，操作方法，请参见生成测试数据。
2. 创建用于测试的lineitem和part表，建表语句如下：

```
CREATE TABLE LINEITEM (
    L_ORDERKEY    BIGINT NOT NULL,
    L_PARTKEY     INTEGER NOT NULL,
    L_SUPPKEY     INTEGER NOT NULL,
    L_LINENUMBER  INTEGER NOT NULL,
    L_QUANTITY    NUMERIC(15,2) NOT NULL,
    L_EXTENDEDPRICE  NUMERIC(15,2) NOT NULL,
    L_DISCOUNT    NUMERIC(15,2) NOT NULL,
    L_TAX         NUMERIC(15,2) NOT NULL,
    L_RETURNFLAG  CHAR(1) NOT NULL,
    L_LINESTATUS  CHAR(1) NOT NULL,
    L_SHIPDATE    DATE NOT NULL,
    L_COMMITDATE  DATE NOT NULL,
    L_RECEIPTDATE DATE NOT NULL,
    L_SHIPINSTRUCT CHAR(25) NOT NULL,
    L_SHIPMODE     CHAR(10) NOT NULL,
    L_COMMENT      VARCHAR(44) NOT NULL
) WITH (APPENDONLY=TRUE, ORIENTATION=COLUMN)
DISTRIBUTED BY (L_ORDERKEY, L_LINENUMBER);
CREATE TABLE PART (
    P_PARTKEY     INTEGER NOT NULL,
    P_NAME        VARCHAR(55) NOT NULL,
    P_MFGR        CHAR(25) NOT NULL,
    P_BRAND       CHAR(10) NOT NULL,
    P_TYPE        VARCHAR(25) NOT NULL,
    P_SIZE        INTEGER NOT NULL,
    P_CONTAINER   CHAR(10) NOT NULL,
    P_RETAILPRICE NUMERIC(15,2) NOT NULL,
    P_COMMENT     VARCHAR(23) NOT NULL
) WITH (APPENDONLY=TRUE, ORIENTATION=COLUMN)
DISTRIBUTED BY (P_PARTKEY);
```

3. 使用\COPY命令导入1 GB测试数据，语句如下：

```
\COPY LINEITEM FROM 'lineitem.tbl' with DELIMITER '|' NULL '';
\COPY PART FROM 'part.tbl' with DELIMITER '|' NULL '';
```

4. 查看关闭Dynamic Join Filter功能的情况下，Q17查询的执行耗时。具体步骤如下：
   i. 关闭Dynamic Join Filter功能：

```
SET adbpg_enable_dynamic_join_filter TO off;
```

   ii. 执行TPC-H Q17查询：

```
SELECT
    sum(l_extendedprice) / 7.0 as avg_yearly
FROM
    lineitem,
    part
WHERE
    p_partkey = l_partkey
    and p_brand = 'Brand#54'
    and p_container = 'SM CAN'
    and l_quantity < (
        SELECT
            0.2 * avg(l_quantity)
        FROM
            lineitem
        WHERE
            l_partkey = p_partkey
    );
```

   iii. 返回结果如下，查询耗时为3468ms：

```
     avg_yearly
--------------------
 336452.465714285714
(1 row)
Time: 3468.411 ms
```

5. 查看开启Dynamic Join Filter功能的情况下，Q17查询的执行耗时。具体步骤如下：
   i. 开启Dynamic Join Filter功能：

```
SET adbpg_enable_dynamic_join_filter TO on;
```

ii. 执行TPC-H Q17查询：

```
SELECT
    sum(l_extendedprice) / 7.0 as avg_yearly
FROM
    lineitem,
    part
WHERE
    p_partkey = l_partkey
    and p_brand = 'Brand#54'
    and p_container = 'SM CAN'
    and l_quantity < (
        SELECT
            0.2 * avg(l_quantity)
        FROM
            lineitem
        WHERE
            l_partkey = p_partkey
    );
```

iii. 返回结果如下，查询耗时为305ms：

```
     avg_yearly
---------------------
 336452.465714285714
(1 row)
Time: 305.632 ms
```

