HighGo Postgres Server (HG-PGSQL) User Guide

Suitable for

HG-PGSQL v1.X

Last updated on

May 19, 2020

HighGo Software Inc. ™
10318 Whalley Blvd #1 Surrey, BC, V3T 4H4, Canada
T: +1 (604) 781-6749 E: contact@highgo.ca W: www.highgo.ca

Table of Contents

1 Introduction

1.1 What's New

HG-PGSQL is built on top of already feature-rich PostgreSQL 12 and provides additional useful features as follows:

- Enhanced Partition Creation Syntax
- Shard Management
- Parallel Backup

1.2 Typographical Conventions

Certain typographical conventions are used in this document to distinguish various commands, statements, programs, examples ...etc. This section provides a summary of these conventions.

Italic font	**Italic is used in sentences that required extra attention. Normally used in "Warning" or "Important" sections**
`Fixed-width font`	This font is used on user commands, inputs, SQL column names, programming keywords ...etc. **For example:** `SELECT pg_reload_conf();`
`Italic fixed-width font`	This font is used on terms in which the user must substitute a value in actual usage: **For example:** `DELETE FROM` *`table_name;`*
Dollar sign - $	Dollar sign represents the start of a user or SQL commands, that the user can issue on a command line terminal. The dollar sign is commonly used with fixed-width font **For example:** `$ SELECT * from test_table;`
Vertical pipe - |	Vertical pipe denotes a choice between the terms on either side of the pipe. It is commonly used with square brackets or braces to separate two or more alternatives choices.
Square brackets - [ ]	Square brackets denote that one or none of the enclosed terms may be substituted. Normally a vertical pipe is used within the square brackets to denote choices. **For example:** [a | b] means to choose one of "a" or "b" or none at all.
Braces - { }	Braces denote that exactly one of the enclosed terms

	must be specified. Normally a vertical pipe is used within the braces to denote choices. **For example:** { a | b | c } means exactly “a”, “b” or “c” must be specified.
Ellipses - ...	Ellipses denote that the preceding terms may be repeated. Normally a vertical pipe is used together to denote choices. **For example:** [a | b] ... means that you may have the sequence, “a a a b b”.

2 Product Installation

2.1 End User License Agreement

Make sure you have read and agreed to the End User License Agreement (EULA) from the link below before installing and using HighGo Postgres Server.

https://yum.highgo.ca/#license

2.2 Supported Platforms

HighGo Postgres Server installation is supported on the following platforms

- CentOS (X86_64) 6.x
- CentOS (X86_64) 7.x

2.3 Installation Prerequisite

Prior to installing HG-PGSQL and its supporting components, you will need to install the HighGo yum repository entry on your system so that the *yum* utility is able to download the desired HG-PGSQL packages

```
$ yum -y install https://yum.highgo.ca/dists/rpms/repo/highgo-release-1.0-2.noarch.rpm
```

Upon successful installation, a new HighGo yum repository entry will be created at:

```
/etc/yum.repos.d/highgo.repo
```

And make sure the GPG key for HighGo is also created at:

```
/etc/pki/rpm-gpg/HIGHGO-SOFTWARE-GPG-KEY
```

Alternatively, you may also follow the link below to download the HG-PGSQL RPM packages from Highgo.ca for local installation.

https://yum.highgo.ca/

2.4 Detailed Installation and Quick Start Guide

The detailed installation and quick start guide for HG-PGSQL product can be found at the highgo.ca product page or visit the link below:

https://www.highgo.ca/products/highgo-postgresql-server

3 Partition Syntax Enhancement

3.1 Overview

HG-PGSQL greatly simplifies the table partition creation syntax by allowing the user to create partitioned table along with its partitions and sub-partitions in one consolidated SQL statement.

Partition syntax enhancement is particularly useful in a real-world scenario where a user is required to create a complex partitioned table structure. This feature greatly reduces the syntax complexity and reduce the number of SQL statement, therefore reducing the possibility of error.

The new table partition syntax follows this general rule below and please note that this rule contains every possible usage in one large syntax. We will break down each usage in the sections that follow.

```
CREATE TABLE table_name ( column_name TYPE column_constraint )
PARTITION BY { RANGE | LIST | HASH } ( column_name )
(
    /* Partition by Range */
    PARTITION partition_name FOR VALUES FROM ( lower_bound ) TO ( upper_bound ),

    /* Partition by Hash */
    PARTITION partition_name FOR VALUES WITH ( hash_partition_definition ),

    /* Partition by List */
    PARTITION partition_name FOR VALUES IN ( list_partition_definition ),

    /* Default Partition */
    PARTITION partition_name DEFAULT,

    /* Default Partition with Constraint */
    PARTITION partition_name ( CONSTRAINT constrain_name partition_constraint )
        DEFAULT,

    /* Sub Partition Extended from Partition by Range */
    PARTITION partition_name FOR VALUES FROM ( lower_bound ) TO ( upper_bound )
    PARTITION BY { RANGE | LIST | HASH }( column_name )
    (
        /* Sub Partition by Range
        PARTITION sub_partition_name FOR VALUES FROM ( lower_bound ) TO
            ( upper_bound ),

        /* Sub Partition by List
        PARTITION sub_partition_name FOR VALUES IN ( list_partition_definition ),

        /* Sub Partition by Hash
        PARTITION sub_partition_name FOR VALUES WITH ( hash_partition_definition)

        /* more sub partitions follow ...*/
    )
    /* more partition definitions follow ...*/
);
```

Let’s examine the new syntax in more detail:

- First, the user defines a table and list of columns using the `CREATE TABLE` clause following the standard PostgreSQL syntax. Refer to the following link for more detailed information on the `CREATE TABLE` clause.
  http://www.postgresqltutorial.com/postgresql-create-table/

