Cloud Data Warehouse for PostgreSQL Best Practices Product Documentation

Contents

Best Practices

Data Warehouse Table Development

Last updated：2024-02-02 15:36:51

Tables in CDWPG are similar to those in other relational databases. The difference is that the rows of a CDWPG table are distributed on different segments as determined by the distribution policy of the table.

Creating common table

The `CREATE TABLE` command is used to create a table. The following can be defined during table creation:

Table column and data type

Table constraint definition

Table distribution definition

Table storage format

Table partition definition

Use the `CREATE TABLE` command to create a table in the following format:

```
CREATE TABLE table_name (
[ { column_name data_type [ DEFAULT default_expr ]   -- Table column definition
   [column_constraint [ ... ]                        -- Column constraint definitio
]
   | table_constraint                                -- Table constraint definition
   ])
   [ WITH ( storage_parameter=value [, ... ] )       -- Table storage format defini
   [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]  -- Table distribut
   [ partition clause]                               -- Table partition definition
```

Example:

The table creation statement in the following example creates a table with **trans_id** as the distribution key and sets `RANGE` partitioning based on **date**.

```
CREATE TABLE sales (
  trans_id int,
  date date,
  amount decimal(9,2),
  region text)
  DISTRIBUTED BY (trans_id)
  PARTITION BY RANGE(date)
```

```
    (start (date '2018-01-01') inclusive
     end (date '2019-01-01') exclusive every (interval '1 month'),
     default partition outlying_dates);
```

Creating Temporary Table

A temporary table stores temporary intermediate results and is deleted automatically at the end of the session or selectively at the end of the current transaction. The command to create a temporary table is as follows:

```
CREATE TEMPORARY TABLE table_name(…)
    [ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP}]
```

Description: The temporary table behavior at the end of a transaction block can be controlled by `ON COMMIT` in the above statement.

PRESERVE ROWS: The data will be retained at the end of the transaction. This is the default behavior.

DELETE ROWS: All rows in the temporary table will be deleted at the end of each transaction block.

DROP: The temporary table will be dropped at the end of the current transaction block.

Example:

Create a temporary table and drop it at the end of the transaction.

```
CREATE TEMPORARY TABLE temp_foo (a int, b text) ON COMMIT DROP;
```

Table Constraint Definition

You can define constraints on columns and tables to restrict the data, but there are some limitations:

Columns referenced by a check constraint can only be in the same table.

Unique and primary key constraints must contain the distribution key column. They are not supported for append-optimized and column-oriented tables.
Foreign key constraints are allowed to be invalid in CDWPG.
The actual commands to use constraints are as follows:

```
UNIQUE ( column_name [, ... ] )
    | PRIMARY KEY ( column_name [, ... ] )
    | CHECK ( expression )
```

Check constraint

A check constraint specifies that the values in the column must satisfy a Boolean expression; for example:

```
CREATE TABLE products
            ( product_no integer,
              name text,
              price numeric CHECK (price > 0) );
```

Not-null constraint

A not-null constraint specifies that columns cannot have null values; for example:

```
CREATE TABLE products
        ( product_no integer NOT NULL,
          name text NOT NULL,
          price numeric );
```

Unique constraint

A unique constraint ensures that the data contained in a column or a group of columns is unique for all rows in a table. A table containing a unique constraint must be hash distributed, and the constraint column must contain the distribution key column; for example:

```
CREATE TABLE products
       ( product_no integer UNIQUE,
         name text,
         price numeric)
      DISTRIBUTED BY (product_no);
```

Note:

Primary key constraints are supported only for row-oriented heap tables but not append-only tables.

Primary key constraint

A primary key constraint is a combination of a unique constraint and a not-null constraint. A table containing a primary key constraint must be hash distributed, and the constraint column must contain the distribution key column. If the table has a primary key, this column (or group of columns) will be selected as the distribution key for the table by default; for example:

```
CREATE TABLE products
        ( product_no integer PRIMARY KEY,
          name text,
          price numeric)
      DISTRIBUTED BY (product_no);
```

Note:

Primary key constraints are supported only for row-oriented heap tables but not append-only tables.

Table Distribution Key Selection

Last updated：2024-02-02 15:36:51

This document describes how to select a distribution key in CDWPG.

Table Distribution Policy Selection

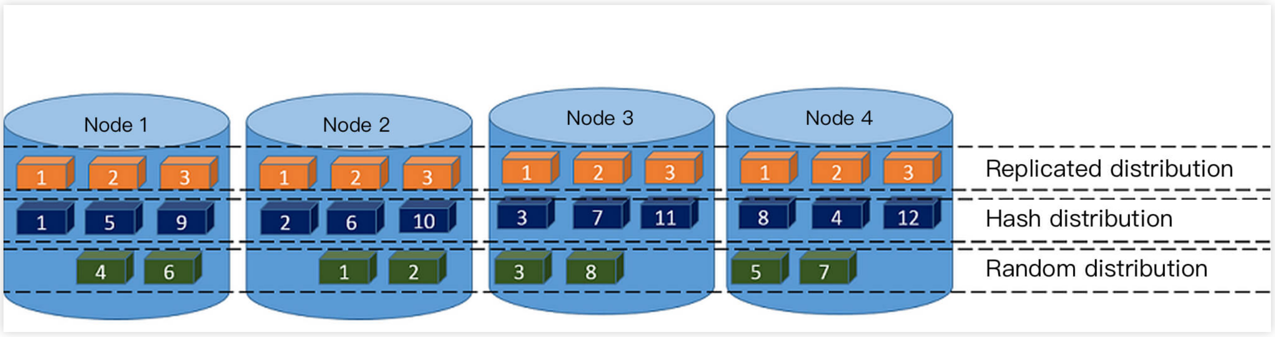
CDWPG supports three methods of data distribution among nodes: hash, random, and replicated.

```
CREATE TABLE <table_name> (...) [ DISTRIBUTED BY (<column>  [,..] ) | DISTRIBUTED R
```

The `CREATE TABLE` statement supports the following three distribution policy clauses:

`DISTRIBUTED BY (column, [ ... ])` specifies to distribute data rows among nodes (segments) according to the hash value of the distribution column. The same value will be always hashed to the same segment. Choosing a unique distribution key (such as the primary key) will ensure a more even data distribution. Hash distribution is the default distribution policy for tables, and if the `DISTRIBUTED` clause is not provided when the table is created, the primary key or the first eligible column of the table will be used as the distribution key. If there are no eligible columns in the table, the distribution policy will degrade to random distribution.