通过以上示例可以看出，开启Dynamic Join Filter功能后，执行时间从3468ms降低到305ms，有效地提升了执行性能。

24.Query Profiling Statistics

AnalyticDB PostgreSQL版提供了查询执行信息收集（Query Profiling Statistics）功能，开启该功能后，系统会自动收集并记录查询执行过程的统计信息，您可以通过系统视图浏览并检查执行较慢的SQL查询语句。

在AnalyticDB PostgreSQL版中，查询的执行过程被分解为多个算子并依次执行。在特定情况下，为了排查与分析查询执行的异常状态，需要查看查询执行过程的细节数据，例如某一特定算子执行过程消耗的时间，输入输出的行数，占用的资源（内存，I/O）等。AnalyticDB PostgreSQL版支持查询执行细节信息收集和记录，通过查询中各个算子的执行记录，您可以发现执行过程中存在的问题，进而排查并分析系统故障。

> **说明** AnalyticDB PostgreSQL版内核小版本需为V6.3.8.2及以上版本。升级版本具体操作，请参见版本升级。

开启查询执行信息收集功能

默认情况下，查询执行信息收集功能处于关闭状态，您可以通过queryprofile.enable参数开启或关闭该功能。

- 查看当前查询执行信息收集状态：

```
SHOW queryprofile.enable;
```

- 会话级别开启查询执行信息收集功能：

```
SET queryprofile.enable = ON;
```

- 会话级别关闭查询执行信息收集功能：

```
SET queryprofile.enable = OFF;
```

- 开启当前库的查询执行信息收集功能：

```
ALTER DATABASE <dbname> SET queryprofile.enable = ON;
```

如果需要实例级别开启或关闭Query Profiling Statistics功能，请提交工单联系技术支持进行修改。

查看查询执行信息

开启查询执行信息收集功能后，您可以通过查询执行信息提供的系统视图查看运行中或历史的查询和执行过程。

查询执行信息视图

- queryprofile.query_exec_history：用于查看历史查询的信息。
- queryprofile.query_exec_status：用于查看正在运行的查询的信息。
- queryprofile.node_exec_history：用于查看历史查询执行过程信息。
- queryprofile.node_exec_status：用于查看正在运行的查询执行过程信息。

queryprofile.query_exec_history和queryprofile.query_exec_status视图结构一致，视图中字段说明如下：

字段	类型	说明
queryid	int8	查询ID，即查询的唯一标识。

字段	类型	说明
sessid	integer	查询所属的会话ID。
commandid	integer	查询在其所属会话中的编号。
starttime	timestamptz	查询开始的时间。
runtime	float8	查询运行的总时间，单位为秒（s）。
stmt_text	text	查询对应的SQL文本。

queryprofile.node_exec_history和queryprofile.node_exec_status视图结构一致，视图中字段说明如下：

字段	类型	说明
queryid	int8	算子所属的查询ID，即查询的唯一标识。
stmtid	int8	算子所属查询ID所对应的SQL ID。
sessid	integer	算子所属的会话ID。
commandid	integer	算子所属的查询在其会话中的编号ID。
nodeid	integer	算子在查询执行计划中的ID。
sliceid	integer	算子在执行计划中所属的Slice ID。
nodetypeid	integer	算子类型ID。
nodename	text	算子类型。
tuplesout	int8	算子执行过程中输出的数据行数。
tuplesin	int8	算子执行过程中输入的数据行数。
tuplesplan	int8	算子在执行计划中的输入行数。
execmem	float8	执行器分配给算子的内存。
workmem	float8	算子工作占用内存。
starttime	timestamptz	算子开始执行的时间。
endtime	timestamptz	算子结束执行的时间。
duration	float8	算子执行消耗的时间，单位为秒（s）。 **说明** 该字段并非算子执行开始结束执行时间的间隔（可能包含下层算子执行消耗的时间），而是该算子本身执行所消耗的时间。

字段	类型	说明
diskreadsize	int8	算子从磁盘读取数据量。
diskreadtime	float8	算子从磁盘读取数据消耗的时间，单位为秒（s）。
netiosize	int8	数据在不同节点间网络传输的数据量。
netiotime	float8	数据在不同节点间网络传输消耗的时间，单位为秒（s）。

queryprofile.node_exec_history和queryprofile.node_exec_status视图中每一个查询算子对应一行数据，其中记录了查询执行过程中在该算子阶段的基本执行信息和消耗的资源，您可以根据视图中的信息定位到执行异常的阶段，优化查询执行过程。

