Alibaba Cloud
AnalyticDB for PostgreSQL

Best Practices

Issue: 20190514

Legal disclaimer

Generic conventions

Table -1: Style conventions

Style	Description	Example
⛔	This warning information indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	⛔ Danger: Resetting will result in the loss of user configuration data.
⚠	This warning information indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	⚠ Warning: Restarting will cause business interruption. About 10 minutes are required to restore business.
📋	This indicates warning information, supplementary instructions, and other content that the user must understand.	❗ Notice: Take the necessary precautions to save exported data containing sensitive information.
	This indicates supplemental instructions, best practices, tips, and other content that is good to know for the user.	📋 Note: You can use Ctrl + A to select all files.
>	Multi-level menu cascade.	Settings > Network > Set network type
Bold	It is used for buttons, menus, page names, and other UI elements.	Click OK.
`Courier font`	It is used for commands.	Run the `cd / d C :/ windows` command to enter the Windows system folder.
Italics	It is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	It indicates that it is a optional value, and only one item can be selected.	`ipconfig` *[-all|-t]*

Style	Description	Example
{} or {a|b}	It indicates that it is a required value, and only one item can be selected.	`swich` *{stand | slave}*

Contents

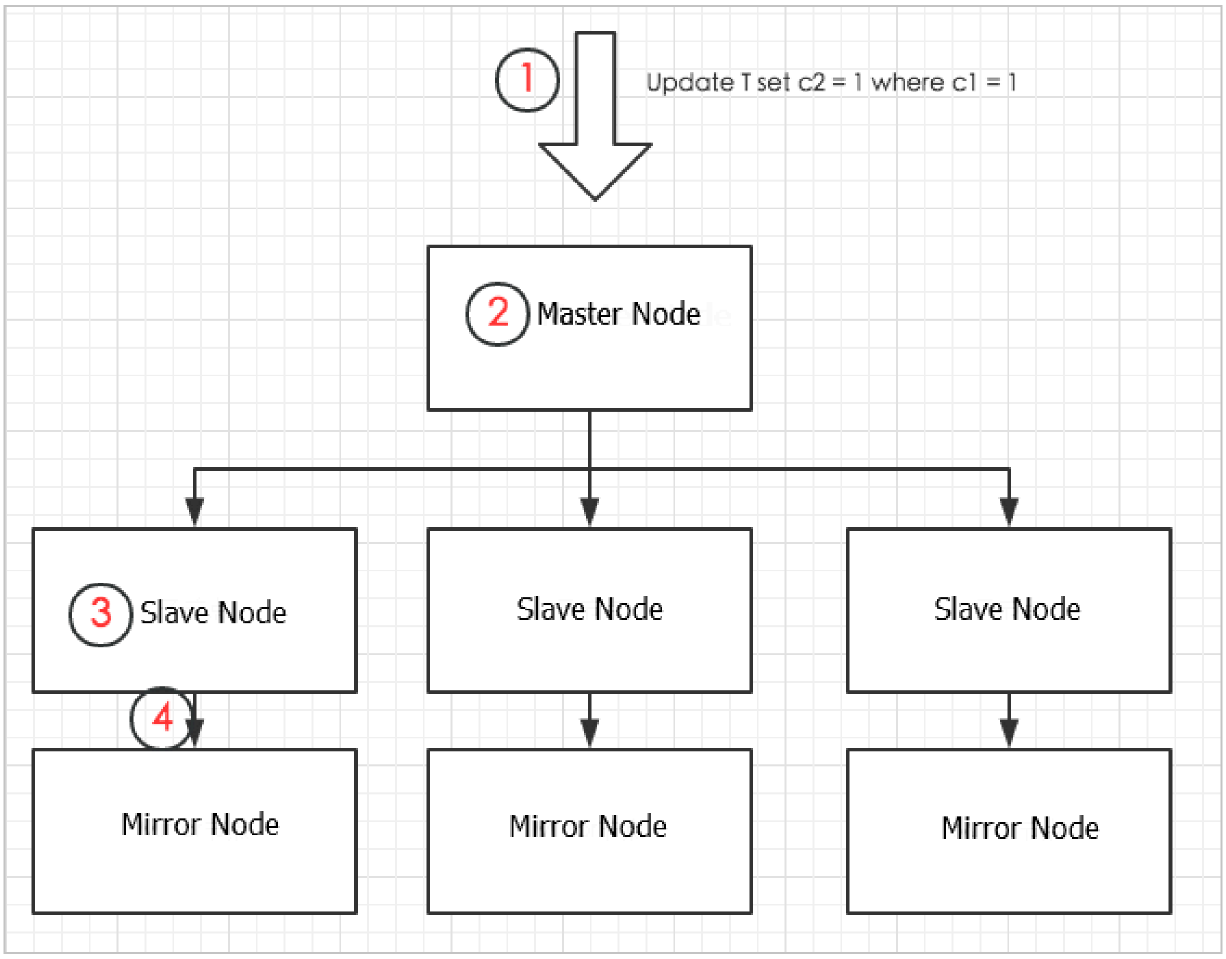
1 Bulk update

Update, also called Merge, indicates updating the latest data to AnalyticDB for PostgreSQL. If the updated data already exists, it replaces the old version. If the updated data does not exist, it is inserted to the database. Such data merge is usually completed offline. For example, you can set to update data on a daily basis to AnalyticDB for PostgreSQL. Some users may require real-time updates, that is, the latency is at the minute or second level.

This document describes how to merge data in AnalyticDB for PostgreSQL and explains the principle behind it. In addition, you can learn how to use the bulk operation to update multiple data.

Simple update

Data merge is about modifying the data, that is, running the Update, Delete, Insert, or Copy operations. Take an Update operation for example, updating the record on a single row in a column-store table. The following figure shows the data updating process in AnalyticDB for PostgreSQL.



The procedure is described as follows:

1. The user sends an Update SQL request to the master node.
2. The master node initiates distributed transactions, locking the table to be updated (AnalyticDB for PostgreSQL does not allow concurrent updates to the same table), and distributing updating requests to matched slave nodes.