`DISTRIBUTED RANDOMLY` specifies to distribute data evenly among nodes (segments) in a circular manner. Unlike in the hash distribution policy, data rows with the same value are not necessarily located on the same segment. Although random distribution ensures an even data distribution, it is only recommended when the table doesn't have a suitable discretely distributed data column that can be used as the hash distribution column.
`DISTRIBUTED REPLICATED` specifies to distribute data in a replicated manner; that is, each node (segment) has all the data in the table. In this distribution policy, data is evenly distributed as each segment stores the same data rows. When large tables are joined with small tables, specifying a sufficiently small table as replicated may also improve the performance.



Below are examples:
The table creation statement in this example creates a hash-distributed table, where data is distributed to segments according to the hash value of the distribution key.

```
CREATE TABLE products (name varchar(40),
                       prod_id integer,
                       supplier_id integer)
                       DISTRIBUTED BY (prod_id);
```

The table creation statement in this example creates a randomly distributed table, where data is circularly placed into each segment. If the table doesn't have a suitable discretely distributed data column that can be used as the hash distribution column, the random distribution policy can be used.

```
CREATE TABLE random_stuff (things text,
                           doodads text,
                           etc text)
                           DISTRIBUTED RANDOMLY;
```

The table creation statement in this example creates a replicated distributed table, where each segment stores all the data of the table.

```
CREATE TABLE replicated_stuff (things text,
                           doodads text,
                           etc text)
                           DISTRIBUTED REPLICATED;
```

For simple queries by distribution key, including `UPDATE` and `DELETE` statements, CDWPG has the feature of pruning segments by distribution key. For example, if the `products` table uses `prod_id` as the distribution key, the following query will only be sent to segments that satisfy `prod_id=101` for execution, which greatly improves the SQL execution performance:

```
select * from products where prod_id = 101;
```

Table Distribution Key Selection

Reasonably planning the distribution key is critical to the performance of table queries. Pay attention to the following principles:

Don't use replicated tables, as they can easily lead to query degradation, which results in slower queries.

Select one or multiple columns with an even data distribution. If the values of the selected distribution column are not evenly distributed, data skew may occur, and some segments may store a lot of data (high query load), in which case, more time will be spent on such segments. Therefore, you should not select data of bool or datetime type as the distribution key.
Select a column that often requires joins as the distribution key. This can implement the **collocated join** calculation as shown in Figure 1; that is, when the join key and the distribution key are the same, the join can be completed inside the segment. Otherwise, the table needs to be redistributed (**redistribution motion**) to implement the **redistributed join** as shown in Figure 2, or some small tables can be broadcast (**broadcast motion**) to implement the **broadcast join** as shown in Figure 3. **The last two methods have a high network overhead.**
Select a query condition column that appears frequently as the distribution key, so that it is possible to prune segments by distribution key.
If no distribution key is specified, the table's primary key will be used as the distribution key by default, and if the table doesn't have a primary key, the first column will be used as the distribution key.
A distribution key can be defined as one or more columns; for example:

```
create table t1(c1 int, c2 int) distributed by (c1,c2);
```

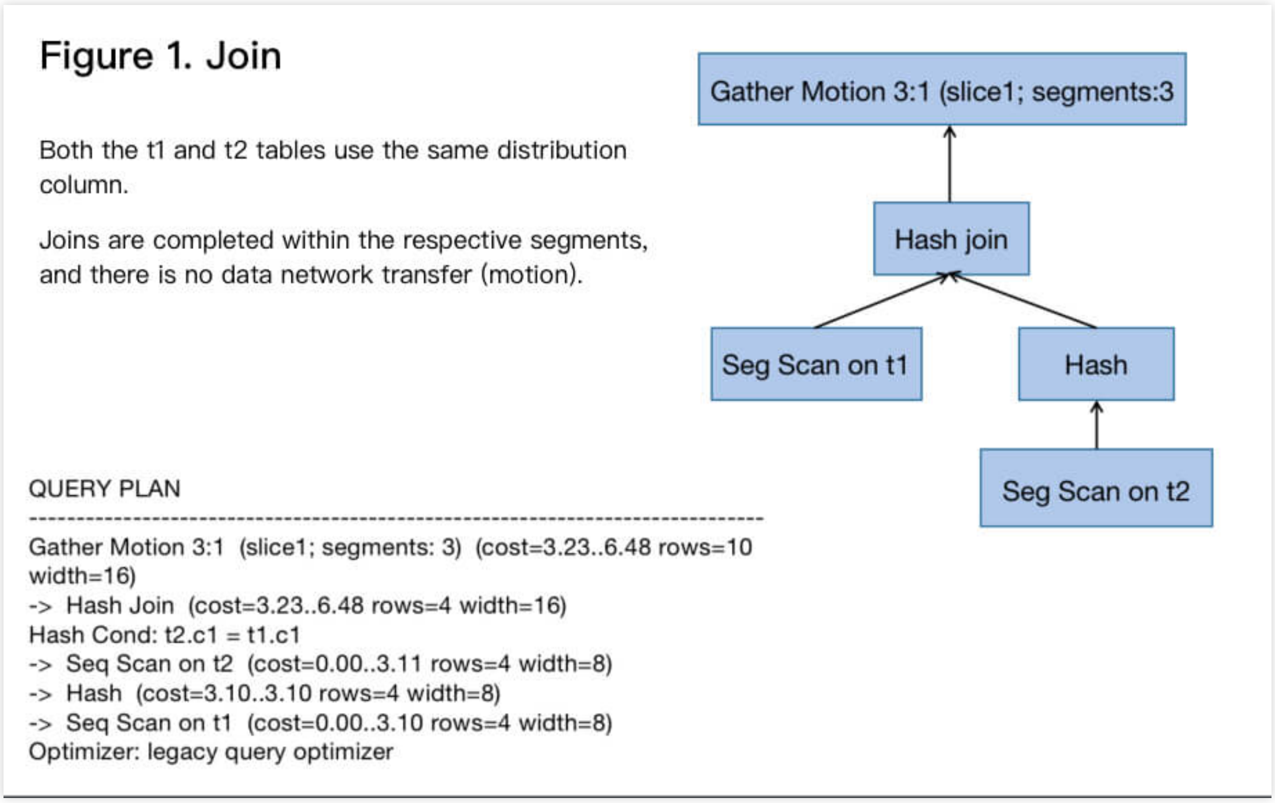


Figure 1. Join

Both the t1 and t2 tables use the same distribution column.