查看正在运行的查询执行信息

示例1

查看正在运行的查询，SQL如下：

```
SELECT * FROM queryprofile.query_exec_status;
```

示例2

查看正在运行的查询执行过程，SQL如下：

```
SELECT * FROM queryprofile.node_exec_status;
```

查看历史的查询执行信息

示例1

查看历史查询，SQL如下：

```
SELECT * FROM queryprofile.query_exec_history;
```

示例2

查看历史查询执行过程，SQL如下：

```
SELECT * FROM queryprofile.node_exec_history;
```

修改运行中查询执行信息视图的更新频率

AnalyticDB PostgreSQL版提供了queryprofile.refresh_interval参数，用于控制queryprofile.query_exec_status和queryprofile.node_exec_status视图的更新频率，该参数值表示执行过程中执行算子每次更新统计数据所间隔的处理数据行数。

queryprofile.refresh_interval参数的默认值为1000，表示算子每读入1000行数据更新一次统计数据；0表示关闭动态收集查询执行信息功能。

您可以通过如下SQL查看查询执行信息更新频率：

```
SHOW queryprofile.refresh_interval;
```

返回示例如下：

```
 queryprofile.refresh_interval
-------------------------------
 1000
(1 row)
```

> **说明** 若需要关闭动态收集查询执行信息功能或修改数据更新频率，请提交工单联系技术支持进行修改。

历史查询执行信息回收

AnalyticDB PostgreSQL版提供了queryprofile.max_query_num和queryprofile.query_time_limit两个参数用于控制历史查询信息回收：

- queryprofile.max_query_num：用于控制保留的最大记录数。该参数值默认为10000，即保存最近10000条历史查询信息。您可以通过如下SQL查看保留的最大记录数：

  ```
  SHOW queryprofile.max_query_num;
  ```

- queryprofile.query_time_limit：用于清除记录中的短查询，总执行时长低于设置阈值的查询在查询信息回收时会被优先回收，单位为秒（s）。该参数值默认为1，即总执行时长小于1秒的查询会优先回收。您可以通过如下SQL查看保留最短查询的时长：

  ```
  SHOW queryprofile.query_time_limit;
  ```

例如，queryprofile.max_query_num的值为10000，queryprofile.query_time_limit的值为1时，表示查询信息数量超过10000时会触发查询信息回收，查询信息回收会优先回收总执行时长小于1s的查询信息，如果记录中已没有总执行时长小于1s的记录，系统将会从最早的查询信息开始回收。

> **说明** 若希望修改上述两个系统参数，请提交工单联系技术支持进行修改。

使用示例

查询执行信息可用于排查与分析查询执行的异常状态。当查询执行信息收集处于打开状态时，系统会收集所有查询的执行过程数据，您可通过执行过程数据获得如下信息：

- 系统中慢SQL或正在执行的查询。
- 查询执行信息中包含的各个算子信息及其输入或输出行数。
- 查询执行过程中耗时多的算子。
- 单一算子执行时占用的资源量，例如内存，磁盘 I/O，网络 I/O。

操作步骤如下：

1. 在当前会话中开启查询执行信息收集功能，SQL如下：

   ```
   SET queryprofile.enable = ON;
   ```

2. 查看历史查询执行信息，此处以获取最近10条历史查询执行信息为例，SQL示例如下：

   ```
   SELECT * FROM queryprofile.query_exec_history ORDER BY starttime DESC limit 10;
   ```

