HighGo Postgres Server (HG-PGSQL) User Guide

Version 2.1 GA

December 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

CHANGE LOG

Date	Version	Author	Description
Nov, 26, 2020	v2.0 Beta	C.Huang	Initial Version for HG-PGSQL 2.0
Dec, 24, 2020	v2.1 GA	Ahsan, Movead	Parallel foreign scan, Package limitations

Table of Contents

1 Introduction

HighGo Software Inc is glad to release HighGo PostgreSQL Sever (HG-PGSQL 2.1 GA) that provides unique additional functionality on top of the open source PostgreSQL database. HighGo Software is committed to delivering value to its end-users through innovation and building on top of open source-based database solution (PostgreSQL 13.1). Our goal is to deliver a solution with high performance, scalability, reliability, and ease of use for small, medium and large-scale enterprises. With this release, HighGo Software continues to innovate and build on top of open source based database solution (PostgreSQL 13.1) with aims to deliver a solution with high performance, reliability and ease of use for small, medium and large-scale enterprises.

The extended functionality provided by HG-PSQL will enable users to build a highly performant and scalable PostgreSQL database clusters with better database compatibility and administration. This simplifies the process of migration to PostgreSQL from other DBMS with enhanced database administration experiences.

The purpose of this guide is to provide comprehensive details with examples to showcase the new features in HG-PSQL 2.1 GA.

1.1 New Feature Overview

- **Global Transaction Management**
  HG-PGSQL 1.0 provided a new syntax for creating partitioned tables that can span over one or more foreign server i.e. Shards. It enabled the user to create Sharded table where partitions live on external foreign servers. The latest version of HG-PGSQL 2.1 release provides the Global Transaction Management (GTM) capability that will enable the Sharded cluster to be used for OLTP write enabled workload. This guide will provide comprehensive details about the global transaction management feature that enables HG-PGSQL sever to be used for Horizontality scalable OLTP workload.

- **Workload Environments**
  HG-PGSQL 2.1 enhances the database compatibility by providing package features to ease the complexities of migration to PostgreSQL from other DBMS systems.

- **Administration**
  Managing orphaned prepared transaction has always been a pain, HG-PGSQL 2.1 provides administration feature that will manage the orphan prepared transactions.

- **Parallel Foreign Scan**

  HG-PGSQL 2.1 provides parallel scan of postgres FDW nodes, if a query fires several foreign scans they would be processed sequentially without this feature. The parallel foreign scan is also referred as async execution of FDW nodes, it enables the foreign scans to be executed in parallel on multiple foreign servers hence providing horizontal scalability and significant performance gains.

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
- CentOS (X86_64) 8.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-1.1-2.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. The Major version upgrade from v1.0 to v2.1 can also be found there:

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

3 Global Transaction Management (GTM)

3.1 Overview

A database transaction involving two or more network hosts is referred as a distributed transaction in this documentation. Normally, hosts provide transactional resources, while the PostgreSQL transaction manager creates a single global transaction and manage all operations around these resources. Similarly, distributed transactions must also conform to all four ACID (Atomicity, Consistency, Isolation, Durability) properties, where atomicity guarantees all-or-nothing outcomes for any unit of work.

Formerly, transactions on foreign servers were simply committed or rolled back one by one. Therefore, when one foreign server had a problem during commit, it was possible that transactions on only part of foreign servers were committed while other transactions were rolled back. This used to leave database data in an inconsistent state in terms of federated database.

HG-PGSQL introduces two new features for managing distributed transactions. Atomic commit of distributed transactions and atomic visibility for the distributed transactions. Both features are built within the postgres_fdw extension in HG-PGSQL and can be extended to other FDWs. These are useful features designed for data consistency and integrity when data is distributed on multiple servers.

These HG-PGSQL features also provide a major step forward in greatly enhancing FDW based Sharding feature described in section 4. 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.

HG-PGSQL v1.0 provides the shard management feature which allows the user to create a sharded table where the partitions can reside on different foreign servers. The Shard management provides the ease of use syntax for creating the sharded table and much less chances of making a mistake in table creation.

