Alibaba Cloud AnalyticDB for PostgreSQL

ベストプラクティス

Document Version20190702

目次

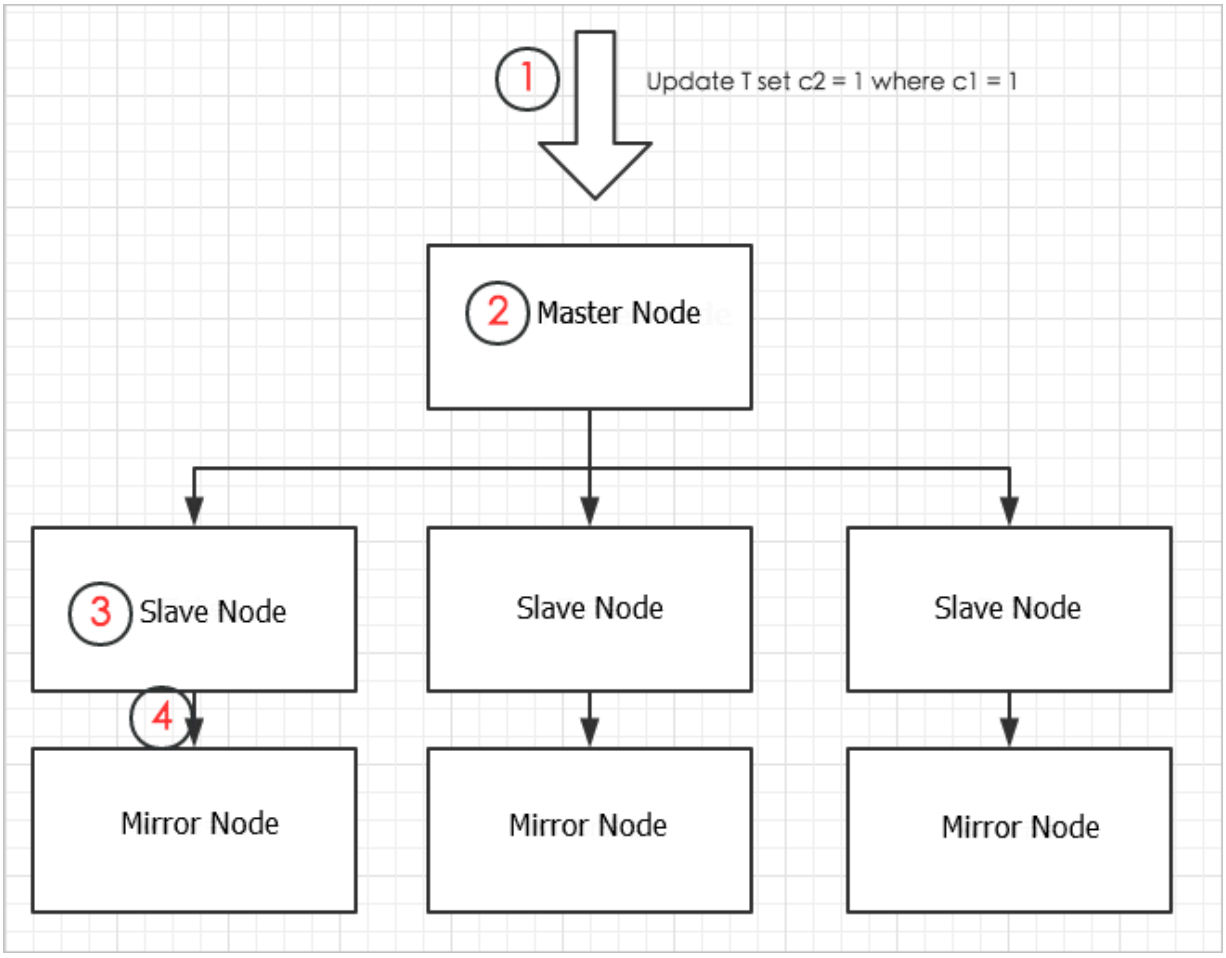
1 一括更新

マージとも呼ばれる更新は、AnalyticDB for PostgreSQL に最新のデータを更新することを示します。 更新されたデータが既に存在する場合は、古いバージョンが置き換えられます。 更新されたデータが存在しない場合は、データベースに挿入されます。 このようなデータのマージは、通常、オフラインで完了します。 たとえば、毎日 AnalyticDB for PostgreSQL にデータを更新するように設定できます。 あるユーザーは、遅延が分または秒レベルのリアルタイム更新が必要かもしれません。