- Next, the user defines the new table as partitioned table using clause `PARTITION BY { RANGE | LIST | HASH } ( column_name )` and selects the `column name` that will be used to partition the data entries. The `PARTITION BY` clause also follows the standard PostgreSQL syntax and partition type can be one of the following: `RANGE`, `LIST` or `HASH`. Please refers to the following for more detailed information of `PARTITION BY` clause.

  https://www.postgresql.org/docs/current/ddl-partitioning.html

- Then, the new syntax unique to HG-PGSQL follows. The user can optionally specifies a number of partition definitions using one or more `PARTITION partition_name` clauses enclosed in a bracket ( ) and placed after the `( column_name )`. Each line of `PARTITION partition_name` clause is separated by comma and depending on the partition type (range, list or hash), different clauses are used to specify the boundary condition for partitions.
  - For `PARTITION BY RANGE`, the user will use `FOR VALUES FROM ( lower_bound ) TO ( upper_bound )` to define a value range.
  - For `PARTITION BY LIST`, the user will use `FOR VALUES IN ( list_partition_definition )` to define a list of values.
  - For `PARTITION BY HASH`, the user will use `FOR VALUES WITH ( hash_partition_definition)` to define a hash condition.
  - For default partition type, the user will simply use the `DEFAULT` clause as partition condition
  - The clauses defining the range, list and hash definitions also follow the standard PostgreSQL syntax and details can be found in the above link.

- Constraints can be specified with "`CONSTRAINT constraint_name partition_constraint`" clause enclosed in brackets ( ) and placed right after "`PARTITION partition_name`" clause. The values for `partition_constraint` follows the standard PostgreSQL syntax involving clauses such as `UNIQUE`, `CHECK`, and `PRIMARY KEY` …etc. Constraint details can be found here:

  https://www.tutorialspoint.com/postgresql/postgresql_constraints.htm

- Sub partitions can be specified by placing another "`PARTITION BY { RANGE | LIST | HASH }( column_name )`" clause at the end of partition definition, following a bracket ( ) containing additional partition boundary conditions using the same `PARTITION sub_partition_name` clauses separated by comma.

3.2 Partition by Range

Using standard PostgreSQL syntax, to create a partitioned table with 3 partition definitions, a user is required to execute at least 4 SQL statements as illustrated below.

```
$ CREATE TABLE prt_com1 ( a INT ) PARTITION BY RANGE ( a );
CREATE TABLE
```

```
$ CREATE TABLE prt_com1_p1 PARTITION OF prt_com1 FOR VALUES FROM (1) TO (10);
CREATE TABLE
$ CREATE TABLE prt_com1_p2 PARTITION OF prt_com1 FOR VALUES FROM (10) TO (20);
CREATE TABLE
$ CREATE TABLE prt_com1_p3 PARTITION OF prt_com1 DEFAULT;
CREATE TABLE
```

With HG-PGSQL partition syntax enhancement, partition by range creation can be shortened with this general rule:

```
$ CREATE TABLE table_name ( column_name TYPE ) PARTITION BY RANGE ( column_name )
  (
      PARTITION partition_name_1 FOR VALUES FROM (lower_bound) TO (upper_bound),
      PARTITION partition_name_2 FOR VALUES FROM (lower_bound) TO (upper_bound),
      PARTITION partition_name_3 DEFAULT

      /* more partition definitions follow ...*/
  );
```

For example:

```
$ CREATE TABLE prt_com1 ( a INT ) PARTITION BY RANGE ( a )
  (
      PARTITION prt_com1_p1 FOR VALUES FROM (1) TO (10),
      PARTITION prt_com1_p2 FOR VALUES FROM (10) TO (20),
      PARTITION prt_com1_p3 DEFAULT
  );
CREATE TABLE
```

3.3 Partition by List

Partition by List can be defined using the following general rule.

```
$ CREATE TABLE table_name ( column_name TYPE ) PARTITION BY LIST ( column_name )
  (
      PARTITION partition_name_1 FOR VALUES IN ( list_partition_definition ),
      PARTITION partition_name_2 FOR VALUES IN ( list_partition_definition ),
      PARTITION partition_name_3 DEFAULT

      /* more partition definitions follow */
  );
```

For example:

```
$ CREATE TABLE prt_com2 ( a INT ) PARTITION BY LIST ( a )
  (
      PARTITION prt_com2_p1 FOR VALUES IN (1, 3, 5 ,7 ,9),
      PARTITION prt_com2_p2 FOR VALUES IN (2, 4, 6, 8,10),
      PARTITION prt_com2_p3 DEFAULT
  );
CREATE TABLE
```

3.4 Partition by Hash

Partition by Hash can be defined using the following general rule.

```
$ CREATE TABLE table_name ( column_name TYPE ) PARTITION BY HASH ( column_name )
  (
      PARTITION partition_name_1 FOR VALUES WITH ( hash_partition_definition ),
      PARTITION partition_name_2 FOR VALUES WITH ( hash_partition_definition )

      /* more partition definitions follow */
  );
```

For example:

```
$ CREATE TABLE prt_com3 ( a INT ) PARTITION BY HASH ( a )
  (
      PARTITION prt_com3_p1 FOR VALUES WITH ( MODULUS 3, REMAINDER 0),
      PARTITION prt_com3_p2 FOR VALUES WITH ( MODULUS 3, REMAINDER 1),
      PARTITION prt_com3_p3 FOR VALUES WITH ( MODULUS 3, REMAINDER 2)
  );
CREATE TABLE
```

3.5 Sub Partition

Sub-partitioning is an act of dividing a partition further into more partitions and it can be declared by placing another block of `PARTITION BY [ RANGE | LIST | HASH ]` clause right after a partition definition.