HG-PGSQL 2.1 provides the features to manage global transactions, which enables write workloads to be used with Sharded tables which was not possible in v1.0 due missing distributed transaction feature. This is a great stride in completing the feature set for Sharding in HG-PGSQL and making it possible for both write and read workloads to benefit from the Sharding solution.

3.2 Atomic Commit

Atomic commit of a distributed transaction is an operation that applies a set of changes in a single operation globally. This guarantees all-or-nothing results for the changes on all remote hosts involved. PostgreSQL provides a way to perform read-write transactions with foreign resources using foreign data wrappers.

A common protocol for ensuring correct completion of a distributed transaction is the two-phase commit (2PC) (Also known as Prepared Transaction). This protocol is usually applied to updates that can be committed in a short period of time, ranging from a couple of milliseconds to a couple of minutes.

Unlike single-phase transactions, two-phase transactions are not associated with any database sessions. This means the transaction can still be committed later even if the originating database session who prepared the transaction exits or crashes. The prepared transaction is persisted in the database cluster unless it is properly resolved or committed.

For this reason, using a 2PC protocol requires a centralized transaction manager that can keep track of all prepared transactions and provide a mechanism to view and resolve the pending prepared transactions.

DTM can be built into the database server or be deployed externally outside of the database system. HG-PGSQL solution of atomic commit implements the DTM inside the PostgreSQL core. For this reason, there will not be a need to install a separate transaction manager server as all the transaction management functions come as a part of the PostgreSQL server and can be invoked by a standard PostgreSQL client easily.

3.2.1 Internals of Atomic Commit

3.2.1.1 Atomic Commit Using Two-Phase Commit Protocol

To achieve commit among all foreign servers automatically, HG-PGSQL employs two-phase commit protocol, which is a type of atomic commitment protocol (ACP). The commit sequence of distributed transaction is as follows:

1. **Prepare**:
   HG-PGSQL's distributed transaction manager prepares all transactions on the foreign servers if a two-phase commit is deemed required. HG-PGSQL decides if 2PC is required based on if the transaction modifies data on two or more servers (including the local one) and also based on the value of `foreign_twophase_commit` parameter.

   If the preparation on all foreign servers is successful, then go to the next step otherwise in case of any failure rollback all prepared transactions including the local transaction.

2. **Commit locally**:
   The server commits the local transaction. Failure in doing so results in rollback of all previously prepared foreign transactions.

3. **Resolve all PREPARED TRANSACTIONS on foreign servers**:
   Prepared transactions are committed or rolled back according to the result of the local transaction. This step is performed by a foreign transaction resolver process.

Each commit of a distributed transaction waits until confirmation is received that all prepared transactions are committed or rolled-back. This guarantees that the application will not receive explicit acknowledgement of the successful commit of a distributed transaction until all foreign transactions are resolved on the foreign servers.

If synchronous replication is used, the distributed transaction waits for synchronous replication first, and then waits for foreign transaction resolution.

3.2.1.2 Foreign Transaction Resolver Processes

Foreign transaction resolver processes are auxiliary processes that are responsible for resolving both foreign transactions that are prepared by online transactions and in-doubt transactions. Their main task is to commit or rollback prepared transactions on foreign servers.

One of the foreign transaction resolver processes is responsible for transaction resolutions on the database it is connected to. On failure during resolution, it keeps retrying at `foreign_transaction_resolution_interval` intervals.

DROP DATABASE is not allowed when the foreign transaction resolver process is active on a database. You can call `pg_stop_foreign_xact_resovler` function to stop a particular resolver process before dropping the database.

3.2.1.3 In-Doubt Transactions

The atomic commit mechanism ensures that all foreign servers either commit or rollback using two-phase commit protocol. However foreign transactions can become in-doubt in two cases:

- The local node crashed during either preparing or resolving foreign transaction.
- User explicitly cancels the query.

3.2.2 Atomic Commit and Rollback of Distributed Transaction

If the distributed transaction involves the FDW servers that support the prepared transactions, then committing and rolling back the distributed transaction does not require any special command. PostgreSQL core of HG-PGSQL server automatically prepares and commit/rollback the transactions on FDW servers when needed.