Joins are completed within the respective segments, and there is no data network transfer (motion).

```
QUERY PLAN
--------------------------------------------------------------------------------
Gather Motion 3:1  (slice1; segments: 3)  (cost=3.23..6.48 rows=10
width=16)
-> Hash Join  (cost=3.23..6.48 rows=4 width=16)
Hash Cond: t2.c1 = t1.c1
-> Seq Scan on t2  (cost=0.00..3.11 rows=4 width=8)
-> Hash  (cost=3.10..3.10 rows=4 width=8)
-> Seq Scan on t1  (cost=0.00..3.10 rows=4 width=8)
Optimizer: legacy query optimizer
```

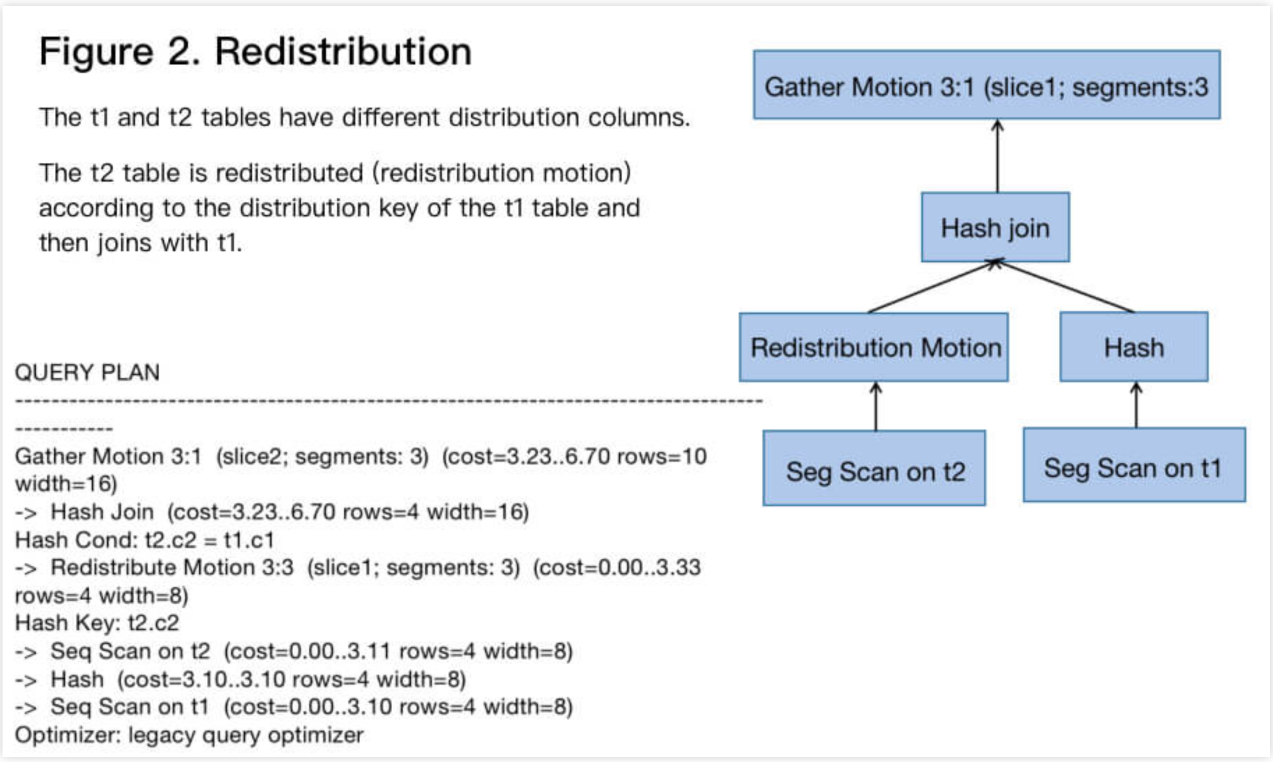


Figure 2. Redistribution

The t1 and t2 tables have different distribution columns.

The t2 table is redistributed (redistribution motion) according to the distribution key of the t1 table and then joins with t1.

```
QUERY PLAN
--------------------------------------------------------------------------------
-----------
Gather Motion 3:1  (slice2; segments: 3)  (cost=3.23..6.70 rows=10
width=16)
-> Hash Join  (cost=3.23..6.70 rows=4 width=16)
Hash Cond: t2.c2 = t1.c1
-> Redistribute Motion 3:3  (slice1; segments: 3)  (cost=0.00..3.33
rows=4 width=8)
Hash Key: t2.c2
-> Seq Scan on t2  (cost=0.00..3.11 rows=4 width=8)
-> Hash  (cost=3.10..3.10 rows=4 width=8)
-> Seq Scan on t1  (cost=0.00..3.10 rows=4 width=8)
Optimizer: legacy query optimizer
```