The following general rule illustrates the syntax to define a partition by range, and then sub-partition it with range, list and hash partitions. The partition type does not limit the ability to create sub-partitions and you may define a partition by list or hash and then sub-partition it with another range, list and hash partitions. It is also possible to divide a sub-partition further by placing another block of `PARTITION BY [ RANGE | LIST | HASH ]` clause right after a sub-partition definition.

```
$ CREATE TABLE table_name ( column_name TYPE ) PARTITION BY RANGE ( column_name )
  (
      PARTITION partition_name_1 FOR VALUES FROM (lower_bound) TO (upper_bound)
      PARTITION BY RANGE( column_name )
      (
          /* Sub-Partition by RANGE */
          PARTITION sub_partition_name_1 FOR VALUES FROM (lower_bound) TO
              (upper_bound),
          PARTITION sub_partition_name_2 FOR VALUES FROM (lower_bound) TO
              (upper_bound),
          PARTITION sub_partition_name_3 DEFAULT

          /* more sub partition definitions follow ...*/
      ),
      PARTITION partition_name_2 FOR VALUES FROM (lower_bound) TO (upper_bound)
      PARTITION BY LIST( column_name )
      (
          /* Sub-Partition by LIST */
          PARTITION sub_partition_name_4 FOR VALUES IN
              (list_partition_definition),
          PARTITION sub_partition_name_5 FOR VALUES IN
              (list_partition_definition),
          PARTITION sub_partition_name_6 DEFAULT
          /* more sub partition definitions follow ...*/
      ),
      PARTITION partition_name_3 FOR VALUES FROM (lower_bound) TO (upper_bound)
      PARTITION BY HASH( column_name )
      (
          /* Sub-Partition by HASH */
          PARTITION sub_partition_name_7 FOR VALUES WITH
              (hash_partition_definition),
          PARTITION sub_partition_name_8 FOR VALUES WITH
              (hash_partition_definition)

          /* more sub partition definitions follow ...*/
      ),
      PARTITION partition_name_10 DEFAULT

      /* more partition definitions follow ...*/

  );
```

For example:

```
$ CREATE TABLE prt_com4(a int, b int, c int)
PARTITION BY RANGE( a )
(
      PARTITION prt_com4_p1 FOR VALUES FROM (0) TO (100)
      PARTITION BY RANGE( a )
      (
              PARTITION prt_com4_p1_1 FOR VALUES FROM (0) TO (10),
              PARTITION prt_com4_p1_2 FOR VALUES FROM (10) TO (20),
              PARTITION prt_com4_p1_3 DEFAULT
      ),
      PARTITION prt_com4_p2 FOR VALUES FROM (100) TO (200)
      PARTITION BY LIST( b )
      (
              PARTITION prt_com4_p2_1 FOR VALUES IN (1,2,3,4),
              PARTITION prt_com4_p2_2 FOR VALUES IN (5,6,7,8),
              PARTITION prt_com4_p2_3 DEFAULT
      ),
      PARTITION prt_com4_p3 DEFAULT
      PARTITION BY HASH( c )
      (
              PARTITION prt_com4_p3_1 FOR VALUES WITH(MODULUS 3, REMAINDER 0),
              PARTITION prt_com4_p3_2 FOR VALUES WITH(MODULUS 3, REMAINDER 1),
              PARTITION prt_com4_p3_3 FOR VALUES WITH(MODULUS 3, REMAINDER 2)
      )
);
CREATE TABLE
```

Defining all the partitions and sub-partitions in one SQL statement may introduce a very large and lengthy statement, but with proper indentation and style, user is able to better visualize the hierarchical relationship between the partitions, sub-partitions and their respective conditions. This, in fact, reduces the possibility of human errors in comparison to the original statement by statement partition creation syntax.

3.6 Partition Constraint

Constraints are the rules enforced on data columns on a table or a partition. These are used to prevent invalid data from entering into the database.

Constraints can be specified on a partition with the new syntax by enclosing them in brackets after the partition name declaration. The partition constraint can be defined following this general rule below and please note that the rule uses partition by range as example. The same rule applies to other types of partitions.

```
$ CREATE TABLE table_name ( column_name TYPE ) PARTITION BY RANGE ( column_name )
(
    PARTITION partition_name_1 ( CONSTRAINT constrain_name partition_constraint )
        FOR VALUES FROM ( lower_bound ) TO ( upper_bound ),
    PARTITION partition_name_2 ( CONSTRAINT constrain_name partition_constraint )
        FOR VALUES FROM ( lower_bound ) TO ( upper_bound ),
    PARTITION partition_name_3 ( CONSTRAINT constrain_name partition_constraint )
        DEFAULT

    /* more partition definitions follow */
);
```

Please note that `partition_constraint` can be defined according to standard PostgreSQL syntax involving common clauses such as “`NOT NULL`”, “`UNIQUE`”, “`CHECK`”…etc. Visit below for more information regarding constraint definition.

https://www.tutorialspoint.com/postgresql/postgresql_constraints.htm

The constraints are only created at the local server and not in the foreign server.

Consider the example below that includes partition constrains using different clauses while incorporating the new sub partition creation syntax.

```
$ CREATE TABLE f_prt_com5
(
    a int DEFAULT 10,
    b int,
    c VARCHAR NOT NULL DEFAULT ‘def val parent’
)
PARTITION BY RANGE(a)
(
    PARTITION f_prt_com5_p1 (CONSTRAINT con_f_prt_com5_p1 CHECK ( a != 19 ))
        FOR VALUES FROM (0) TO (100)
    PARTITION BY RANGE(a)
    (
        PARTITION f_prt_com5_p1_1 (CONSTRAINT con_f_prt_com5_p1_1 CHECK ( a != 9 ))
            FOR VALUES FROM (0) TO (10),
        PARTITION f_prt_com5_p1_2 FOR VALUES FROM (10) TO (20),
        PARTITION f_prt_com5_p1_3 (CONSTRAINT con_f_prt_com5_p1_3 CHECK (a != 90 ))
            DEFAULT
    ),
    PARTITION f_prt_com5_p2 FOR VALUES FROM (100) TO (200)
    PARTITION BY LIST(b)
    (
        PARTITION f_prt_com5_p2_1 (UNIQUE(a)) FOR VALUES IN (1,3,5,7),
        PARTITION f_prt_com5_p2_2 (UNIQUE(b)) FOR VALUES IN (2,4,6,8),
        PARTITION f_prt_com5_p2_3 (UNIQUE(c)) DEFAULT
    ),
    PARTITION f_prt_com5_p3 DEFAULT
    PARTITION BY HASH(c)
    (
        PARTITION f_prt_com5_p3_1 (CONSTRAINT PK_f_prt_com5_p3_1 PRIMARY KEY(a))
            FOR VALUES WITH (MODULUS 3, REMAINDER 0),
        PARTITION f_prt_com5_p3_2 (CONSTRAINT FK_f_prt_com5_p3_2 FOREIGN KEY(a)
            REFERENCES f_prt_com5_p2_2(b)) FOR VALUES WITH(MODULUS 3, REMAINDER 1),
        PARTITION f_prt_com5_p3_3 FOR VALUES WITH (MODULUS 3, REMAINDER 2)
    )
);
CREATE TABLE
```