3. Slave nodes scan the index to locate the data to update, and update the data. For column-store tables, the updating logic is to delete the old data row and write the new data row at the end of the table. The updated data page in the column-store table is written to the memory cache, and the change in the corresponding table file length (because data is written to the table end, the length of the correspond ing table file is increased) is written to the log (xlog file).
4. Before the Update process ends, the updated data page and xlog file in the memory are both synchronized to the mirror node. After the synchronization is complete, the master node ends the distributed transaction, and returns the message about successful execution to the user.

The whole process is long and contains lots of operations, such as SQL statement parsing, transactions distributing, locking, connection establishment between the master node and slave nodes, and the synchronization of data and log between slave nodes and the mirror node. These operations all consume CPU or I/O resources and prolong the response of the request.

Therefore, for AnalyticDB for PostgreSQL, we recommend that you avoid updates to a single data row, and try to update data by using bulk operations as much as possible. That is:

- Put updates in one SQL statement to reduce the overhead for statement parsing, node communications, and data synchronization.
- Put updates in one transaction to avoid unnecessary overhead.

Bulk update

Follow these steps to use one SQL statement to update multiple independent data rows.

1. Prepare the target table

Suppose that the table to be updated is target_table. The target_table is defined as follows.

```
create   table   target_tab  le ( c1   int ,  c2   int ,  primary
key  ( c1 ));
insert   into   target_tab  le   select   generate_s  eries ( 1 ,
10000000 );
```

The target table is usually quite big. Suppose that you want to insert 10 million rows of data to target_table. The target_table is indexed to facilitate updates. A primary key is defined and a unique index is included consequently.

2. Prepare the stage table

The stage table (source_table in this example) is necessary for bulk update. It is a temporary table created for updating data. To update the data in target_table, you first insert the new data to source_table, and then import the new data by using the *Copy command*, *OSS external table*, or other means to target_table.

In the following example, some data is directly generated in source_table.

```
create   table   source_tab  le ( c1   int ,  c2   int );
```

```
insert   into   source_tab  le   select   generate_s  eries ( 1 ,
100 ),  generate_s  eries ( 1 , 100 );
```

3. Bulk update

After the source_table data is ready, run the `update set … from … where ..` statement.