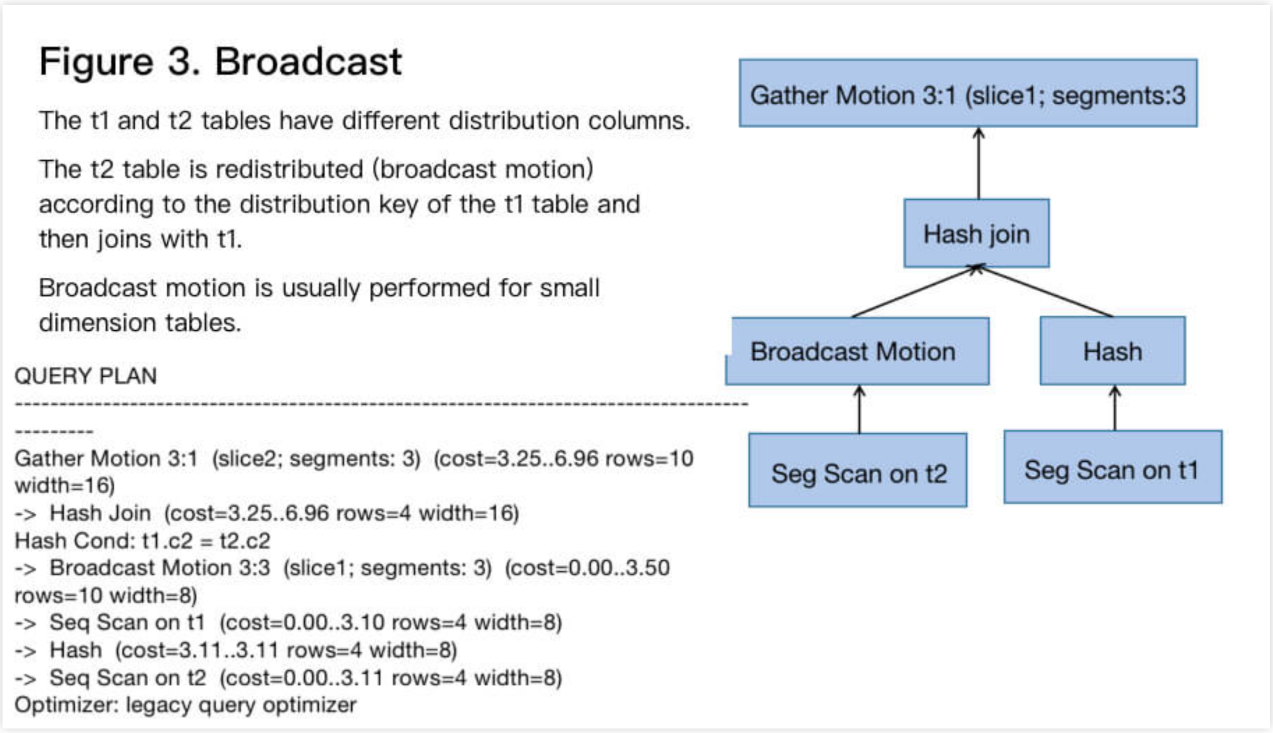


Figure 3. Broadcast

The t1 and t2 tables have different distribution columns.

The t2 table is redistributed (broadcast motion) according to the distribution key of the t1 table and then joins with t1.

Broadcast motion is usually performed for small dimension tables.

```
QUERY PLAN
-----------------------------------------------------------------------------------
---------
Gather Motion 3:1  (slice2; segments: 3)  (cost=3.25..6.96 rows=10
width=16)
-> Hash Join  (cost=3.25..6.96 rows=4 width=16)
Hash Cond: t1.c2 = t2.c2
-> Broadcast Motion 3:3  (slice1; segments: 3)  (cost=0.00..3.50
rows=10 width=8)
-> Seq Scan on t1  (cost=0.00..3.10 rows=4 width=8)
-> Hash  (cost=3.11..3.11 rows=4 width=8)
-> Seq Scan on t2  (cost=0.00..3.11 rows=4 width=8)
Optimizer: legacy query optimizer
```

Table Distribution Key Limits

Primary and unique keys must contain a distribution key; for example:

```
create table t1(c1 int, c2 int, primary key (c1)) distributed by (c2);
will fail to create.
```

Distribution Key Reasonableness Analysis

An inappropriate distribution key will cause data inconsistency in the table. You can run the following statement to check the data distribution:

```
create table t1(c1 int, c2 int) distributed by (c1);
select gp_segment_id,count(1) from  t1 group by 1 order by 2 desc;
 gp_segment_id | count
---------------+--------
             0 |     1000
             1 |     68
(2 rows)
```

When you find that the difference between segments is too large, you can modify the distribution key to make the data more even.

```
ALTER TABLE <table_name> SET WITH (REORGANIZE=true)
DISTRIBUTED BY (<distribution columns>);
```

Table Storage Format Selection

Last updated：2024-02-02 15:36:51

This document describes how to select a storage format in CDWPG.

Storage Format Overview

Greenplum (GP) stores data in heap or AO (AORO or AOCO) tables:
Heap table: It is inherited from PostgreSQL and is currently the default storage format of GP. It only supports row-oriented storage.
AO table: AO table was originally designed to only support `APPEND` (i.e., `INSERT`), so it was called append-only. It has been optimized since v4.3 and now supports `UPDATE` and `DELETE` , so it has been renamed append-optimized. AO supports both row-oriented (AORO) and column-oriented (AOCO) storage.

Heap Table

Heap table is inherited from PostgreSQL and uses MVCC for consistency. If you don't specify any storage format when creating a table, GP will use the heap table format.
A heap table supports partitioned table and row storage but not column storage or compression. It should be noted that when processing `UPDATE` and `DELETE` operations, the heap table does not actually delete data; instead, it relies on version information to block old data. Therefore, if your table has a large number of `UPDATE` or `DELETE` operations, the physical space used by the table will keep increasing. In this case, you need to use `VACUUM` to clear old data.
A heap tables doesn't support logical incremental backup, so if you want to take a snapshot of the heap table, you need to export the full data each time.
Table creation statement:

```
CREATE TABLE heap(
    a int,
    b varchar(32)
) DISTRIBUTED BY (a);
```

Best practices

For small tables such as dimension tables in the data warehouse or those containing fewer than one million data records, heap tables are recommended.

In OLTP scenarios where many `UPDATE` and `DELETE` operations exist and queries are mostly point queries with indexes, heap tables are recommended.

AO Table

AO table is designed to be used as large fact table in the data warehouse. It supports row storage (not recommended), column storage, and data compression.
An AO table is very different from a heap table in both the logical and physical table structures. For example, the heap table mentioned above uses MVCC to control the visibility of data after `UPDATE` and `DELETE` operations, while the AO table uses an additional bitmap table to indicate what data is visible in the AO table.
For an AO table with a large number of `UPDATE` and `DELETE` operations, you also need to use `VACUUM` for maintenance. However, in the AO table, `VACUUM` needs to reset the bitmap and compress the physical file, so it is usually slower than in a heap table.

AOCO

An AOCO table organizes data in columns and supports column-level compression.
The table creation statement is as follows, with the partitioning feature added:

```
CREATE TABLE aoco(
    a int  ENCODING (compresstype=zlib, compresslevel=5),
    b int  ENCODING (compresstype=none),
    c varchar(32) ENCODING (compresstype=RLE_TYPE, blocksize=32768),
    d varchar(32),
    fdate date
)
WITH (appendonly=true, orientation=column, compresstype=zlib, compresslevel=6, bloc
DISTRIBUTED BY (a)
PARTITION BY RANGE(fdate)
(
    PARTITION pn START ('2018-11-01'::date) END ('2018-11-10'::date) EVERY ('1 day'
    DEFAULT PARTITION pdefault
);
```

Compression

Compression is mainly used for column-oriented tables or append-write (`appendonly=true`) row-oriented tables.