For example, consider a distributed transaction:

```
BEGIN;
UPDATE LT  SET col = ‘a’;
UPDATE FT1 SET col = 'b';
UPDATE FT2 SET col = 'c';
COMMIT;
```

FT1 and FT2 are foreign tables on different foreign servers and may even be using different Foreign Data Wrappers, while LT is the table on the local server. When the core executor accesses the foreign servers, the foreign servers that support transaction management callback routines are registered as a

participant. During the pre-commit phase of local transaction, the foreign transaction manager prepares all the foreign transactions on the respective foreign servers. If all foreign transactions are prepared successfully the local transaction is committed by the core. In case any failure happens or user requests to cancel during preparation, the distributed transaction manager changes the action from 'commit transaction' to 'rollback transaction'.

Once local transaction is committed successfully 'COMMIT PREPARED' is issued on all foreign servers and control is handed over to foreign transaction resolver process and the local server will wait for the resolver process to commit all prepared transactions. Once it is done, the transaction is marked as completed and backend gets ready for next user command.

Two-phase commit is only utilized when:

- *Enabled by the configuration setting, foreign_twophase_commit.*
- *The transaction modifies data on more than one server including the local one*

3.2.3 Viewing the in-Progress Foreign Transactions

The view pg_foreign_xacts displays information about foreign transactions that are opened on foreign servers for atomic distributed transaction commit.

pg_foreign_xacts contains one row per foreign transaction. An entry is removed when the foreign transaction is committed or rolled back. Below is a summary of all the column values.

Name	**Type**	**Reference**	**Description**
dbid	OID	pg_database.oid	OID of the database which the foreign transaction resides in
xid	XID	pg_foreign_server.oid	Numeric transaction identifier with which this foreign transaction associates
serverid	OID	pg_user.oid	The OID of the foreign server on which the foreign transaction is prepared
userid	OID		The OID of the user that prepared this foreign transaction.
status	TEXT		Status of foreign transaction. Possible values are: • preparing : This foreign transaction is being prepared. • prepared : This foreign transaction has been

			prepared. • committing : This foreign transaction is being committed. • aborting : This foreign transaction is being aborted.
in-double	BOOLEAN		If true this foreign transaction is in-doubt status. A foreign transaction becomes in-doubt status when the user canceled the query during transaction commit or the server crashed during transaction commit.
identifier	TEXT		The identifier of the prepared foreign transaction.

When pg_foreign_xacts view is accessed, the internal transaction manager data structures are momentarily locked, and a copy is made for the view to display. This ensures that the view produces a consistent set of results, while not blocking normal operations longer than necessary. Nonetheless there could be some impact on database performance if this view is frequently accessed.

3.2.4 Foreign Transaction Management and Control Functions

Signature	Description
bool **pg_resolve_foreign_xact**(transaction xid, serverid oid, userid oid)	Resolve a foreign transaction. The function searches for foreign transactions matching the arguments and resolves it. Once the foreign transaction is resolved successfully, this function removes the corresponding entry from foreign transactions list. This function will not resolve a foreign transaction which is being processed.
bool **pg_remove_foreign_xact**(transaction xid, serverid oid, userid oid)	This function works the same as pg_resolve_foreign_xact except that this removes the foreign transaction entry without resolution.
bool **pg_stop_foreign_xact_resolver**(dbid oid)	Stop the foreign transaction resolver running on the given database. This function is useful for stopping a resolver process on the database that you want to drop.

3.2.5 Configuring Atomic Commit

3.2.5.1 foreign_twophase_commit (enum)

Specifies whether distributed transaction commits ensure that all involved changes on foreign servers are committed or not. Valid values are ‘required’ or ‘disabled’.

Setting to disabled disables the use of two-phase commit protocol. When set to required, the distributed transactions strictly requires that all written servers should use two-phase commit protocol.

This parameter can be changed at any time; the behavior for any one transaction is determined by the setting in effect when it commits.

When disabled, there can be a risk of database inconsistency if one or more foreign servers crash while committing the distributed transactions.

3.2.5.2 max_prepared_foeign_transactions (integer)

Sets the maximum number of foreign transactions that can be prepared simultaneously. A single local transaction can give rise to multiple foreign transactions. If a user expects N local transactions and each of those involves K foreign servers, this value needs to be set at least N * K, not just N. Changing this parameter requires a server restart.