Note:
To utilize the index to a maximum extent, you can use `set optimizer = on` to start the ORCA optimizer before the update operation. If the ORCA optimizer is not started, you can run `set enable_nes tloop = on` to use the index.

```
set   optimizer = on ;
update   target_tab  le   set   c2  =  source_tab  le . c2   from
source_tab  le   where   target_tab  le . c1 =  source_tab  le . c1 ;
```

The update operation' s query plan is as follows:

```
=>  explain   update   target_tab  le   set   c2  =  source_tab
 le . c2   from   source_tab  le   where   target_tab  le . c1 =
 source_tab  le . c1 ;
                                                          QUERY
 PLAN
-------------------------------------------------------------------------------------------
  Update   ( cost = 0 . 00 .. 586 . 10   rows = 25   width = 1 )
   ->   Result   ( cost = 0 . 00 .. 581 . 02   rows = 50   width = 26
 )
         ->   Redistribu  te   Motion   4 : 4   ( slice1 ;  segments
 :  4 )  ( cost = 0 . 00 .. 581 . 02   rows = 50   width = 22 )
               Hash   Key :  public . target_tab  le . c1
               ->   Assert   ( cost = 0 . 00 .. 581 . 01   rows = 50
 width = 22 )
                     Assert   Cond :  NOT   public . target_tab  le .
 c1   IS   NULL
                     ->   Split   ( cost = 0 . 00 .. 581 . 01   rows =
 50   width = 22 )
                           ->   Nested   Loop   ( cost = 0 . 00 .. 581
 . 01   rows = 25   width = 18 )
                                  Join   Filter :  true
                                 ->   Table   Scan   on   source_tab
 le   ( cost = 0 . 00 .. 431 . 00   rows = 25   width = 8 )
                                 ->   Index   Scan   using
 target_tab  le_pkey   on   target_tab  le   ( cost = 0 . 00 .. 150 .
 01   rows = 1   width = 14 )
                                       Index   Cond :  public .
 target_tab  le . c1  =  source_tab  le . c1
```

From the plan, AnalyticDB for PostgreSQL uses the index. But if you add more data to source_table, the optimizer may deem that using Nest Loop associated method and index scanning is not as efficient as dropping the index. As a result, it may use Hash associated method and table scanning for the execution. For example,

```
postgres =>  insert   into   source_tab  le   select   generate_s
eries ( 1 ,  1000 ),  generate_s  eries ( 1 , 1000 );
INSERT   0   1000
postgres =>  analyze   source_tab  le ;
ANALYZE
postgres =>  explain   update   target_tab  le   set   c2  =
source_tab  le . c2   from   source_tab  le   where   target_tab  le
. c1 =  source_tab  le . c1 ;
                                              QUERY   PLAN
-----------------------------------------------------------------------------------------------
 Update   ( cost = 0 . 00 .. 1485 . 82   rows = 275   width = 1 )
  ->   Result   ( cost = 0 . 00 .. 1429 . 96   rows = 550   width =
26 )
        ->   Assert   ( cost = 0 . 00 .. 1429 . 94   rows = 550
width = 22 )
              Assert   Cond :  NOT   public . target_tab  le . c1
IS   NULL
              ->   Split   ( cost = 0 . 00 .. 1429 . 93   rows = 550
  width = 22 )
                    ->   Hash   Join   ( cost = 0 . 00 .. 1429 . 92
rows = 275   width = 18 )
                           Hash   Cond :  public . target_tab  le .
c1   =  source_tab  le . c1
                          ->   Table   Scan   on   target_tab  le   (
cost = 0 . 00 .. 477 . 76   rows = 2500659   width = 14 )
                          ->   Hash   ( cost = 431 . 01 .. 431 . 01
rows = 275   width = 8 )
                                ->   Table   Scan   on   source_tab
le   ( cost = 0 . 00 .. 431 . 01   rows = 275   width = 8 )
```

The bulk update approach described reduces SQL compilation, inter-node communications, transactions, and other overheads, and can greatly boost data updating performance and reduce resource consumption.