ここでは、 AnalyticDB for PostgreSQL データをマージする方法とその原則について説明します。 さらに、一括操作を使用して複数のデータを更新する方法も学べます。

簡単なアップデート

データのマージは、データの変更、つまり更新、削除、挿入、またはコピー操作の実行に関するものです。 更新操作を例にとると、column-store テーブルの単一の行で記録を更新することです。 次の図は、AnalyticDB for PostgreSQL のデータ更新プロセスを示しています。



手順は次のとおりです。

1. ユーザーは、Update SQL リクエストをマスターノードに送信します。
2. マスターノードは、分散トランザクションを開始し、更新するテーブルをロックします (AnalyticDB for PostgreSQL は、同じテーブルへの同時更新を許可しません）。一致するスレーブノードに更新リクエストを配信します。
3. スレーブノードは、インデックスをスキャンして更新するデータを探し出し、データを更新します。 column-store テーブルの場合、古いデータ行を削除し、新しいデータ行をテーブルの末尾に書き込むことが更新のロジックです。 column-store テーブル内の更新されたデータページがメモリキャッシュに書き込まれ、対応するテーブルファイルの長さの変更 (データがテーブルの終わりに書き込まれるため、対応するテーブルファイルの長さが増えます) がログ (xlog ファイル) に記載されます。
4. 更新プロセスが終了する前に、メモリ内の更新されたデータページと xlog ファイルは、両方ともミラーノードに同期されます。 同期が完了すると、マスターノードは分散トランザクションを終了し、成功した実行に関するメッセージをユーザーに返します。

プロセス全体は長く、SQL 文の解析、トランザクションの配布、ロック、マスターノードとスレーブノード間の接続、スレーブノードとミラーノード間のデータとログの同期など、多くの操作が含まれています。 これらの操作はすべて CPU または I/O リソースを消費し、リクエストの応答が遅くなります。

したがって、 AnalyticDB for PostgreSQL の場合は、単一のデータ行への更新を避け、可能な限り一括操作を使用してデータを更新する以下のことを推奨します。

- 文の解析、ノード通信、およびデータ同期のオーバーヘッドを軽減するために、1 つの SQL 文を更新します。
- 不要なオーバーヘッドを避けるために、1 つのトランザクションを更新します。

一括更新

1 つの SQL 文を使用して複数の独立したデータ行を更新するには、次の手順に従います。

1. 対象テーブルの準備

更新するテーブルが target_table であるとします。 target_table は以下のように定義されます。

```
create  table  target_tab  le ( c1  int ,  c2  int ,  primary
key  ( c1 ));
```

```
insert into target_tab le select gen - USerate_se ries ( 1
, 10000000 );
```

対象テーブルは、通常非常に大きくなります。 target_table に 1,000万行のデータを挿入するとします。 target_table は、更新を容易にするために索引付けされています。 A プライマリーキーが定義され、その結果、ユニークなインデックスが含まれます。

2. ステージテーブルの準備

一括更新にはステージテーブル (この例ではsource_table) が必要です。 これは、データを更新するために作成された一時テーブルです。 target_table のデータを更新するには、最初に新しいデータを source_table に挿入し、*[COPY コマンド]*、*[OSS 外部テーブル]*、または target_table への別の手段で、新しいデータをインポートします。

次の例では、データがいくつか source_table に直接生成されています。

```
create table source_tab le ( c1 int , c2 int );
insert into source_tab le select gen - USerate_se ries ( 1
, 100 ), gen - USerate_se ries ( 1 , 100 );
```

3. 一括更新

source_table データの準備が完了したら、 `update set … from … where ..`文を実行します。

注 :

インデックスを最大限に活用するには、更新操作の前に `set optimizer = on` を使用して ORCA オプティマイザを起動します。 ORCA オプティマイザが起動していない場合は、 `set enable_nes tloop = on` を実行してインデックスを使用できます。

```
set optimizer = on ;
target_tab le を source_tab le . c1 から source_tab le . c2 に設定します。
```

更新操作のクエリプランは次のとおりです。

```
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 .. 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
```