   返回示例如下：

```
postgres=# select * from queryprofile.query_exec_history order by starttime desc limit 10;
     queryid      | sessid | commandid |           starttime           | duration  |                          stmt_text
------------------+--------+-----------+-------------------------------+-----------+------------------------------------------------------------------------------------------------
 6902[redacted]93 |     51 |         2 | 2021-11-15 13:56:10.009499+08 |  7.081067 | select                                                                                         +
                  |        |           |                               |           |         o_orderpriority,                                                                       +
                  |        |           |                               |           |         count(*) as order_count                                                                +
                  |        |           |                               |           | from                                                                                           +
                  |        |           |                               |           |         tpch_row.orders                                                                        +
                  |        |           |                               |           | where                                                                                          +
                  |        |           |                               |           |         o_orderdate >= date '1993-07-01'                                                       +
                  |        |           |                               |           |         and o_orderdate < date '1993-07-01' + interval '3' month                               +
                  |        |           |                               |           |         and exists (                                                                           +
                  |        |           |                               |           |                 select                                                                         +
                  |        |           |                               |           |                         *                                                                      +
                  |        |           |                               |           |                 from                                                                           +
                  |        |           |                               |           |                         tpch_row.lineitem                                                      +
                  |        |           |                               |           |                 where                                                                          +
                  |        |           |                               |           |                         l_orderkey = o_orderkey                                                +
                  |        |           |                               |           |                         and l_commitdate < l_receiptdate                                       +
                  |        |           |                               |           |         )                                                                                      +
                  |        |           |                               |           | group by                                                                                       +
                  |        |           |                               |           |         o_orderpriority                                                                        +
                  |        |           |                               |           | order by                                                                                       +
                  |        |           |                               |           |         o_orderpriority;
 6902[redacted]15 |     50 |         2 | 2021-11-15 13:55:16.851294+08 |  2.030295 | UPDATE pgbench_accounts SET abalance = abalance + 2417 WHERE aid = 2413784;
 6902[redacted]53 |      4 |       130 | 2021-11-15 13:54:21.275643+08 |  0.010542 | select * from gp_dist_wait_status();
 6902[redacted]61 |     49 |         2 | 2021-11-15 13:53:52.169785+08 | 37.360402 | select                                                                                         +
                  |        |           |                               |           |         l_orderkey,                                                                            +
                  |        |           |                               |           |         sum(l_extendedprice * (1 - l_discount)) as revenue,                                    +
                  |        |           |                               |           |         o_orderdate,                                                                           +
                  |        |           |                               |           |         o_shippriority                                                                         +
                  |        |           |                               |           | from                                                                                           +
                  |        |           |                               |           |         tpch_row.customer,                                                                     +
                  |        |           |                               |           |         tpch_row.orders,                                                                       +
                  |        |           |                               |           |         tpch_row.lineitem                                                                      +
                  |        |           |                               |           | where                                                                                          +
                  |        |           |                               |           |         c_mktsegment = 'BUILDING'                                                              +
                  |        |           |                               |           |         and c_custkey = o_custkey                                                              +
                  |        |           |                               |           |         and l_orderkey = o_orderkey                                                            +
                  |        |           |                               |           |         and o_orderdate < date '1995-03-15'                                                    +
                  |        |           |                               |           |         and l_shipdate > date '1995-03-15'                                                     +
                  |        |           |                               |           | group by                                                                                       +
                  |        |           |                               |           |         l_orderkey,                                                                            +
                  |        |           |                               |           |         o_orderdate,                                                                           +
                  |        |           |                               |           |         o_shippriority                                                                         +
                  |        |           |                               |           | order by                                                                                       +
                  |        |           |                               |           |         revenue desc,                                                                          +
                  |        |           |                               |           |         o_orderdate;
 6902[redacted]27 |     48 |         2 | 2021-11-15 13:53:34.800577+08 |  5.130622 | UPDATE pgbench_accounts SET abalance = abalance + 2417 WHERE aid = 2413784;
 6902[redacted]96 |     20 |         6 | 2021-11-15 13:53:07.890833+08 |  0.005257 | select * from queryprofile.query_exec_history order by starttime desc limit 10;
 6902[redacted]65 |     20 |         4 | 2021-11-15 13:52:59.982712+08 |  0.004991 | select * from queryprofile.query_exec_history order by starttime desc limit 10;
 6902[redacted]63 |     20 |         2 | 2021-11-15 13:52:31.892249+08 |  0.085879 | INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (137, 86, 2837845, 3097, CURRENT_TIMESTAMP);
 6902[redacted]34 |     47 |       179 | 2021-11-15 13:52:27.419061+08 |  0.001344 | SELECT abalance FROM pgbench_accounts WHERE aid = 2884169 and bid = 17;
 6902[redacted]24 |     47 |       177 | 2021-11-15 13:52:27.416713+08 |  0.001163 | INSERT INTO pgbench_tellers values(677, 65, 0);
(10 rows)
```