Bulk delete

For delete operations, you can use a stage table similar to that used for bulk update, and use the following delete command with a “Using” clause to delete data by bulk:

```
delete   from   target_tab  le   using   source_tab  le   where
target_tab  le . c1  =  source_tab  le . c1 ;
```

The bulk delete operation also uses the index.

```
explain   delete   from   target_tab  le   using   source_tab  le
where   target_tab  le . c1  =  source_tab  le . c1 ;
                                             QUERY   PLAN
-----------------------------------------------------------------------------------------------
 Delete   ( slice0 ;  segments :  4 )  ( rows = 50   width = 10 )
  ->   Nested   Loop   ( cost = 0 . 00 .. 41124 . 40   rows = 50
width = 10 )
        ->   Seq   Scan   on   source_tab  le   ( cost = 0 . 00 .. 6
. 00   rows = 50   width = 4 )
```

```
        ->   Index   Scan   using   target_tab  le_pkey   on
target_tab  le   ( cost = 0 . 00 .. 205 . 58   rows = 1   width = 14
)
               Index   Cond :  target_tab  le . c1  =  source_tab  le
. c1
```

Merge data by using Delete and Insert

To merge data, you must first put the data to merge to the stage table. If you know in advance that the data to be merged already exists in the target table, you can use update statements to merge the data. But in most cases, part of the data to be merged already exists in the target table, and part of it is new, with no matched records in the target table. In this case, you can use a combination of bulk delete and bulk insert. The sample code is as follows.

```
set   optimizer = on ;
delete   from   target_tab  le   using   source_tab  le   where
target_tab  le . c1  =  source_tab  le . c1 ;
insert   into   target_tab  le   select  *  from   source_tab  le ;
```

Update data in real time by using Values() expressions

To use the stage table, you must maintain its lifecycle. Some users want to update data to AnalyticDB for PostgreSQL by bulk in real time, that is, to continuously synchroniz e data or merge data to AnalyticDB for PostgreSQL.

If you use the aforementioned method, you must create and delete (or truncate) the stage table repeatedly. In fact, you can use Values expressions to achieve an effect similar to stage tables, without the effort to maintain the table. The approach is to first splice the data to update into a Values expression, and then run the update or delete commands by using the following method:

```
update   target_tab  le   set   c2  =  t . c2   from  ( values ( 1 ,
1 ),( 2 , 2 ),( 3 , 3 ),…( 2000 , 2000 ))  as   t ( c1 , c2 )  where
  target_tab  le . c1 = t . c1
delete   from   target_tab  le   using  ( values ( 1 , 1 ),( 2 , 2 ),
( 3 , 3 ),…( 2000 , 2000 ))  as   t ( c1 , c2 )  where   target_tab
 le . c1  =  t . c1
```

Note:

Both `set   optimizer = on ;` and `set   enable_nes  tloop = on ;` generate query plans that use indexes. In complicated cases, however, such as multiple index fields or partition tables are involved, the ORCA optimizer must be used to match the index.

2 Improve performance of AnalyticDB for PostgreSQL

AnalyticDB for PostgreSQL is developed based on Greenplum Database and is enhanced with some in-depth extensions by Alibaba Cloud. It is a distributed cloud database that is composed of multiple *groups* to provide MPP (Massively Parallel Processing) data warehousing service.

This document introduces the best practices for using AnalyticDB for PostgreSQL. We recommend that you choose from the mentioned methods to follow in order to improve the performance of AnalyticDB for PostgreSQL, speed up the import process , and reduce the cost.

Use Compressed Column Storage

For tables with infrequent updates and many fields, we recommend that you use Compressed Column Storage. This method increases the compression ratio three-fold while guaranteeing performance, and the import speed is usually faster.

For example, you can add the clause `WITH ( APPENDONLY = true , ORIENTATIO N = column , COMPRESSTY PE = zlib , COMPRESSLE VEL = 3 , BLOCKSIZE = 1048576 )` to the tabulation statements to create compressed column store tables.

For the specific syntax, see *CREATE TABLE*.

Use the Nested Loop JOIN

By default, the Nested Loop JOIN is not enabled for AnalyticDB for PostgreSQL instances. For queries that only involve or return a small amount of data, the performance may not be optimal.