このプランから、 AnalyticDB for PostgreSQL はインデックスを使用します。 しかし、 source_table にさらにデータを追加すると、オプティマイザは、Nest Loop 関連メソッドとインデックススキャンを使用する方がインデックスを削除するよりは効率的ではないと判断する可能性があります。 結果として、ハッシュ関連方法とテーブルスキャンを使用して実行するかもしれません。 たとえば、

```
 postgres =>  insert   into   source_tab  le   select   gen -
 USerate_se  ries ( 1 ,  1000 ),  gen - USerate_se  ries ( 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 )
```

ここで説明している一括更新アプローチでは、SQLのコンパイル、ノード間通信、トランザクション、およびその他のオーバーヘッドが削減され、データ更新パフォーマンスが大幅に向上し、リソース消費が削減されます。

一括削除

削除操作に関して、一括更新に使用されるのと同様のステージテーブルを使用し、次の削除コマンドと "Using" 構文でデータを一括削除できます。

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

一括削除操作でもインデックスが使用されます。

```
explain delete from target_tab le using source_tab le
where target_tab le . c1 = source_tab le . c1 ;
                                        QUERY PLAN
------------------------------------------------
 ------------------------------------------------
  Delete ( slice0 ; segmen - USts : 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
```

削除と挿入を使ってデータをマージする

データをマージするには、まずデータをステージテーブルにマージする必要があります。 マージするデータが既にターゲットテーブルに存在することが事前に分かっている場合は、更新文を使用してデータをマージします。 しかし、ほとんどの場合、マージするデータのある部分は既にターゲットテーブルに存在し、一部が新しくなっており、ターゲットテーブルに一致するレコードが存在していません。 この場合、一括削除と一括挿入の組み合わせが使用できます。 サンプルコードは次のとおりです。

```
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 ;
```

Values( ) を使用してリアルタイムでデータを更新する

ステージテーブルを使用するには、そのライフサイクルを維持する必要があります。 一部のユーザーは、リアルタイムに一括で PostgreSQL に関して HybridDB にデータを更新したいと考えています、つまり、継続的にデータを同期したり PostgreSQL 用に HybridDB にデータをマージしたいと考えています。

前述の方法を使用する場合は、ステージテーブルを繰り返し作成および削除 (または切り捨て) する必要があります。 実際、テーブルを維持するために苦労することなく、Values 式を使用して

ステージテーブルと同様の効果を得ることができます。 まず、データをスプライスして Values 式に更新し、次の方法を使用して、更新コマンドまたは削除コマンドを実行します。

```
 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
```

注：

どちらも「オプティマイザ= on 」と「 enable_nes tloop = on 」を設定し、インデックスを使用するクエリプランを生成します。ただし、複数の索引フィールドやパーティションテーブルが含まれるような複雑な場合は、ORCA オプティマイザを使用して索引を一致させる必要があります。

2 AnalyticDB for PostgreSQL のパフォーマンスを改善する

AnalyticDB for PostgreSQL は、Greenplum Databaseに基づいて開発され、Alibaba Cloud の細かい拡張機能でさらに強力になっています。 MPP (Massively Parallel Processing) データウェアハウスサービスを提供するための、複数の *[グループ]*で構成される分散クラウドデータベースです。

ここでは、AnalyticDB for PostgreSQL を使用するためのベストプラクティスを紹介します。AnalyticDB for PostgreSQL のパフォーマンスを向上させ、インポートプロセスをスピードアップし、コストを削減するために、以下の方法から選択することを推奨します。

圧縮カラムストレージの使用する

更新頻度が低くフィールド数が多いテーブルの場合は、圧縮カラムストレージの使用を推奨します。 この方法では、パフォーマンスを保証しながらも圧縮率を 3 倍まで増加させます。そして、インポート速度も通常より速くなります。

例えば、集計ステートメントに `WITH 列 ( APPENDONLY = true 、 ORIENTATIO N =列、 COMPRESSTY PE = zlib 、 COMPRESSLE VEL = 3 、 BLOCKSIZE = 1048576 )` を追加して、圧縮カラムストアテーブルを作成します

具体的な構文に関しては、*「CREATE TABLE」*をご参照ください。

Nested Loop JOIN を使用する

Nested Loop JOIN はデフォルトでは、 AnalyticDB for PostgreSQL インスタンスで有効ではありません。 少量のデータを含む、または返すのみのクエリでは、パフォーマンスが最適でない可能性があります。