The following two types of compression are available:

Table-level compression.

Column-level compression, where you can apply different compression algorithms to different columns.

Currently, CDWPG supports zstd, zlib, and rle_type compression algorithms.

Examples:

Create a column-oriented table using level-5 zlib compression:

```
CREATE TABLE foo (a int, b text)
   WITH (appendonly=true, orientation=column, compresstype=zlib, compresslevel=5);
```

Create a column-oriented table using level-5 zstd compression:

```
CREATE TABLE foo (a int, b text)
   WITH (appendonly=true, orientation=column, compresstype=zstd, compresslevel=5);
```

Best practices

AOCO is typically used for fact tables in the data warehouse. Such tables have many fields and large data volumes and are mainly used in OLAP scenarios, where only some fields in the tables are read and aggregated when queried, with no `SELECT \\* FROM` involved.

As AOCO is generally used for large tables, compression and partitioning are often used together to reduce the actual storage capacity and improve the performance.
In general, you can select the zlib compression algorithm at level 4 or 5; however, be sure to use the rle_type algorithm for fields with many repeated values.
Do not make the blocksize too large, especially for partitioned tables. GP maintains a buffer for each field in each partition, so if the blocksize is too large, the memory usage will be very high. The default value of 32768 is suitable in most cases.

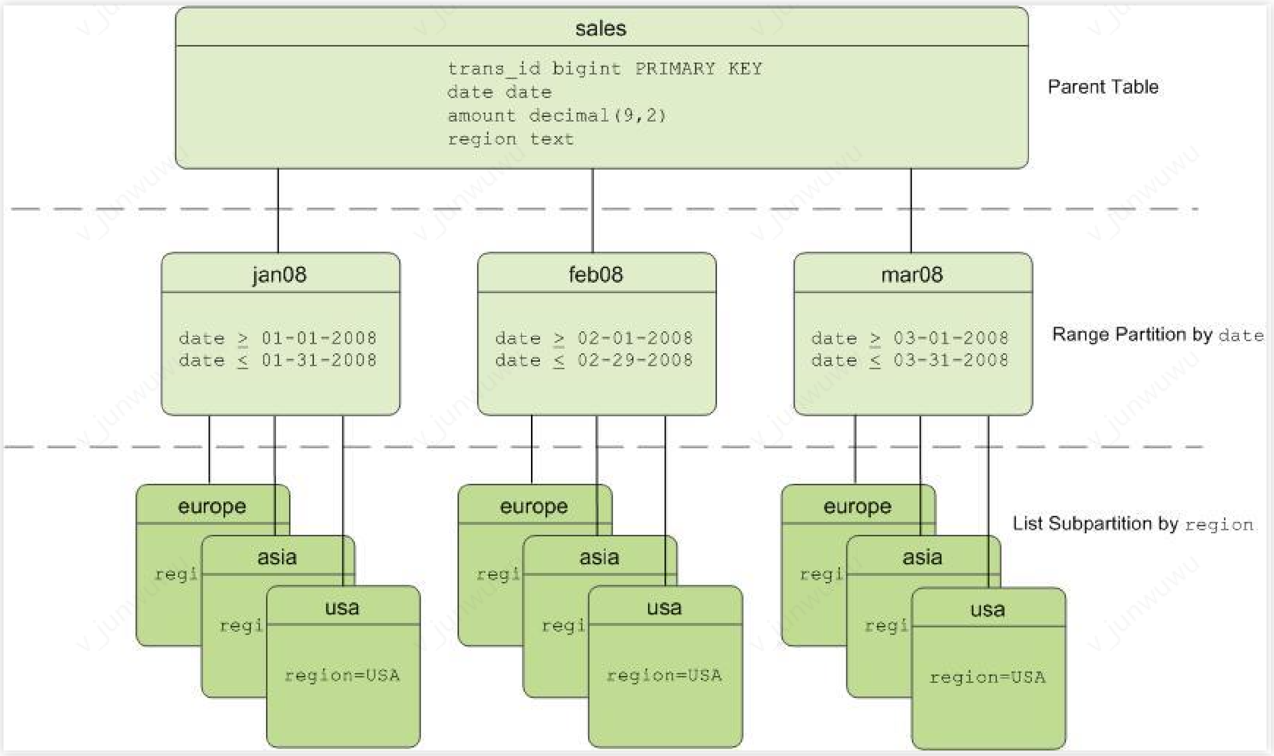
Table Partition Usage

Last updated：2024-02-02 15:36:51

This document describes how to use table partitioning in CDWPG.

Partitioned Table Overview

Partitioned tables are small tables imperceptibly divided from a large table, indicating that you can manipulate the large table without caring about which small table the data actually falls into. CDWPG applies the same table partitioning principle as PostgreSQL, both of which implement table inheritance and constraints.
Below is a sample partitioned table:



Partitioned Table Use Cases

You can consider the following aspects to determine whether to use a partitioned table:
Whether the table data volume is large enough: Partitioning can be used for fact tables with tens to hundreds of millions of data records. There is no absolute criterion for the data volume, and the decision is usually made based on experience and whether you are satisfied with the current performance.

Whether the table has suitable partition fields: If the data volume is large enough, you need to look for suitable fields that can be used for partitioning. Ideally, you can use time dimensions such as day and month if present.
Whether the data in the table has a lifecycle: The data in the data warehouse will not be stored forever and generally has a lifecycle, such as data in the past year. This involves the management of legacy data. If there is a partitioned table, it will be easy to delete legacy data or archive it to a cheaper storage medium such as COS.
Whether the query statement contains partition fields: If a table is partitioned, but none queries contain partition fields, the performance will be lowered rather than improved, because all partitioned tables will be scanned for all queries.