Take the following SQL statement as an example:

```
select  *  from   T1   join   T2   on   T1 . c1  =  T2 . c1   where
  T1 . c2  >= ' 230769548 '  and  T1 . c2  < ' 230769549 '  limit
100 ;
```

In this example, the T1 and T2 tables are both big in size. The selection conditions of `T1 ( T1 . c2 >= ‘ 230769548 ’` and `T1 . c2 < ‘ 23432442 ’)` filter a vast majority of data records and contain LIMIT clauses.

As a result, the query actually involves only a small portion of the total data size. In this case, the Nested Loop JOIN method is optimal.

You can perform the following SET command to activate the Nested Loop JOIN:

```
 show   enable_nes  tloop  ;
  enable_nes  tloop
-----------------
  off
 SET   enable_nes  tloop  =  on  ;
 show   enable_nes  tloop  ;
  enable_nes  tloop
-----------------
  on
 explain   select  *  from   T1   join   T2   on   T1 . c1  =  T2 . c1
   where   T1 . c2  >= ' 230769548 '  and   T1 . c2  < ' 23432442 '
 limit   100 ;
                                             QUERY   PLAN
--------------------------------------------------------------------------------
  Limit   ( cost = 0 . 26 .. 16 . 31   rows = 1   width = 18608 )
   ->   Nested   Loop   ( cost = 0 . 26 .. 16 . 31   rows = 1   width
 = 18608 )
         ->   Index   Scan   using   T1   on   c2   ( cost = 0 . 12 ..
 8 . 14   rows = 1   width = 12026 )
               Filter : (( c2  >= ' 230769548 ':: bpchar )  AND  ( c2
  < ' 230769549 ':: bpchar ))
         ->   Index   Scan   using   T2   on   c1   ( cost = 0 . 14 ..
 8 . 15   rows = 1   width = 6582 )
               Index   Cond : (( c1 ):: text  = ( T1 . c1 ):: text )
```

From the query plan, the T1 and T2 tables adopt the Nested Loop JOIN, and achieve the optimal performance.

Use the ORCA optimizer

AnalyticDB for PostgreSQL supports the ORCA optimizer. When you perform a complicated SQL statement and find the unsatisfactory performance, you can try the ORCA optimizer.

You can enable the ORCA by running the following SET command in the database connection.

Note:
The SET command acts at the connection level and is only valid within the same connection. You need to run the SET command again for a new connection to enable the ORCA.

```
 EXPLAIN  < SQL   text >
 SET   optimizer  =  on ;
```

```
EXPLAIN  < SQL   text >
```

In the preceding example, you can view that the query plan uses the EXPLAIN command before and after enabling the ORCA respectively. In this way, you can check whether the ORCA has actually changed the SQL query plan.

Use a compression method

Now, AnalyticDB for PostgreSQL supports two compression methods for storage: zlib and RLE.

- RLE is applicable to the scenario where the same data values are physically stored continuously.
- Zlib is applicable to other scenarios.

The compression approach can be specified at the field level or table level. For details, see *CREATE TABLE*.

Use other numeric types

If the query contains the unique value statistics operation of COUNT(DISTINCT), we recommend that you do not use the string or numeric type for the statistic fields, but try to use other numeric types (such as integer type). This method can improve the performance several-fold.

3 Scheduled maintenance tasks

When an update operation (including INSERT VALUES, UPDATE, DELETE, and ALTER TABLE ADD COLUMN) is performed on a system, junk data that may no longer be used is left in the system table and the updated data table. This junk data reduces the system performance and takes up a large amount of disk space. We recommend that you clear such data on a regular basis by following the methods provided in this document.

Clear junk data without locking the table

You can clear some junk data without locking the table. The method is as follows.