4 Shard Management

4.1 Overview

Sharding is an act of partitioning a table over multiple database server instances in a distributed database environment whereas partitioning refers to dividing a table on the same database server. Sharding is also known as horizontal partition as it partitions the data horizontally based on a sharded key. This is analogous to how a table is partitioned using a partition key. In case of sharding, the data is partitioned across multiple instances using the sharded key. A table created on a foreign server is normally referred to as a "shard" and is normally accessed via a Foreign Data Wrapper (FDW) handler (ex. **postgres_fdw** extension) from the local database server.

HG-PGSQL introduces a new optional clause `WITH PUSHDOWN` that can be used in the creation of foreign partition tables. The clause provides an ability to automatically create foreign partition table on the foreign servers instead of having the user to do it manually. In standard PostgreSQL server, on contrary, the user needs to manually create the partition table on the foreign server as well as on the local one.

In addition to the new `WITH PUSHDOWN` clause, HG-PGSQL includes another optional clause `INCLUDE REMOTE` that can be used with the existing `DROP FOREIGN TABLE` clause to automatically drop the tables in the foreign servers instead of having the user to drop it manually. In standard PostgreSQL server, the user still needs to manually drop the partitioned tables on the foreign server as well as on the local one.

This feature is built in **postgres_fdw** extension in HG-PGSQL and can be extended to other FDWs. This is a useful feature designed to automate the creation and deletion of sharded tables.

4.2 Syntax and Placement of WITH PUSHDOWN Clause

`WITH PUSHDOWN` is an optional clause that can be added at the end of an existing `CREATE FOREIGN TABLE` SQL statement to trigger the automatic table creation on foreign servers. This clause can be used together with the creation of foreign tables and partitions and cannot be used with `ALTER`, `UPDATE`, `DROP` or `INSERT` clauses. The following sections outline different use cases for the `WITH PUSHDOWN`.

4.2.1 Foreign Table Creation

Consider this general rule below when using `WITH PUSHDOWN` in the end of foreign table creation

```
CREATE FOREIGN TABLE
    IF NOT EXISTS table_name ( column_name TYPE ) SERVER server_name
    OPTIONS ( options ) WITH PUSHDOWN;
```

Please note that `WITH PUSHDOWN` must be placed towards the end of the statement; it cannot be placed before `OPTIONS` or before `SERVER`. Syntax error will be given if it is placed elsewhere.

*The server object defined by `SERVER server_name` needs to be created with **postgres_fdw** extension prior to running the statement. Detailed* *explanation is in the next section.*

4.2.2 Foreign Partition Table Creation using Enhance Partition Syntax

`WITH PUSHDOWN` can be used in partition table creation using the enhanced partition creation syntax defined in section 3. Consider the general rule below that incorporates `WITH PUSHDOWN` clause in partition by `RANGE` using the enhanced partition creation syntax. Same syntax rule applies to other partition types such as `LIST` and `HASH`.

```
/* Partition by Range with PUSHDOWN */
/* Same syntax rule applies to partition by List and Hash */

$ CREATE TABLE table_name ( column_name TYPE ) PARTITION BY RANGE ( column_name )
  (
      PARTITION partition_name_1 FOR VALUES FROM (lower_bound) TO (upper_bound)
          SERVER server_name WITH PUSHDOWN,
      PARTITION partition_name_2 FOR VALUES FROM (lower_bound) TO (upper_bound)
          SERVER server_name WITH PUSHDOWN,
      PARTITION partition_name_3 DEFAULT
          SERVER server_name WITH PUSHDOWN

      /* more partition definitions follow ...*/
  );
```

4.2.3 Foreign Sub-Partition Creation using Enhanced Partition Syntax

`WITH PUSHDOWN` can be used in sub-partition creation using the enhance partition creation syntax defined in section 3. Consider the general rule below that defines and automatically creates 3 sub-partitions on foreign server. Same syntax rule applies to other sub-partition types such as `LIST` and `HASH`.

```
/* Sub Partition Creation with PUSHDOWN
$ CREATE TABLE table_name ( column_name TYPE ) PARTITION BY RANGE ( column_name )
  (
      PARTITION partition_name_1 FOR VALUES FROM (lower_bound) TO (upper_bound)
      PARTITION BY RANGE( column_name )
      (
          /* Sub-Partition by RANGE */
          PARTITION sub_partition_name_1 FOR VALUES FROM (lower_bound) TO
              (upper_bound) SERVER server_name WITH PUSHDOWN,
          PARTITION sub_partition_name_2 FOR VALUES FROM (lower_bound) TO
              (upper_bound) SERVER server_name WITH PUSHDOWN,
          PARTITION sub_partition_name_3 SERVER server_name WITH PUSHDOWN

          /* more sub partition definitions follow ...*/
      )

      /* more partition definitions follow ...*/
  );
```

If sub-partitions are defined, the parent partition cannot be made foreign and therefore WITH PUSHDOWN clause cannot be used on the parent *partition. This is the expected behavior.*

4.3 Syntax and Placement of INCLUDE REMOTE Clause

`INCLUDE REMOTE` is an optional clause that can be added at the end of an existing `DROP TABLE` or `DROP FOREIGN TABLE` SQL statement to trigger the automatic table deletion on foreign servers. This clause can be used to drop both local and foreign tables and cannot be used with `ALTER`, `UPDATE, CREATE` or `INSERT` clauses. The following sections outline different use cases for the `INCLUDE REMOTE`.