次の SQL ステートメントを例としてご参照ください。

```
select * from T1 join T2 on T1 . c1 = T2 . c1 where
  T1 . c2 >= &# 039 ; 230769548 &# 039 ; and T1 . c2 < &# 039 ;
230769549 &# 039 ; limit 100 ;
```

この例では、T1 テーブルと T2 テーブルのサイズはどちらも大きくなっています。 選択条件 `T1 ( T1 . c2 >= ‘ 230769548 ’` and `T1 . c2 <‘ 23432442 ’)` が大多数のデータレコードをフィルタリングし、LIMIT 節を含んでいます。

その結果、クエリには実際には合計データサイズのわずかな部分のみが含まれます。この場合は、Nested Loop JOIN メソッドが最適です。

You can 次の SET コマンドを実行して、Nested Loop JOIN をアクティブにすることができます。

```
 show   enable_nes  tloop  ;
  enable_nes  tloop
-----------------
  off
 SET   en - USable_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  >= &# 039 ; 230769548 &# 039 ;  and   T1 . c2  <
 &# 039 ; 23432442 &# 039 ;  limit   100 ;
                                                  クエリプラン
-------------------------------------------------
 -----------------------------------------------
  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  >= &# 039 ; 230769548 &# 039 ;:: 
 bpchar )  AND  ( c2  < &# 039 ; 230769549 &# 039 ;:: bpchar ))
        ->   Index   Scan   using   T2   on   c1   ( cost = 0 . 14 ..
 8 . 15   rows = 1   width = 6582 )
               Index   Cond : ( ( c1 ) ::  text  = ( T1 . c1 ) ::
 text )
```

このクエリプランから、T1 および T2 テーブルは Nested Loop JOIN を採用し、最適なパフォーマンスを実現します。

ORCA オプティマイザの使用方法

AnalyticDB for PostgreSQL は ORCA オプティマイザをサポートしています。 複雑な SQL ステートメントを実行しても結果が不十分な場合、ORCA オプティマイザを試します。

以下の SET コマンドをデータベース接続に実行し、ORCA を有効にします。

注：

SET コマンドは接続レベルで動作し、同じ接続内でのみ有効です。 新たに接続するには、SET コマンドを再度実行して、ORCA を有効にする必要があります。

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

前述の例では、ORCA を有効にする前後で、クエリプランで EXPLAIN コマンドが使用されていることがわかります。 この方法では、ORCA が実際に SQL クエリプランを変更したかどうかを確認できます。

圧縮方法の使用方法

HybridDB for PostgreSQL は zlib と RLE の 2 つの圧縮方法をサポートしています。

- RLE は同じデータ値が物理的に連続で保存されているシナリオで威力を発揮します。
- Zlib は他のシナリオにも適用可能です。

圧縮方法は、フィールドレベルまたはテーブルレベルで指定できます。 詳細に関しては、「*CREATE TABLE*」をご参照ください。

その他の数値型の使用方法

クエリに COUNT の固有値統計演算（DISTINCT）が含まれている場合は、統計フィールドに文字列や数値タイプを使用せずに、他の数値タイプ（整数タイプなど）を使用することを推奨します。この方法により、パフォーマンスを数倍向上できます。

3 定期メンテナンスタスク

更新操作 (INSERT VALUES、UPDATE、DELETE、および ALTER TABLE ADD COLUMN) をシステム上で実行した場合、不要なデータが システムテーブルや更新データテーブルに残ってしまう可能性があります。 不要データはシステムパフォーマンスを低下させ、 ディスクスペースの多くを占有します。 このようなデータは、以下の方法で 定期的に消去することを推奨します。