- Command: connect to every database, log on to the database as the database owner, and run the `VACUUM` command.
- Frequency: at least once a day.
  - If data is updated in real time (that is, INSERT VALUES, UPDATE, and DELETE operations are performed continuously), we recommend that you run the `VACUUM` command once every two hours.
  - If data update is performed by bulk once a day, you can run the command once after the bulk update every day.
- Impact to the system: no table is locked, and tables can be read and written to normally. But it may increase the CPU and I/O usage, and impact query performance.
- Example: you can use the following Linux Shell script file and run it as a scheduled crontab task.

```
#!/ bin / bash
 export   PGHOST = myinst . gpdb . rds . tbsite . net
 export   PGPORT = 3432
 export   PGUSER = myuser
 export   PGPASSWORD = mypass
# do   not   echo   command ,  just   get   a   list   of   db
 dblist =` psql  - d   postgres  - c  " copy  ( select   datname
 from   pg_stat_da  tabase )  to   stdout "`
 for   db   in  $ dblist  ;  do
    # skip   the   system   databases
     if  [[ $ db  ==  template0  ]] ||  [[ $ db  ==  template1  ]] ||
 [[ $ db  ==  postgres  ]] || [[ $ db  ==  gpdb  ]] ;  then
         continue
     fi
     echo   processing  $ db
    # vacuum   all   tables   ( catalog   tables / user   tables )
     psql  - d  $ db  - e  - a  - c  " VACUUM ;"
```

```
done
```

Clear junk data during maintenance windows

You can clear all junk data during maintenance windows of services when services are suspended. The method is as follows.

- Command: connect to every database, and log on to the database as the database owner (you must have the owner permission for all the operation objects).
  1. Run the `REINDEX   SYSTEM  < database   name >` command.
  2. Run the `VACUUM   FULL  < table   name >`, `REINDEX   TABLE  < table   name >` command on every data table (non-system table).
- Frequency: at least once per week. If almost all the data is updated every day, you can run the command once a day.
- Impact on the system: the command locks tables for VACUUM FULL or REINDEX and these tables become not readable or writable. This may increase the CPU and I /O usage.
- Example: you can use the following Linux Shell script file and run it as a scheduled crontab task.

```
#!/ bin / bash
 export   PGHOST = myinst . gpdb . rds . tbsite . net
 export   PGPORT = 3432
 export   PGUSER = myuser
 export   PGPASSWORD = mypass
# do   not   echo   command ,  just   get   a   list   of   db
 dblist =` psql  - d   postgres  - c  " copy  ( select   datname
 from   pg_stat_da  tabase )  to   stdout "`
 for   db   in  $ dblist  ;  do
    # skip   system   databases
     if  [[ $ db  ==  template0  ]] ||  [[ $ db  ==  template1  ]] ||
 [[ $ db  ==  postgres  ]] || [[ $ db  ==  gpdb  ]] ;  then
        continue
     fi
     echo   processing   db  "$ db "
    # do   a   normal   vacuum
     psql  - d  $ db  - e  - a  - c  " VACUUM ;"
    # reindex   system   tables   firstly
     psql  - d  $ db  - e  - a  - c  " REINDEX   SYSTEM  $ db ;"
    # use   a   temp   file   to   store   the   table   list ,  which
   could   be   vary   large
     cp  / dev / null   tables . txt
    # query   out   only   the   normal   user   tables ,  excluding
 partitions   of   parent   tables
     psql  - d  $ db  - c  " copy  ( select  '\"'|| tables .
 schemaname ||'\".' || '\"'|| tables . tablename ||'\"'  from  (
 select   nspname   as   schemaname ,  relname   as   tablename   from
   pg_catalog . pg_class ,  pg_catalog . pg_namespa  ce ,  pg_catalog
 . pg_roles   where   pg_class . relnamespa  ce  =  pg_namespa  ce
 . oid   and   pg_namespa  ce . nspowner  =  pg_roles . oid   and
   pg_class . relkind =' r '  and  ( pg_namespa  ce . nspname  = '
 public '  or   pg_roles . rolsuper  = ' false ' ) )  as   tables (
 schemaname ,  tablename )  left   join   pg_catalog . pg_partiti  ons
   on   pg_partiti  ons . partitions  chemaname = tables . schemaname
   and   pg_partiti  ons . partitiont  ablename = tables . tablename
   where   pg_partiti  ons . partitiont  ablename   is   null )  to
 stdout ;" >  tables . txt
     while   read   line ;  do
        # some   table   name   may   contain   the  $  sign ,  so
 escape   it
         line =` echo  $ line  | sed  ' s /\\\$/\\\\\\\$/ g '`
         echo   processing   table  "$ line "
        # vacuum   full   this   table ,  which   will   lock   the
 table
         psql  - d  $ db  - e  - a  - c  " VACUUM   FULL  $ line ;"
        # reindex   the   table   to   reclaim   index   space
         psql  - d  $ db  - e  - a  - c  " REINDEX   TABLE  $ line ;"
     done  < tables . txt
 done
```

4 Special characters in import

AnalyticDB for PostgreSQL supports multiple data importing methods:

- *Import and export data in parallel by using OSS*
- *Import data from MySQL*
- *Import data from PostgreSQL*
- *Import data by using the COPY command*
- *Synchronize data by using Data Integration*

Special characters often cause import failures during data imports. This document describes how to pre-process special characters in the imported data in advance to eliminate problems arising thereof.