4.3.1 Foreign Table Deletion

Consider this general rule below when using `INCLUDE REMOTE` in the end of `DROP FOREIGN TABLE`;

```
DROP FOREIGN TABLE IF EXISTS table_name INCLUDE REMOTE;
```

This general command will drop the table named *`table_name`* in both local and the foreign server.

4.3.2 Foreign Partition Table Deletion

To automatically drop the local and foreign partition tables, the user can follow the general rule below:

```
DROP TABLE table_name INCLUDE REMOTE;
```

This will drop the local table, named *`table_name`*, along with all the partition tables created on the local and foreign servers. A list of foreign partition table and foreign server names and will be printed as NOTICE as the query executes.

If the user creates a table containing foreign sub-partitions, the same rule can be used to drop the table and all the sub-partitions created on foreign servers

4.3.3 Foreign Table Deletion with CASCADE clause

Similar to the standard PostgreSQL syntax rule, an error will be returned if the user attempts to drop foreign or local tables that have dependencies from other tables, such as views. To resolve this error, the user can manually drop the dependent tables first or simply include the `CASCADE` clause in the query statement to automatically drop the table and its dependencies. `CASCADE` clause can also work with the `INCLUDE REMOTE` clause according to this general rule below:

```
DROP TABLE table_name CASCADE INCLUDE REMOTE;
```

This will drop all the dependency tables on both local and foreign server related to *`table_name`*, before dropping *`table_name`* and its associated partitions or sub-partitions on local and foreign servers.

Please note that when "CASCADE" must be placed before "INCLUDE REMOTE" and after table_name. Syntax error will return if placed elsewhere.

4.4 Create Foreign Server Objects on Local Database Server

Before we can use the new `WITH PUSHDOWN` clause to create foreign tables automatically, we need to define server objects on the local database that represent the foreign servers. The server object creation follows the standard PostgreSQL syntax and more details can be found here:

https://www.postgresql.org/docs/12/postgres-fdw.html

As examples, we will define 2 server objects, S1 and S2, to represent 2 foreign servers in which we will refer to them throughout the rest of this section. As an example, both foreign servers and local server have super user named 'highgo' and database named 'postgres'.

The procedure to create server objects can be broken down into the following steps:

- Install the **postgres_fdw** extension using `CREATE EXTENSION`. Please note that you must have installed the **hg-pgsql12-contrib** package to get access to **postgres_fdw** extension. Please refer to the detailed installation guide if it has not been installed.

```
$ CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
```

- Next, we will create the server objects S1 and S2 to represent 2 foreign servers

```
$ CREATE SERVER S1 FOREIGN DATA WRAPPER postgres_fdw OPTIONS
(
    dbname 'postgres',
    host '172.17.0.3',
    port '5333'
);
CREATE SERVER

$ CREATE SERVER S2 FOREIGN DATA WRAPPER postgres_fdw OPTIONS
(
    dbname 'postgres',
    host '172.17.0.4',
    port '5333'
);
CREATE SERVER
```

- Then, we will create user mappings that map local super user 'highgo' to remote super user 'highgo' for both server objects.

```
$ CREATE USER MAPPING FOR highgo SERVER S1 OPTIONS ( user 'highgo' );
CREATE USER MAPPING

$ CREATE USER MAPPING FOR highgo SERVER S2 OPTIONS ( user 'highgo' );
CREATE USER MAPPING
```

- Finally, the server objects, S1 and S2, are ready to be used and we will be using them in the coming sections to illustrate the convenience of `WITH PUSHDOWN` clause.

4.5 Automatic Foreign Table Creation and Deletion Example

Having the server objects created in the previous section, we can then create a foreign table at server object S1.
'

```
$ CREATE FOREIGN TABLE
    IF NOT EXISTS tf1_f ( i INT, j VARCHAR ) SERVER S1
    OPTIONS ( schema_name 'public', table_name 'tf1_data' );

CREATE FOREIGN TABLE
```

The statement above only creates a foreign table representation on the local database server and this table does not yet exist in the foreign server S1. We will get an error if we attempt to operate data against this foreign table.

To automatically create the foreign table on the foreign database server represented by S1, use the syntax as following:

```
$ CREATE FOREIGN TABLE
    IF NOT EXISTS tf1_f ( i INT, j VARCHAR ) SERVER S1
    OPTIONS ( schema_name 'public', table_name 'tf1_data' )
    WITH PUSHDOWN;

CREATE FOREIGN TABLE
```

`WITH PUSHDOWN` clause is optional and when added at the end of statement, the foreign table **tf1_f** will be created at the local server and the table **tf1_data** will be created automatically on the foreign server. Please note that the above example uses the `OPTION (table_name ‘tf1_data’)` to illustrate that we can have different table names existing in both local and foreign servers. In this case, the data inserted to **tf1_f** table on local server will be available at **tf1_data** table in the foreign server.

To drop the foreign table created in the previous example:

```
$ DROP FOREIGN TABLE tf1_f INCLUDE REMOTE;

NOTICE: Drop remote table ‘public.tf1_data’ on foreign server ‘S1’
DROP FOREIGN TABLE
```

Note that a line of NOTICE will print the name of the relation to be dropped and indicate the residing foreign server object.