テーブルをロックしないで不要データを消去する

テーブルをロックすることなく、不要データを消去することができます。 その方法は次のとおりです。

- コマンド : 全てのデータベースに接続し、データベースオーナーとしてログインします。 そして、 `VACUUM` コマンドを実行します。
- 頻度: 少なくとも 1 日 1 回。
  - データがリアルタイムで更新されている場合 (INSERT VALUES、UPDATE、 DELETE 操作が連続的に実行されている場合)、 `VACUUM` コマンドを 2 時間ごとに実行することを推奨します。
  - データが一日一回、一括更新される場合、 一括更新が行われた後に、そのコマンドを実行できます。
- システムへの影響: テーブルはロックされず、テーブルを正常に読み書きすることができます。しかし、CPU と I/O の使用量が増加するため、クエリのパフォーマンスに影響する可能性があります。
- たとえば、次のような Linux Shell スクリプトファイルに 定期的な crontab タスクを実行します。

```
#!/ 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  & quot ; copy  ( select
 datname   from   pg_stat_da  tabase )  to   stdout & quot ;`
 for   db   in  $ dblist  ;  do
    # skip   the   system   databases
    # skip   system   databases
         continue
     fi
     echo   processing  $ db
    # vacuum   all   tables  ( catalog   tables / user   tables )
     psql  - d  $  db  - e  - a  - c  & quot ; VACUUM ;& quot ;
```

```
done
```

メンテナンス中の不要データの消去

サービスが中断した場合、サービスのメンテナンス中にすべての不要データを消去できます。 その方法は次のとおりです。

- コマンド: すべてのデータベースに接続し、 データベースオーナーとしてログインします。すべての操作に対してオーナー権限を持つ必要があります。
  1. `REINDEX SYSTEM < database name >` コマンドを実行します。
  2. `VACUUM FULL < table name >`、`REINDEX TABLE < table name >` の順でコマンドをすべてのデータテーブル (非システムテーブル) において実行します。
- 頻度：少なくとも週 1 回。 毎日データが更新されるのであれば、 1 日 1 回はこのコマンドを実行します。
- システムへの影響: VACCUM FULL あるいは REINDEX は、 テーブルをロックし読み書きできないようにします。 これにより、CPU への負荷や I/O の利用が増える可能性があります。
- たとえば、次のような Linux Shell スクリプトファイルに 定期的な crontab タスクを実行します。

```
#!/ 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  & quot ; copy  ( select
 datname   from   pg_stat_da  tabase )  to   stdout & quot ;`
 for   db   in  $ dblist  ;  do
    # skip   system   databases
    # skip   system   databases
        continue
     fi
     echo   processing   db  "$ db "
    # do   a   normal   vacuum
     psql  - d  $  db  - e  - a  - c  & quot ; VACUUM ;& quot ;
    # reindex   system   tables   firstly
     psql  - d  $  db  - e  - a  - c  & quot ; REINDEX   SYSTEM  $  db
 ;& quot ;
    # 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   paren - USt   tables
     psql  - d  $ db  - c  " copy  ( select  '\"'|| tables .
 schemaname ||'\".' || '\"'|| tables . tablen - USame ||'\"'  from
  ( select   nspname   as   schemaname ,  relname   as   tablen -
 USame   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 ,  tablen - USame )  left   join
```

```
pg_catalog . pg_partiti  ons   on   pg_partiti  ons . partitions
chemaname = tables . schemaname   and   pg_partiti  ons . partitiont
ablen - USame = tables . tablen - USame   where   pg_partiti  ons .
partitiont  ablen - USame   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 "
        echo   processing   table  "$ line "
        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 インポートにおける特殊文字

AnalyticDB for PostgreSQL は、複数のデータのインポート方法をサポートします。

- *OSS* を使用したデータの並行インポートおよびエクスポート
- *MySQL* からのデータインポート
- *PostgreSQL* からのデータインポート
- *COPY* コマンドを用いたデータインポート
- *Data Integration* 使用したデータの同期

特殊文字は、データのインポート中にインポートエラーを引き起こすことがよくあります。 ここでは、インポートされたデータ内の特殊文字を事前処理し、それによって発生する問題を排除する方法について説明します。

前述のインポート方法では、MySQL および PostgreSQL からデータをインポートするツールが、自動的に特殊文字をエスケープしてパッケージ化するため、 追加設定なしでツールを直接使用できます。 次の内容では、OSS と COPY コマンドを使用してデータをインポートする方法と、特殊文字の処理方法について説明します。

OSS を使用したデータの並行インポート

データのインポート時には、ファイル内のすべての行がタプルと見なされることが多く、 各列のデータは行ごとに区切り文字で分割されます。 以下 では、区切り記号の使用法と制約、および各列の特殊文字の処理方法を紹介しています。

区切り記号

OSS 外部テーブルを作成する構文では、次の ように FORMAT 句の後に区切り記号を指定します。