When running a standby server, you must set this parameter to the same or higher value than on the master server. Otherwise, queries will not be allowed in the standby server.

3.2.6 Configuring Foreign Transaction Resolvers

3.2.6.1 max_foreign_transaction_resolves (int)

Specifies maximum number of foreign transaction resolver workers. A foreign transaction resolver is responsible for foreign transaction resolution on one database.

Foreign transaction resolution workers are taken from the pool defined by max_worker_processes.

3.2.6.2 foreign_transaction_resolution_retry_interval (integer)

Specify how long the foreign transaction resolver should wait when the last resolution fails before retrying to resolve foreign transaction. This parameter can only be set in the postgresql.conf file or on the server command line.

3.2.6.3 foreign_transaction_resolver_timeout (integer)

Terminate foreign transaction resolver processes that do not have any foreign transactions to resolve longer than the specified number of milliseconds. A value of zero disables the timeout mechanism, meaning it connects to one database until stopping manually by pg_stop_foreign_xact_resovler(). This parameter can only be set in the postgresql.conf file or on the server command line.

3.3 Atomic Visibility

Atomic visibility ensures that when a distributed transaction is started, the changes done by the other transactions committing at the same point in time are either fully visible to this transaction or are totally invisible to it. To achieve the atomic visibility in the distributed transaction HG-PGSQL provides the CSN based global snapshots and employs Clock-SI approach for catering the time skew among the participating servers.

3.3.1 CSN Based Snapshot

By default, the snapshots in PostgreSQL uses the XID (TransactionID) to identify the status of the transaction, the in-progress transactions, and the future transactions for all its visibility calculations. HG-PGSQL also provides the CSN (Commit-Sequence-Number) based mechanism to identify the past-transactions and the ones that are yet to be started/committed.

When a CSN based snapshot is used, each commit is assigned a Commit Sequence Number, or CSN for short, using a monotonically increasing counter (current time in this case). A snapshot is represented by the value of the CSN counter at the time the snapshot was taken. All (committed) transactions with a CSN <= the snapshot's CSN are considered as visible to the snapshot.

3.3.2 Configuring Atomic Visibility

3.3.2.1 enable_csn_snapshot (boolean)

Enable/disable the CSN based transaction visibility tracking for the snapshot.

PostgreSQL uses the clock timestamp as a CSN, so enabling the CSN based snapshots can be useful for implementing the global snapshots and global transaction visibility.

when enabled, PostgreSQL creates a pg_csn directory under $PGDATA to keep the track of CSN and XID mappings.

3.3.2.2 csn_snapshot_defer_time (integer)

Sets the amount of time to hold the old snapshot data from getting vacuumed. This is useful when some remote (global) transaction that has imported our snapshot and requests the old data version from us, which may be vacuumed away because it was no more useful for us otherwise.

On the downside, enabling this defer time can cause bloating as it will make the dead tuples hang around longer than they otherwise would have.

The default is 0, which means do not hold the old data.

3.3.2.3 CSN Based Global Snapshots

When global snapshots are enabled then HG-PGSQL exports the current transaction CSN value to all foreign servers that are part of the distributed transaction before sending any query to the foreign server. This ensures that distributed transactions have the same view of data across all servers.

Similarly, before committing the distributed transaction, HG-PGSQL collects the current CSN value from all participating servers and calculates the current global CSN value using the collected CSNs and commits the transaction using that global CSN on all servers.

Currently only HG-PGSQL's Postgres-fdw provides the functionally to global snapshots, when global snapshots are enabled, each distributed transaction performs the following operations:

- At translation start, exports the local CSN snapshot and pushes that CSN value to all foreign participants.
- After doing PREPARE TRANSACTION, collect the CSN values from all distributed transaction participants
- Calculate the GLOBAL CSN value
- Assign the GLOBAL CSN to all remote participants including the local transaction
- Commit

Global snapshots are only supported with REPEATABLE READ isolation level.
For example:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_csn_snapshot_export() AS snapshota1 \gset
INSERT INTO prt_com1 VALUES(3);
INSERT INTO prt_com1 VALUES(30);
INSERT INTO prt_com1 VALUES(30);
COMMIT;
```

3.3.2.4 Enable_global_snapshot

Enables the global snapshot for distributed transactions. This only works when the CSN snapshots are enabled.

3.4 Sharding Enhancement

Atomic Commit in HG-PGSQL ensures that DML and DDL operations on a sharded table always produce consistent results.

For example:
Consider creating a sharding table using the 'WITH PUSHDOWN' option.