Based on the same example, if the user later creates a `VIEW` on the foreign table named `view1`, which depends on table **tf1_f,** the previous `DROP FOREIGN TABLE` command will fail due to dependency. In this case, we will use the `CASCADE` clause to automatically drop the dependencies and the table.

```
$ DROP FOREIGN TABLE tf1_f CASCADE INCLUDE REMOTE;

NOTICE: Drop remote table 'public.tf1_data' on foreign server 'S1'
NOTICE: drop cascades to view public.view1
DROP FOREIGN TABLE
```

Note that another NOTICE will be printed to show the dependency that has been dropped as the result of `CASCADE` clause.

4.6 Automatic Foreign Partition Table Creation and Deletion Example

The `WITH PUSHDOWN` clause can also be used with the new partition creation syntax described in section 3. Consider the following 2 examples, one using `WITH PUSHDOWN` clause while the other does not.

```
$ CREATE TABLE prt_com1_f ( a INT ) PARTITION BY RANGE ( a )
  (
      PARTITION prt_com1_p1_f FOR VALUES FROM (1) TO (10) SERVER S1,
      PARTITION prt_com1_p2_f FOR VALUES FROM (10) TO (20) SERVER S2,
      PARTITION prt_com1_p3 DEFAULT
  );
CREATE TABLE
```

The example above creates 3 partitions tables in which **prt_com1_p1_f** and **prt_com1_p2_f** are declared as foreign tables referencing to server objects **S1** and **S2**. These two foreign tables are created in the local database server only and they do not exist yet in serves **S1** and **S2**.

Consider the same example with "`WITH PUSHDOWN`" clause added at the end of partition declaration.

```
$ CREATE TABLE prt_com1_f ( a INT ) PARTITION BY RANGE ( a )
  (
      PARTITION prt_com1_p1_f FOR VALUES FROM (1) TO (10) SERVER S1 WITH PUSHDOWN,
      PARTITION prt_com1_p2_f FOR VALUES FROM (10) TO (20) SERVER S2 WITH PUSHDOWN,
      PARTITION prt_com1_p3 DEFAULT
  );
CREATE TABLE
```

The above statement will not only create **prt_com1_p1_f** and **prt_com1_p2_f** in the local database but also in their corresponding foreign servers S1 and S2. This eliminates the need to manually create the same tables on the foreign servers.

To drop the foreign partition table created in the previous example:

```
$ DROP TABLE prt_com1_f INCLUDE REMOTE;

NOTICE: Drop remote table 'public.prt_com1_p2_f' on foreign server 'S2'
NOTICE: Drop remote table 'public.prt_com1_p1_f' on foreign server 'S1'
DROP TABLE
```

This SQL statement will drop the local table and partitions as well as the partition table **prt_com1_p2_f** and **prt_com1_p2_f** on the foreign servers automatically.

Based on the same example, if the user later creates a `VIEW` on the foreign table named `view1` on `S1` and `view2` on `S2` that depends on table **prt_com1_p2_f** and **prt_com1_p2_f** respectively, the previous `DROP TABLE` command will fail due to dependency. In this case, we will use the `CASCADE` clause to automatically drop the dependencies and partitioned tables on foreign servers.

```
$ DROP TABLE prt_com1_f CASCADE INCLUDE REMOTE;

NOTICE: Drop remote table 'public.prt_com1_p2_f' on foreign server 'S2'
NOTICE: drop cascades to view public.view2
NOTICE: Drop remote table 'public.prt_com1_p1_f' on foreign server 'S1'
NOTICE: drop cascades to view public.view1
DROP TABLE
```

4.7 Automatic Foreign Sub-Partition Table Creation and Deletion Example

Consider the following 2 examples of creating foreign sub partitions with and without `WITH PUSHDOWN` clause.

```
$ CREATE TABLE prt_com4(a INT, b INT, c INT)
PARTITION BY RANGE( a )
(
      PARTITION prt_com4_p1 FOR VALUES FROM (0) TO (100)
      PARTITION BY RANGE( a )
      (
              PARTITION prt_com4_p1_1 FOR VALUES FROM (0) TO (10) SERVER S1,
              PARTITION prt_com4_p1_2 FOR VALUES FROM (10) TO (20) SERVER S2,
              PARTITION prt_com4_p1_3 DEFAULT
      )
);
CREATE TABLE
```

The example above creates 3 partitions tables in which **prt_com4_p1_1** and **prt_com4_p1_2** are declared as foreign tables referencing to server objects **S1** and **S2**. These two foreign tables are created in the local database server only and they do not exist yet in serves **S1** and **S2**.

Consider the same example with `WITH PUSHDOWN` clause added at the end of sub-partition declaration.

```
$ CREATE TABLE prt_com4(a INT, b INT, c INT)
PARTITION BY RANGE( a )
(
      PARTITION prt_com4_p1 FOR VALUES FROM (0) TO (100)
      PARTITION BY RANGE( a )
      (
              PARTITION prt_com4_p1_1 FOR VALUES FROM (0) TO (10) SERVER S1
                  WITH PUSHDOWN,
              PARTITION prt_com4_p1_2 FOR VALUES FROM (10) TO (20) SERVER S2
                  WITH PUSHDOWN,
              PARTITION prt_com4_p1_3 DEFAULT
      )
);
CREATE TABLE
```

The above statement will not only create **prt_com4_p1_1** and **prt_com4_p2_2** in the local database but also in their corresponding foreign servers S1 and S2. This eliminates the need to manually create the same tables on the foreign servers.