```
FORMAT  &# 039 ; TEXT &# 039 ; ( DELIMITER  &# 039 ;,&# 039 ;)
```

- `FORMAT ' TEXT '` に対する `DELIMITER` は、 デフォルトでは `\ t` です。
- `FORMAT ' CSV '` に対する `DELIMITER` は、 デフォルトでは `,` です。

外部テーブルを作成する構文での規定に従い、カスタマイズされた区切り記号が 次の制約を満たすという前提であれば、独自の区切記号を定義できます。

- それらは、ASCII 文字でなければなりません。 漢字や、2 つ以上の ASCII 文字は使用できません。
- `\ n` および `\ r` はサポートされていません。

- `\ n` および `\ r` を除くエスケープ文字は、 "E" あるいは "e" を文字の前に追加することでサポートされます。
- エスケープ文字 `\ t` に "E" をつけないものも サポートされます。
- TEXT 形式では、DELIMITER を OFF に設定し、単一列の 外部テーブルを使用できます。

データを正しく読み取るには、OSS ファイルでは、指定した区切り文字に厳密に従う必要があります。

データ内の特殊文字

データのインポート中に、特殊文字が使用されるケースには、 次のものがあります。

- 列に区切り文字と同じ文字が含まれる場合
  - TEXT 形式を使う場合、各 DELIMITER の前に ESCAPE 文字をつけなければなりません。 ESCAPE 文字は、外部テーブルを作成するときに次のコマンドを使用して指定します。 デフォルト値は、バックスラッシュ (\) です。

    ```
    FORMAT  &# 039 ; TEXT &# 039 ; ( ESCAPE  &# 039 ;\&# 039 ;)
    ```

  - CSV 形式を使用する場合には、 二重引用符 (") を各 DELIMTER 前に追加する必要があります。
- 列に漢字が含まれる場合。 OSS 外部テーブルは、 漢字データをサポートします。 ただし、表示が正しいことを確認するには、 作成した外部テーブルを次のようにエンコードする必要があります。

  ```
  ENCODING  ' UTF8 '
  ```

- 列に Null データが含まれる場合。 Null 値を文字に一致させるように設定し、 データのインポート中に指定された文字を Null に置き換えます。CSV 形式の場合、デフォルト値は 引用符のない Null 値です。 TEXT 形式の場合、デフォルト値は "\N" です。 The 次のコマンドは、スペースを Null にマップします。 列がスペースの場合、 列の値は OSS ファイルからインポートされたデータの Null になります。

  ```
  FORMAT  &# 039 ; text &# 039 ; ( null  &# 039 ; &# 039 ; )
  ```

- 列にエスケープ文字が含まれる場合。 エスケープ文字の前に "ESCAPE" を追加することができます。 ESCAPE 値は、外部テーブルの作成時に指定されます。 CSV 形式の場合、デフォル

ト値は二重引用符（"）です。 TEXT 形式の場合、デフォルト値はバックスラッシュ（\）です。

- ESCAPE の値を 1 文字にカスタマイズすることができます。 たとえば、次のコマンドは、ESCAPE 値を バックスラッシュに設定します。

```
FORMAT  &# 039 ; csv &# 039 ; ( ESCAPE  &# 039 ;\&# 039 ;)
```

- すべての文字を自動的にエスケープしないように、ESCAPE を OFF に設定することもできます。

・ 列に、単一引用符 (') または二重引用符 (") が含まれる場合。

- TEXT 形式を用いる場合、単一引用符または二重引用符の前に ESCAPE 文字を追加する必要があります。 デフォルト値はバックスラッシュ (\) です。
- CSV 形式の場合、単一引用符か二重引用符の前に ESCAPE 文字を追加する必要があります。 デフォルト値は二重引用符 (") です。 また、列を引用符で囲むには、 列の最初と最後の両方に二重引用符を追加する必要があります。

日付を COPY コマンドを使用してインポート

データをインポートするために、 \COPY 文を用いる場合、 区切り文字は OSS の場合と同じです。 データ内の特殊文字の扱いも 同様です。

COPY 文と `CREATE EXTERNAL TABLE` 文は使用上わずかな差異があります。 詳細は、「*COPY* コマンドを用いた データインポート」をご参照ください。

5 英語版なし