Creating Partitioned Table

Range partition
List partition
A combination of both types
Range partition example:

```
CREATE TABLE test_range_partition
(
    uid int,
    fdate character varying(32)
)
PARTITION BY RANGE(fdate)
(
        PARTITION p1 START ('2018-11-01') INCLUSIVE END ('2018-11-02') EXCLUSIVE,
        PARTITION p2 START ('2018-11-02') INCLUSIVE END ('2018-11-03') EXCLUSIVE,
        DEFAULT PARTITION pdefault
);
```

The above example creates a table by day. If the time span is large, the table creation statement will be very long and inconvenient to write. In this case, you can use the following syntax:

```
CREATE TABLE test_range_partition_every_1
(
    uid int,
    fdate date
)
partition by range (fdate)
(
    PARTITION pn START ('2018-11-01'::date) END ('2018-12-01'::date) EVERY ('1 day'
    DEFAULT PARTITION pdefault
);
```

List partition example:

```
CREATE TABLE test_list_partition
(
    uid int,
    gender char(1)
)
PARTITION BY LIST (gender)
```

```
(
    PARTITION girls VALUES ('F'),
    PARTITION boys VALUES ('M'),
    DEFAULT PARTITION pdefault
);
```

Managing Partitioned Table

Just like a common table, a partitioned table supports many operations as listed below, among others:

Clearing partition

```
ALTER TABLE test_range_partition TRUNCATE PARTITION p1;
```

Dropping partition

```
ALTER TABLE test_range_partition DROP PARTITION p1;
```

Note:

`DROP PARTITION` is followed by the partition name, not the partitioned table name. There is a difference between the two. If the partitioned table is created by using the `EVERY` syntax, you need to query the name of the particular partition through the `pg_partitions` table.

```
ALTER TABLE test_range_partition ADD PARTITION p3 START ('2018-11-03') INCLUSIVE EN
```

Note:

If the partitioned table contains the `DEFAULT` partition, the following error will occur: `ERROR: cannot add RANGE partition "p3" to relation "test_range_partition" with DEFAULT partition "pdefault"` . You can see Rolling Partition for solution.

Rolling Partition

In tables partitioned by time, partitions usually keep rolling forward. For example, if a table is partitioned by day to save data in the past ten days, the partition created ten days ago will be deleted every day, and a new partition will be created to store the latest data.
If there is a default partition, you can use partition split.

```
ALTER TABLE test_range_partition SPLIT DEFAULT PARTITION START ('2018-11-03') INCLU
```

In this way, the new partition is added, while the default partition is retained. Then, the replacement of the old and new partitions can be completed when the old partition is deleted.

Exchanging Partition

Exchanging partition is to exchange a common table with a partitioned table. This feature is very useful in tiered data storage.
For example, if you need to set partitions according to different COS directories, you can use partition exchanging to implement this, so that less queried historical data in a large table can be placed in COS. The syntax is as follows:

```
ALTER TABLE {table_name} EXCHANGE PARTITION {partition_name|FOR (RANK(number))|FOR
```

Querying partition

System tables or views related to partitions are as follows:

```
pg_partition
pg_partition_columns
pg_partition_encoding
pg_partition_rule
pg_partition_templates
pg_partitions
```

Viewing partition information

```
t2=# select * from pg_partitions where partitiontablename = 'test_range_partition_1
-[ RECORD 1 ]------------+----------------------------------------------------------
schemaname               | public
tablename                | test_range_partition
partitionschemaname      | public
partitiontablename       | test_range_partition_1_prt_p1
partitionname            | p1
parentpartitiontablename |
parentpartitionname      |
partitiontype            | range
partitionlevel           | 0
partitionrank            | 1
partitionposition        | 2
partitionlistvalues      |
partitionrangestart      | '2018-11-01'::character varying(32)
partitionstartinclusive  | t
partitionrangeend        | '2018-11-02'::character varying(32)
partitionendinclusive    | f
partitioneveryclause     |
partitionisdefault       | f
partitionboundary        | PARTITION p1 START ('2018-11-01'::character varying(32))
parenttablespace         | pg_default
partitiontablespace      | pg_default
```

Viewing partition definition

```
t2=# select pg_get_partition_def('test_range_partition'::regclass,true);
-[ RECORD 1 ]--------+-------------------------------------------------------------
pg_get_partition_def | PARTITION BY RANGE(fdate)
                     |           (
                     |           PARTITION p1 START ('2018-11-01'::character varyin
                     |           PARTITION p2 START ('2018-11-03'::character varyin
                     |           DEFAULT PARTITION pdefault
                     |           )
```

Best Practices for Partitioned Table

Partition granularity

Range partitioned tables usually involves granularity selection, such as partitioning by day, week, or month. The finer the granularity, the less data per table, but the more the partitioned tables, and vice versa.

There is no absolute criterion for the number of partitioned tables. Generally, 100 is a high number in this regard.

If there are too many partitioned tables, various problems will occur; for example, the query optimizer will be slower to generate execution plans, and many maintenance tasks will also become slower, such as vacuuming, segment recovering, cluster scaling, and disk usage checking.

Query statement

In order to take full advantage of table partitioning, it is better to include a partition condition in a query statement. The ultimate goal is to scan as few partitioned tables as possible.

Extension Usage

Last updated：2024-02-02 15:36:51

Background

CDWPG is based on the massively parallel processing (MPP) cluster architecture of PostgreSQL, so it is compatible with certain extensions in the PostgreSQL ecosystem. This document lists such extensions and how to use them. If you need to use other extensions, contact us.

Extension List

postgis: v2.5.2, a spatial database extension as detailed in geospatial.
hll: v2.14, a HyperLogLog algorithm extension as detailed in postgresql-hll.
roaringbitmap: v0.2.66, a compressed bitmap algorithm extension as detailed in gpdb-roaringbitmap.
orafce: v3.7, an Oracle function compatibility extension as detailed in orafce.
pgcrypto: v1.1, an encryption extension as detailed in pgcrypto.
fuzzystrmatch: v1.0, an extension used to determine similarities and distance between strings as detailed in fuzzystrmatch.

Directions

CDWPG does not create the above extensions by default. You can create or delete extensions as needed with the following syntax:

```
CREATE EXTENSION {extension name};
DROP EXTENSION {extension name}
```

Note:

The scope of extensions is the database, which means that within each database where an extension is needed, you need to first run the `CREATE` statement.