3. 从以上返回结果中找到需要分析的查询执行信息，使用从queryprofile.query_exec_history视图中获取的queryid字段，在queryprofile.node_exec_history视图中获取该查询的执行过程，SQL示例如下：

```
SELECT * FROM queryprofile.node_exec_history WHERE queryid = 6902*********93;
```

返回示例如下：

```
postgres=# select * from queryprofile.node_exec_history where queryid = 6902[redacted]93;
     queryid      |      stmtid       | sessid | commandid | nodeid | sliceid | nodetypeid |       nodename        | tuplesout | tuplesin | tuplesplan | execmem | workmem
 |          starttime              |            endtime            | duration | diskreadsize | diskreadtime | netiosize
------------------+-------------------+--------+-----------+--------+---------+------------+-----------------------+-----------+----------+------------+---------+--------
-+-------------------------------+-------------------------------+----------+--------------+--------------+-----------
 690270970009093 | 690270970009093 |     51 |         2 |     10 |       1 |        134 | Hash Aggregate        |   2290502 |  6316985 |    1611885 | 65210K  | 98817K
 | 2021-11-15 13:56:10.026638+08 | 2021-11-15 13:56:16.969234+08 |    2.408 | 55296K       |        0.000 | 0K
 690270970009093 | 690270970009093 |     51 |         2 |      3 |       2 |        133 | Sort                  |         3 |        3 |          5 | 129K    | 65K
 | 2021-11-15 13:56:10.024229+08 | 2021-11-15 13:56:17.087943+08 |    0.003 | 0K           |        0.000 | 0K
 690270970009093 | 690270970009093 |     51 |         2 |      9 |       1 |        138 | Hash                  |         0 |        0 |    1611885 | 83969K  | 42534K
 | 2021-11-15 13:56:10.026638+08 | 2021-11-15 13:56:16.969272+08 |    0.000 | 0K           |        0.000 | 0K
 690270970009093 | 690270970009093 |     51 |         2 |      6 |       1 |        134 | Hash Aggregate        |         5 |    87606 |          5 | 0K      | 0K
 | 2021-11-15 13:56:10.026631+08 | 2021-11-15 13:56:16.969343+08 |    0.014 | 0K           |        0.000 | 0K
 690270970009093 | 690270970009093 |     51 |         2 |      2 |       2 |        134 | Group Aggregate       |         1 |        3 |          5 | 0K      | 0K
 | 2021-11-15 13:56:10.024223+08 | 2021-11-15 13:56:17.088002+08 |    0.000 | 0K           |        0.000 | 0K
 690270970009093 | 690270970009093 |     51 |         2 |      7 |       1 |        131 | Hash Join             |     87606 |  2386083 |    1611885 | 0K      | 0K
 | 2021-11-15 13:56:10.026636+08 | 2021-11-15 13:56:16.969308+08 |    0.981 | 24544K       |        0.000 | 0K
 690270970009093 | 690270970009093 |     51 |         2 |      4 |       2 |        142 | Redistribute Motion   |         3 |        3 |          5 | 0K      | 0K
 | 2021-11-15 13:56:10.02423+08  | 2021-11-15 13:56:17.087545+08 |    7.060 | 0K           |        0.000 | 1K
 690270970009093 | 690270970009093 |     51 |         2 |      8 |       1 |        111 | Seq Scan              |     95581 |  2497711 |    2310848 | 0K      | 0K
 | 2021-11-15 13:56:10.026636+08 | 2021-11-15 13:56:16.96899+08  |    0.879 | 354144K      |        0.000 | 0K
 690270970009093 | 690270970009093 |     51 |         2 |      1 |       0 |        142 | Gather Motion         |         5 |        5 |          5 | 0K      | 0K
 | 2021-11-15 13:56:10.009499+08 | 2021-11-15 13:56:17.090566+08 |    7.063 | 0K           |        0.000 | 1K
 690270970009093 | 690270970009093 |     51 |         2 |     11 |       1 |        111 | Seq Scan              |   6316985 |  9990397 |   11999982 | 0K      | 0K
 | 2021-11-15 13:56:10.02664+08  | 2021-11-15 13:56:16.969189+08 |    2.659 | 1524640K     |        0.000 | 0K
(10 rows)
```

从返回结果中可以看到算子名称、耗时、输入行数、输出行数、占用资源等信息。您可根据表中结果对每一个算子的执行记录进行分析，以定位和解决性能问题。