To drop the foreign sub-partition tables created in the previous example:

```
$ DROP TABLE prt_com4 INCLUDE REMOTE;

NOTICE: Drop remote table 'public.prt_com4_p1_2' on foreign server 'S2'
NOTICE: Drop remote table 'public.prt_com4_p1_1' on foreign server 'S1'
DROP TABLE
```

if the user later creates a `VIEW` on the foreign table named `view1` on `S1` and `view2` on `S2`, which depend on **prt_com4_p1_1** and **prt_com4_p2_2**, the previous `DROP TABLE` command will fail due to dependency. In this case, we will use the `CASCADE` clause to automatically drop the dependencies on and sub-partitioned tables on foreign servers.

```
$ DROP TABLE prt_com4 CASCADE INCLUDE REMOTE;

NOTICE: Drop remote table 'public.prt_com4_p1_2' on foreign server 'S2'
NOTICE: drop cascades to view public.view2
NOTICE: Drop remote table 'public.prt_com4_p1_1' on foreign server 'S1'
NOTICE: drop cascades to view public.view1
DROP TABLE
```

4.8 Common Misuse and Misinterpretation

4.8.1 Table and Partition Constraints are not Affected by 'WITH PUSHDOWN'

Consider the SQL statement below that creates a foreign partition with a constraint:

```
$ CREATE TABLE prt_com1_f ( a INT ) PARTITION BY RANGE ( a )
  (
      PARTITION prt_com1_p1_f (CONSTRAINT con_prt_com1_p1 CHECK ( a != 19 )) FOR
          VALUES FROM (1) TO (10) SERVER S1 WITH PUSHDOWN,
      PARTITION prt_com1_p2_f FOR VALUES FROM (10) TO (20) SERVER S2 WITH PUSHDOWN,
      PARTITION prt_com1_p3 DEFAULT
  );
CREATE TABLE
```

The constraint defined will only exist in the local database server and not propagated to the foreign server with the "WITH PUSHDOWN" clause. This is intended behavior and the user will be required to manually add the constraint to foreign server.

4.8.2 'WITH PUSHDOWN' cannot be Used on Temporary Tables

Consider the SQL statement below that attempts to perform "WITH PUSHDOWN" on a temporary table.

```
CREATE TEMP TABLE prt_com5(a INT, b INT, c INT) PARTITION BY RANGE(a)
(
    PARTITION prt_com5_p1 FOR VALUES FROM (0) TO (100)
    PARTITION BY RANGE(a)
    (
        PARTITION prt_com5_p1_1 FOR VALUES FROM (0) TO (10) SERVER S1
        WITH PUSHDOWN
    )
);
ERROR:  WITH PUSHDOWN is only allowed for permanent relations
```

"WITH PUSHDOWN" cannot be applied to a table declared as "TEMP", an error will be given if the pushdown is attempted on temporary table.

4.8.3 'WITH PUSHDOWN' Returns Error if Foreign Table Exists Already

Consider the example below where we attempt to create partitions as foreign tables using `WITH PUSHDOWN` clause twice in a row with a regular `DROP TABLE` clause in between. First attempt will succeed, the second will fail because `WITH PUSHDOWN` clause will not create a table if it exists already.

```
$ CREATE TABLE prt_com1_f ( a INT ) PARTITION BY RANGE ( a )
  (
      PARTITION prt_com1_p1_f FOR VALUES FROM (1) TO (10) SERVER S1 WITH PUSHDOWN,
      PARTITION prt_com1_p2_f FOR VALUES FROM (10) TO (20) SERVER S2 WITH PUSHDOWN
  );

CREATE TABLE

/* this will drop the local tables, not foreign */
$ DROP TABLE prt_com1_f;

DROP TABLE

$ CREATE TABLE prt_com1_f ( a INT ) PARTITION BY RANGE ( a )
  (
      PARTITION prt_com1_p1_f FOR VALUES FROM (1) TO (10) SERVER S1 WITH PUSHDOWN,
      PARTITION prt_com1_p2_f FOR VALUES FROM (10) TO (20) SERVER S2 WITH PUSHDOWN
  );
ERROR: Failed to execute CREATE TABLE on remote server
```

"WITH PUSHDOWN" will NOT overwrite the remote tables having the same relation names.

4.8.4 The Parent Partition and Sub-Partition Tables cannot be Foreign at the Same Time

Consider the SQL statement below that attempts to make both the parent and sub partitions foreign at the same time. Doing so will result in error message being given due to incorrect usage. The clauses causing the error are highlighted in below example.

```
$ CREATE TABLE prt_com4(a int, b int, c int) PARTITION BY RANGE(a)
(
      PARTITION prt_com4_p1 FOR VALUES FROM (0) TO (100) SERVER S1
      PARTITION BY RANGE(a)
      (
              PARTITION prt_com4_p1_1 FOR VALUES FROM (0) TO (10) SERVER s1,
              PARTITION prt_com4_p1_2 FOR VALUES FROM (10) TO (20) SERVER s2,
              PARTITION prt_com4_p1_3 DEFAULT
      ),
      PARTITION prt_com4_p2 FOR VALUES FROM (100) TO (200) SERVER S2
```

```
        PARTITION BY LIST(b)
        (
                PARTITION prt_com4_p2_1 FOR VALUES IN (1,2,3,4) SERVER s1,
                PARTITION prt_com4_p2_2 FOR VALUES IN (5,6,7,8) SERVER s2,
                PARTITION prt_com4_p2_3 DEFAULT
        ),
        PARTITION prt_com4_p3 DEFAULT SERVER S1
        PARTITION BY HASH(c)
        (
                PARTITION prt_com4_p3_1 FOR VALUES WITH(MODULUS 3, REMAINDER 0),
                PARTITION prt_com4_p3_2 FOR VALUES WITH(MODULUS 3, REMAINDER 1),
                PARTITION prt_com4_p3_3 FOR VALUES WITH(MODULUS 3, REMAINDER 2)
        )
);
ERROR: Partition table can not be a foreign table
```

Please note that only the partitions that do not have any children sub-partitions can be created as 'foreign'. In other words, only the partitions at the end of the partition hierarchy tree can be made 'foreign'; their parent partitions cannot be made as 'foreign'.