To see the extensions currently installed in the database and their versions, use the following syntax:

```
test_db=> \\dx
                                                    List of installed extensions
  Name    | Version |   Schema   |                                                Descript
ion
---------+---------+------------+-------------------------------------------------------
---------------------------------------------
 hll     | 2.14    | public     | Type for storing hyperloglog data
 orafce  | 3.7     | public     | Functions and operators that emulate a subset of
functions and packages from the Oracle RDBMS
 plpgsql | 1.0     | pg_catalog | PL/pgSQL procedural language
(3 rows)
```

Cold Data Backup

Last updated：2024-02-02 15:36:51

This document describes how to back up data regularly.

Background

Although CDWPG has a master-standby data storage architecture, some scenarios require cold backup for all important data, such as when data is abnormally deleted. As automatic cold backup is not supported by CDWPG at the moment, manual operation is required. In CDWPG, COS is used as the storage medium for data backup. For related operations on COS data, see Importing and Exporting COS Data at High Speed with External Table.

Impact

Note that backing up data as instructed in this document may have the following impact on the cluster:
1. Script execution will increase the cluster load, especially the overheads at the network layer. Therefore, we recommend you evaluate the backup time and perform it during off-peak hours of your business.
2. A COS extension will be created in each database during script execution.
3. A COS external table will be created for each table that needs to be backed up during script execution and deleted after the backup is completed.

Issues

Note that you may encounter the following issues when backing up data as instructed in this document:

Error Message	Solution
`ERROR: permission denied for external protocol cos`	`GRANT ALL ON PROTOCOL cos TO {backup_user}`
`ERROR: permission denied for schema {schame_name}`	`GRANT ALL ON SCHEMA {schame_name} to {backup_user}`
`ERROR: permission denied for relation {table_name}`	`GRANT SELECT ON {table_name} to {backup_user}`

Directions

You can use the following `shell` script to back up all data in the CDWPG cluster and scale the cluster as needed to complete regular cold backups with crontab. You can also download and use backup_cdw_v101.sh.

Note:

Deleting a writable external table will not delete the corresponding data in COS.

We recommend you back up data during off-peak hours of your system, as the backup process may increase the system load.

The backup duration depends on the data volume and cluster specification. Simply put, the more the cluster nodes, the faster the backup.

```
#!/bin/bash

set -e

# CDWPG connection parameters that need to be entered
PWD='' # Required
HOST='' # Required
USER='' # Required
DEFAULT_DB='postgres'

# Backup parameters that need to be entered
```

```
SECRET_ID='' # Required
SECRET_KEY='' # Required
COS_URL='' # Required, such as `test-1301111111.cos.ap-guangzhou.myqcloud.com`
COMPRESS_TYPE='gzip' # Whether the files in COS are in compressed format. Valid val

echo -e "\\n`date "+%Y%m%d %H:%M:%S"` backup task start\\n"

# Step 1. Get the list of databases
db_list=`PGPASSWORD=${PWD} psql -t -h ${HOST} -p 5436 -d ${DEFAULT_DB} -U ${USER} -

# Step 2. Traverse the databases that need to be backed up
for db in $db_list
do
    # `template0`, `template1`, and `gpperfmon` are templates and system database a
        if [ "$db" = "template0" -o $db = "template1" -o $db = "gpperfmon" ];then
                continue
        fi

        echo -e "\\n************************************************"
        echo -e "backup database:{$db} start"
        db_start=`date +%s`

    # Step 3. Get the current date
    # Use the date as part of the COS path to distinguish between data backed up on
    cur_date=`date +%Y%m%d`

    # Step 4. Get the list of tables that need to be backed up
    # External, virtual, temporary, and replicated tables (not supported currently)
    table_list=`PGPASSWORD=${PWD} psql -t -h ${HOST} -p 5436 -d ${db} -U ${USER} -c

    # Step 5. Create a COS extension
    PGPASSWORD=${PWD} psql -h ${HOST}  -p 5436 -d ${db} -U ${USER} -c "CREATE EXTEN

    # Step 6. Traverse the list and perform backups in sequence
    for table in $table_list
    do
        sleep 1
        table_start=`date +%s`
        echo -e "backup ${table} start"
            # Here, a name suffix must be added in the format of `{schema}.{table}`
            backup_table="${table}_cdw_backup_cos"

        # Step 7. Create COS backup tables
        PGPASSWORD=${PWD} psql -h ${HOST}  -p 5436 -d ${db} -U ${USER} -c "CREATE W


        # Step 8. Import the data of original tables to backup tables
        PGPASSWORD=${PWD} psql -h ${HOST}  -p 5436 -d ${db} -U ${USER} -c "INSERT I

        # Step 9. Delete the backup external tables
        # Note: Deleting an external table will not delete the corresponding data i
        PGPASSWORD=${PWD} psql -h ${HOST}  -p 5436 -d ${db} -U ${USER} -c "DROP EXT

        table_end=`date +%s`
        echo -e "backup ${table} done, cost $[table_end - table_start]s\\n"
    done

        db_end=`date +%s`
        echo -e "backup database:{$db} done, cost $[db_end - db_start]s"
        echo -e "*************************************************\\n"
done
```

Statistics and Space Maintenance

Last updated：2024-02-02 15:36:51

Background

Cluster statistics are critical to the use of clusters, and CDWPG's query optimizer is based on the dynamically calculated cost to determine how to make selections. How is the cost calculated? Usually, the calculation is based on the cost model and statistics. The cost model reasonableness and statistics accuracy will affect the effectiveness of query optimization.

The utilization of cluster tablespaces also affects the query cost. When a table has `UPDATE` operations (including `INSERT VALUES` , `UPDATE` , `DELETE` , `ALTER TABLE` , and `ADD COLUMN`), it will leave garbage data that is no longer used in the system table and the updated table, causing system performance degradation and taking up a lot of disk space. Therefore, you need to regularly monitor the data bloat of the table.

The following details the regular monitoring and maintenance of statistics and data bloat.

Statistics Collection

`ANALYZE` collects table statistics in a database and stores the results in the `pg_statistic` system directory. The query planner uses such statistics to help determine the most efficient execution plan for a query. The statistics contain various information such as the amount of data and indexes in the table. A good query plan is based on accurate table statistics.