```
CREATE TABLE partationed_table ( id INT, name TEXT)
PARTITION BY RANGE(id)
(
    PARTITION prt_p1 FOR VALUES FROM (1) TO (10)
       SERVER server_1 WITH PUSHDOWN,
    PARTITION prt_p2 FOR VALUES FROM (10) TO (20)
       SERVER server_2 WITH PUSHDOWN,
    PARTITION prt_p3 DEFAULT
);
```

Where **server_1** and **server_2** are created as:

```
CREATE SERVER server_1 FOREIGN DATA WRAPPER
Postgres_fdw OPTIONS
(
       dbname 'highgo'
       Host 'db_server1_host'
);

CREATE SERVER server_2 FOREIGN DATA WRAPPER
Postgres_fdw OPTIONS
(
       dbname 'highgo'
       Host 'db_server2_host'
);
```

The above '**CREATE TABLE partationed_table**' query is internally divided into multiple CREATE statements, one for each partition and sent to respective foreign server. With the absence of Atomic Commit feature (introduced in current HG-PGSQL release) any error occurred in the internally divided CREATE statements will leave a partial table.

For example, if server_2 fails to create **prt_p2** table for some reason and returns an error during the execution of '**CREATE TABLE partationed_table**'. This will cause the original **CREATE TABLE** to fail and produce an error but by that time the **prt_p1** on server_1 is already created and despite the original transaction being failed that prt_p1 would remain on server_1.

Similarly, a DML operation on a sharded table also had a risk of leaving the data in inconsistent state in case of error.

For example, consider a transaction

```
BEGIN;
--Statement: S1 inserts into partition on server_1
```

```
INSERT INTO partationed_table values (1, 'one');
--Statement: S2 inserts into partition on server_2
INSERT INTO partationed_table values (11, 'eleven');
--Statement: S3 inserts into local partition
INSERT INTO partationed_table values (101, 'one O one');
--Statement: S4 Commit
COMMIT:
```

Suppose this transaction fails because of S2 failure. This will cause the transaction to fail but still leave the S1 committed on server_1, and you will end up with the inconsistent data in partitioned_table. The new atomic commit of distributed transactions ensures that if the original transaction fails it cleans up every single change it has made to all shards within the transactions.

3.5 Queries on Sharded Tables Produce Consistent Results with Global Snapshots

Global snapshots feature added to HG-PGSQL ensures that querying a sharded table always returns consistent results.

For example:
Consider the partitioned table having three partitions, two on remote servers and one on local

```
CREATE TABLE partationed_table ( id INT, name TEXT)
PARTITION BY RANGE(id)
(
    PARTITION prt_p1 FOR VALUES FROM (1) TO (10)
       SERVER server_1 WITH PUSHDOWN,
    PARTITION prt_p2 FOR VALUES FROM (10) TO (20)
       SERVER server_2 WITH PUSHDOWN,
    PARTITION prt_p3 DEFAULT
);
```

Two parallel sessions are doing operations on this partitioned table.

1st Session

```
SELECT * from partationed_table;
```

2nd parallel session

```
postgres=# INSERT INTO partationed_table values
   ( 2, 'two'),
   (12,'twelve'),
   (70,'seventy');
INSERT 0 3
```

Without the global snapshot there is no guarantee that querying a sharded table will return the complete result, meaning that there was a possibility that '**SELECT * from partationed_table**' return only two rows