In the aforementioned import methods, tools used for importing data from MySQL and PostgreSQL automatically escape and package the special characters, so you can directly use the tools without any additional setting. The following content focuses on using OSS and the COPY command to import data and describes how to process the special characters.

Import data in parallel by using OSS

During data import, every line in a file are often regarded as a tuple, and the data in each column are divided by specifying a delimiter in each line. The following content introduces the usage and constraints of the delimiter, and how to handle the special characters in each column.

Delimiter

In the syntax of creating an OSS external table, you can specify a DELIMITER after the FORMAT clause as follows:

```
FORMAT  ' TEXT ' ( DELIMITER  ',')
```

- For `FORMAT  ' TEXT '`, `DELIMITER` is `'\ t '` by default.
- For `FORMAT  ' CSV '`, `DELIMITER` is `','` by default.

You can also define your own delimiter, on the premise that the custom delimiter meets the following constraints, as stipulated in the external table creation syntax:

- It must be an ASCII character, and must not be a Chinese character, or two or more ASCII characters.
- `'\ n '`and `'\ r '` are not supported.
- Escape characters except `'\ n '`and `'\ r '` are supported, with "E" or "e" added preceding the character.
- The escape character `'\ t '` with no "E" added is also supported.
- For the TEXT format, you can set the DELIMITER to OFF and use the single-column external tables.

To read data properly, the content of the OSS file you provide must strictly abide by the delimiter you set.

Special characters in the data

During data imports, the use cases where may see special characters include the following:

- The column contains the same character as the delimiter.
  - If you are using the TEXT format, you must add an ESCAPE character before each DELIMITER. The ESCAPE character can be specified by using the following command when you create an external table. The default value is the backslash (\).

    ```
    FORMAT  ' TEXT ' ( ESCAPE  '\' )
    ```
  - If you are using the CSV format, you must add double quotation marks (") before each DELIMITER.
- The column contains Chinese characters. OSS external table supports Chinese character data. But to make sure the display is correct, you must encode the created external table as follows:

  ```
  ENCODING  ' UTF8 '
  ```
- The column contains null data. You can set to match null values to a character, and replace the specified character with null during data imports. For the CSV format, the default value is a null value with no quotation marks. For the TEXT format, the

default value is "\N". The following command maps space to null. If the column is a space, then the value for the column is null in the data imported from the OSS file.

```
FORMAT  ' text ' ( null  ' ' )
```

- The column contains escape characters. You can add "ESCAPE" before the escape characters. The ESCAPE value is specified during external table creation. For CSV format, the default value is the double quotation marks ("). For TEXT format, the default value is the backslash (\).
  - You can customize the ESCAPE value to a single character. For example, the following command sets the ESCAPE value to a backslash:

    ```
    FORMAT  ' csv ' ( ESCAPE  '\' )
    ```
  - You can also set ESCAPE to OFF to avoid escaping all characters automatically.
- The column contains single quotation marks or double quotation marks.
  - If you are using the TEXT format, you must add an ESCAPE character before the single quotation marks or double quotation marks. The default value is a backslash (\).
  - If you are using the CSV format, you must add an ESCAPE character before the single quotation marks or double quotation marks. The default value is double quotation marks ("). You also must add the double quotation marks at both the beginning and end of the column to enclose the column in the quotation marks.

Import date by using the COPY command

When you use \COPY statements to import data, the usage of delimiter is the same as that for using OSS to import data. The handling of special characters in the data is also similar.

The difference is that COPY statements and the `CREATE  EXTERNAL  TABLE` statement are slightly different in usage. For more information, see *Import data by using the COPY command*.