`ANALYZE` description

`ANALYZE` is a command provided by Greenplum to collect statistics and supports column, table, and database granularities as shown below:

```
CREATE TABLE foo (id int NOT NULL, bar text NOT NULL) DISTRIBUTED BY (id); // Creat
ANALYZE foo(bar);  // Collect statistics of the `bar` column only
ANALYZE foo; // Collect statistics of the `foo` table
ANALYZE; // Collect statistics of all tables in the current database. You need to h
```

Use limits of `ANALYZE`

`ANALYZE` will put a `SHARE UPDATE EXCLUSIVE` lock on the target table, which will conflict with DDL statements.

`ANALYZE` speed

`ANALYZE` is a sampling statistical algorithm that usually does not scan all data in a table, but still consumes some time and computing resources for large tables.

Time to use `ANALYZE`

As mentioned above, `ANALYZE` locks tables and consumes system resources, so it is important to run it at the right time and as little as possible. We recommend you run `ANALYZE` in the following four scenarios.
After data is batch loaded; for example, after data is imported to a newly created table.
After an index is created.
After `INSERT` , `UPDATE` , and `DELETE` operations are performed on a large amount of data.
After `VACUUM FULL` is executed.

Analyzing partitioned table

As long as you keep the default value and do not modify the system parameter `optimizer_analyze_root_partition` , there is no difference in manipulating a partitioned table. Just run `ANALYZE` on the root table, and the system will automatically collect the statistics of partitioned tables on all leaf nodes.
If the number of partitioned tables is high, running `ANALYZE` on the root table can be time-consuming. Partitioned tables are usually time-dimensional, and historical partitioned tables are not modified; therefore, we recommend you run `ANALYZE` separately in partitions where data changes.

Data Bloat

The Greenplum database's heap table uses PostgreSQL's multiple version concurrency control (MVCC) storage implementation. The database will logically delete a deleted or updated row, but an invisible image of that row will remain in the table. As more operations are performed, the table will have more and more invisible data, which will take up a lot of storage space, thereby causing a serious performance degradation in table operations. Additionally, the data bloat can occupy a lot of space, which needs to be regularly fixed.

Data bloat monitoring

The `gp_toolkit` schema provides a `gp_bloat_diag` view that determines the table bloat by the ratio of the actual number of pages to the expected number of pages. To use this view, make sure that the latest statistics are collected for all tables in the database. Then, run the following SQL:

```
gpadmin=# SELECT * FROM gp_toolkit.gp_bloat_diag;
 bdirelid | bdinspname | bdirelname | bdirelpages | bdiexppages |                bd
----------+------------+------------+-------------+-------------+------------------
    21488 | public     | t1         |          97 |           1 | significant amoun
(1 row)
```

Here, `bdirelpage` is the actual number of pages and `bdiexppages` is the expected number of pages in the `t1` table. A bloat ratio exceeding 4 will be recorded in the table, and there may also be a slight bloat even when no ratio is recorded. You can also compare the space of tables at different times to determine whether data has bloated.

Cleaning data bloat in table

The `VACUUM <tablename>` command adds expired rows to the shared free space map, so such space can be reused. When `VACUUM` is run periodically on a table with frequent `UPDATE` operations, the space occupied by expired rows can be reused quickly, easing table bloat. The cycle to run `VACUUM` is determined by the speed of `UPDATE` and `DELETE` on the table.

Note:

`VACUUM` holds a `SHARE UPDATE EXCLUSIVE` lock as `ANALYZE` does and may interlock with DDL operations.

When the table experiences a significant bloat, `VACUUM` can only slow down the process but not immediately reclaim the space. Therefore, you need to run `VACUUM FULL` to immediately reclaim all the bloat space. However, `VACUUM FULL` will add `ACCESS EXCLUSIVE` to the manipulated table, during which all other DDL and DML statements on the table will be locked. Proceed with caution when running `VACUUM FULL` as the operation may be time-consuming for large tables.

Another way to address bloat is to redistribute data in the table:

1. Record the distribution key of the table.
2. Change the distribution policy of the table to random distribution.

```
ALTER TABLE <tablename> SET WITH (REORGANIZE=false)
            DISTRIBUTED randomly;
```

3. Change the distribution policy back to the initial one.

```
ALTER TABLE <table_name> SET WITH (REORGANIZE=true)
DISTRIBUTED BY (<original distribution columns>);
```

Handling index bloat

The `VACUUM FULL` command will only reclaim spaces from tables. To reclaim spaces from indexes, you need to rebuild them with the `REINDEX` command.

```
REINDEX TABLE <table_name>    --- Rebuild all indexes in the table
REINDEX INDEX <index_name> --- Rebuild a specified index
```

Note:

Note that both `REINDEX` and `VACUUM FULL` will add `ACCESS EXCLUSIVE` .

Regular Cluster Maintenance

When using a cluster, you need to eliminate data bloat and maintain statistics regularly so that the cluster can deliver an optimal performance.

Cleaning without table locking

As mentioned above, `VACUUM <tablename>` can mark reclaimable space without table locking and slow down data bloat. Cleaning without locking the table will not affect data table reads/writes, but DDL statements cannot be used, and some CPU and memory resources will be used.

Recommended frequency:

Once a day or at least twice a week if a large amount of data is updated in real time and many data records change every day.

Once a week under normal circumstances or once a month if data is not updated frequently.

Use the following script to clean a table with a cron job.

```
#!/bin/bash
export PGHOST=xxx.xxx.x.x
export PGPORT=5436
export PGUSER=test
export PGPASSWORD=test
db="test"
psql -d $db -e -c "VACUUM test_table;"
```

Full cleaning with table locking

If the data in a table is updated and deleted frequently, we recommend you plan a business pause window to run `VACUUM FULL` and `REINDEX` to reclaim all the bloat space in the table. Cleaning with table locking will put an `ACCESS EXCLUSIVE` lock on the manipulated table, during which you cannot perform any operations on the table.

1. Run `VACUUM FULL <tablename>` .
2. Run `REINDEX TABLE <tablename>` (you can skip this step for tables without indexes).
3. Run `ANALYZE <tablename>` .

Recommended frequency: Once a week or once a day if almost all data is updated daily.

Use the following script to regularly clean cluster tables, preferably during off hours in the early morning on the weekend.

```
#!/bin/bash
export PGHOST=xxx.xxx.x.x
export PGPORT=5436
export PGUSER=test
export PGPASSWORD=test
db="test"
psql -d $db -e -c "VACUUM FULL test_table;"
psql -d $db -e -c "REINDEX TABLE test_table;"
psql -d $db -e -c "ANALYZE test_table;"
```