5 Parallel Backup

Full base backup refers to an act of making an identical copy of the database cluster files and is normally performed by the front end tool **pg_basebackup**, which will always make a copy of the entire database cluster. This tool works in single thread mode and its process of taking a full backup of a large database can take a very long time and potentially slow down the online database operations.

HG-PGSQL introduces parallel backup feature that is built within **pg_basebackup** front end tool and user can specify the number of worker threads as command line arguments. This allows **pg_basebackup** tool to spawn multiple parallel workers that help spread the total work load; each worker can perform full base backup on a portion of the target database cluster in parallel and therefore utilizing the system resources more efficiently and reducing the time required.

The desired number of worker threads depends on the size of the target database, the disk`s IO performance and the maximum thread limits settings on unix-based systems. The user should carefully evaluate the system and select an appropriate number of workers such that the backup process is optimized with minimal impact to other system operations in terms of resource usages.

Our team at HighGo has simulated a typical real-world database setup and done several benchmarking tests on parallel backup performance. The benchmark shows the backup performance with respect to the number of parallel workers executed on systems having high Input Output Processor (IOP) rating.

The white paper of HighGo’s benchmark tests can be found at

https://www.highgo.ca/products/highgo-postgresql-server

5.1 Authentication and Connection Limit Configuration

The front-end tool **pg_basebackup** performs the backup based on replication protocol supported in standard PostgreSQL distribution. Therefore, some initial setup must be done for the backup to execute properly.

5.1.1 Set Proper max_wal_sender Parameter

The maximum number of parallel worker that **pg_basebackup** can efficiently spawn depends on the `max_wal_sender` configuration parameter in the target database server. HG-PGSQL defaults this value to 10 and it can be changed by modifying **postgresql.conf.** See the snapshot below

```
#------------------------------------------------------
# REPLICATION
#------------------------------------------------------

# - Sending Servers -

# Set these on the master and on any standby that will send replication data.

#max_wal_senders = 10                # max number of walsender processes
                                     # (change requires restart)
```

*If **pg_basebackup** attempts to spawn more parallel workers than the maximum number that server can accept, the excessive parallel workers will get a connection failure. However, the backup process will continue to perform by the workers that have obtained a replication connection from the target server*

5.1.2 Set Proper Replication Connection Permission

Before **pg_basebackup** can connect successfully to a server to perform full base backup, the server must have allowed such connections first in **pg_hba.conf**. For example, if a user plans to run **pg_basebackup** as database user **highgo** from a host having IP = 172.17.0.2. The Server must have an entry added in **pg_hba.conf** that allows this connection. See snapshot of **pg_hba.conf** below.

```
# Allow replication connections from localhost, by a user with the
# replication privilege.
local    replication    all                                 peer
host     replication    all          127.0.0.1/32           ident
host     replication    highgo       172.17.0.2/32          trust
host     replication    all          ::1/128                ident
```

*If **pg_basebackup** attempts to connect to a database server in which it does not have replication permission configured, authentication failure will be returned and full base backup will abort.*

5.2 Parallel Backup Usages

A new command line option (`-j | --jobs=NUM`) is added to **pg_basebackup** front end tool that accepts a positive integer larger than 0 denoting the number of parallel worker threads that it should spawn to process the full base backup operation. This is an optional argument that can be used in combination with other existing arguments to ensure a successful backup operation.

Some of the most commonly used **pg_basebackup** commands including the new argument are listed below. Please note that if destination hostname and port number are not specified, the tool will use the values defined in environment variables, `PGHOST` and `PGPORT`. If database user is not specified, the current system user will be assumed to be database user. Please also note that the following examples will perform backup to `$BACKUP_DIR`, and this directory must be clean before backup is attempted, otherwise the tool will refuse to do backup. For more information on the usage, use `--help` argument to see all supported arguments and their usages.

- Spawn 4 parallel workers to perform base backup to `$BACKUP_DIR`

```
 $ pg_basebackup -j 4 -D $BACKUP_DIR
```

- Spawn 10 parallel workers to perform base backup to `$BACKUP_DIR` in verbose mode

```
$ pg_basebackup --jobs=10 -D $BACKUP_DIR -v
```

- Use single thread mode to perform base backup to `$BACKUP_DIR`. This essentially disables the parallel backup feature and uses the original ways of performing a backup.

```
$ pg_basebackup -D $BACKUP_DIR
```

- Spawn 4 parallel workers to perform base backup to `$BACKUP_DIR` and re-map table space ts1 and ts2 to a new location `$BACKUP_DIR1` and `$BACKUP_DIR2` while skipping checksum verification.

```
$ pg_basebackup -j 4 -D $BACKUP_DIR                              \
  -T /home/highgo/test/ts1=$BACKUP_DIR1/new_tablespace/    \
  -T /home/highgo/test/ts2=$BACKUP_DIR2/new_tablespace/    \
  --no-verify-checksums
```

- Spawn 4 parallel workers to perform base backup to `$BACKUP_DIR` and show progress information

```
$ pg_basebackup -D $BACKUP_DIR -P -j 4
```

- Create a replication slot named **backup** and spawn 10 parallel workers to connect to this slot to perform base backup to `$BACKUP_DIR` while showing progress information .

```
$ pg_basebackup -D $BACKUP_DIR -j 10 -S backup -C --slot=$SLOT -P
```

- Spawn 4 parallel workers to perform base backup to `$BACKUP_DIR` and populate a `postgres.auto.conf` containing the `primary_conninfo` and `primary_slot_name` to connect to the main DB server as standby. This option is normally used when a user would like to setup the base backup as a standby server.

```
$ pg_basebackup -D $BACKUP_DIR -j 4 -S backup -C --slot=$SLOT -R
```

- Spawn 4 parallel workers to perform base backup but limit the overall bandwidth consumption to be 100 Megabytes per second at most.

```
$ pg_basebackup -D $BACKUP_DIR -j 4 --max-rate=100m -D $BACKUP_DIR
```