```
postgres=# SELECT * from partationed_table;
 id |  name
----+---------
  2 | two
 70 | seventy
(2 rows)
```

Although all three rows were inserted as part of the same transaction but the absence of global snapshot and global visibility means that PostgreSQL was not able to determine which rows on the remote shards belong to which transaction so it returned the half/invalid data.

With a global snapshot feature HG-PGSQL ensures that querying the sharded table always returns the complete transaction data and never returns the half or inconsistent data.

4 Oracle Style Packages

4.1 Package Overview

This section journeys into the “Oracle Style Package” for PostgreSQL. A package by very definition is an object or a group of objects packed together. In terms of databases, this translates into a named schema object that packages within itself a logically grouped collection of procedures, functions, variables, cursors, user-defined record types, and reference records.

It is expected that users are familiar with PostgreSQL and has a good understanding of SQL language to better appreciate the packages and use these more efficiently.

4.1.1 The Need for Packages

Like similar constructs in various other programming languages, there are good reasons for using packages with SQL. In this section we are going to cover a few.

- **Reliability and Reusability of Code**
  Packages provide you the ability to create modular objects that encapsulate code. This makes the overall design and implementation simpler. With the ability to encapsulate variables and related types, stored procedures / functions, and cursors, it allows you to essentially create a self-contained module that is simple and easy to understand, maintain and use.

  Encapsulation comes into play through exposure of a package interface, rather than its implementation details, i.e. package body. This, therefore, benefits in many ways. It allows applications and users to refer to a consistent interface and not worry about the contents of its body. Also, it prevents users from making any decisions based on code implementation as that’s never exposed to them.

- **Ease of Use**
  The ability to create a consistent functional interface in PostgreSQL helps simplify application development by allowing compilation of packages without their bodies.

  Beyond the development phase, the package allows a user to manage access control on the entire package rather than individual objects. This is rather valuable especially if the package contains lots of schema objects.

- **Performance**
  Packages are loaded into memory for maintenance and therefore utilizing minimal I/O resources. Recompilation is simple and only limited to object(s) changed; dependent objects are not recompiled.

- **Additional Features**
  In addition to performance and ease of use, packages offer session-wide persistence for variables and cursors. This means variables and cursors have the same lifetime as a database session and are destroyed when the session is destroyed.

4.1.2 Package Components

In earlier sections, we briefly mention that a package has an interface and a body, which are the major components that make up a package.

- **Package Specification**
  Any object within the package that is to be used from the outside is specified in the package specification section. This is the publicly accessible interface we have been referring to in earlier sections. It does not contain the definition or implementation of them, i.e. the functions and the procedures. It only has their headers defined without the body definitions. The variables can be initialized. The following is the list of objects that can be listed in the specification:
    - Functions
    - Procedures
    - Cursors
    - Types
    - Variables
    - Constants
    - Records types

- **Package Body**
  The body contains all the implementation code of a package, including the public interfaces and the private objects. A package body is optional if the specification does not contain any subprogram or cursor.

  It must contain the definition of the subprograms declared in specification and the corresponding definitions must match.

  A package body can contain its own subprogram and type declarations of any internal objects not specified in the specifications. These objects are then considered private. Private objects cannot be accessed outside the package.

  In addition to subprogram definitions, it can optionally contain a initializer block that initializes the variables declared in specification and is executed only once when the first call to the package is made in a session.

Package body gets invalidated if the specification changes

Care must be taken when identifying the public interfaces and the private ones to avoid exposing critical functions and variables outside the package unexpected.

4.2 Package Syntax

4.2.1 Package Specification Syntax

```
CREATE [ OR REPLACE ] PACKAGE package_source;

package_source:
     [schema.] package_name [invoker_rights_clause]
          [IS | AS] item_list[, item_list] END [package_name];

invoker_rights_clause:
     AUTHID [CURRENT_USER | DEFINER]

item_list:
     [
          function_declaration       |
          procedure_declaration      |
          type_definition            |
          cursor_declaration         |
          item_declaration
     ]

function_declaration:
     function_heading

procedure_declaration:
     procedure_heading

type_definition:
     record_type_definition:
     ref_cursor_type_definition:

cursor_declaration:
     CURSOR name [(cur_param_decl[, ...])] RETURN rowtype;

item_declaration:
     constant_declaration
     cursor_declaration
     cursor_variable_declaration
     record_variable_declaration
     variable_declaration

function_heading:
     FUNCTION function_name [(parameter_declaration[, ...])]
          RETURN datatype;
```

```
procedure_heading:  PROCEDURE procedure_name
[(parameter_declaration[, ...])]

parameter_declaration:
     parameter_name [IN] datatype [[:= | DEFAULT] expr]
```

Where:

Parameter	Description
package_name	Defines the name of the package governed by SQL object naming rules.
invoker_rights_clause	Invoker rights define the access privileges for the package to the database objects. The options are available are: • CURRENT_USER Indicates that the access privileges for the current user executing the package will be used. • DEFINER Indicates that access privileges for the package creator will be used.
procedure_heading	Specifies a procedure name followed by its list of arguments. **procedure_name** is the name of the public procedure.
function_heading	Defines a function name, its arguments and its return type. **function_name** is the name of the public function.
type_definition	Suggests that you can define either a record, or cursor type.
cursor_declaration	Defines that cursor declaration must include its arguments and return type as the desired rowtype.
item_declaration	Allows declaration of: • Constants • Cursors • Cursor variables • Record variables • Variables
parameter_declaration	Defines the syntax for declaring a parameter. The keyword "**IN**" if specified indicates that this is an input parameter. The DEFAULT keyword followed by expression (or value) may only be specific for an

	input parameter.

4.2.2 Package Body Syntax

```
CREATE [ OR REPLACE ] PACKAGE BODY package_body_source;

package_body_source:
      [schema.] package_name [IS | AS] declare_section
            [initialize_section] END [package_name];

declare_section:
      [item_list, ...] | [item_list_2, ...]

initialize_section:
      BEGIN statement[, ...]

item_list_2:
      [
            function_declaration
            function_definition
            procedure_declaration
            procedure_definition
            cursor_declaration
            cursor_definition
      ]

function_definition:
      function_heading
            [IS | AS] [ declare_section [body] ]

body:
      BEGIN statement[, ...] END [name];

statement:
      [<<LABEL>>] pl_statments[, ...]

procedure_definition:
      procedure_heading [IS | AS] [ [declare_section [body] ]

cursor_definition:
      CURSOR name [(cur_param_decl[, ...])] RETURN rowtype;
```

Where:

Parameter	Description
package_body_source	Defines the name of the package governed by SQL object naming rules with the option to qualify it by prefixing the schema name before the dot.
declare_section	This contains all the elements that are to be defined and implemented by this package.
initialize_section	It is a special block in the package body that is executed only once when the package is referenced for the first time. The objective is to initialize variables declared in the package declaration section.
exception_handler	This exception handler is for the 'initialize_section'. It defines one or more exceptions that should be caught or handled during this initialization phase of the package.
item_list_2 **body**	The body is a pair of BEGIN and END statements which contain SQL statements.
function_definition	This specifies the syntax for a function where "**function_heading**" refers to the function declaration. It is followed by a "**declaration_section**" and a body.
procedure_definition	This specifies the syntax for a procedure where "**procedure_heading**" refers to the procedure declaration. It is followed by a "**declaration_section**" and a body.

4.3 Creating and Accessing Packages

4.3.1 Creating Packages

In the previous sections, we have gone through the syntax that dictates the structure of a package. In this section, we are going to take this a step further by understanding the construction process of a package and how we can access its public elements.

As a package is created, PostgreSQL will compile it and report any issues it may find. Once the package is successfully compiled, it is ready to use.

4.3.2 Package State

Like an object-oriented paradigm, the state of a class is identified by its instantiated object through its member variables. The same concept applies to packages too.

The state of a package is identified by its member variables, constants, or cursors. A new package instance is created for each session that accesses it.

4.3.3 Accessing Package Elements

A package is instantiated and initialized when it's referenced in a session for the first time. The following actions are executed during this process:

- Assignment of initial values to public constants and variables
- Execution of the initializer block of the package

There are several ways to access package elements:

- package functions can be utilized just as any other function in a SELECT statement or from other PL blocks
- package procedure can be invoked directly using CALL or from other PL blocks
- package variables can be directly read and written using the package name qualification in a PL block or from an SQL prompt.
- **Direct Access Using Dot Notation:**
  In the dot notation, elements can be accessed in the following manner:
    - package_name.func('foo');
    - package_name.proc('foo');
    - package_name.variable;
    - package_name.constant;
    - package_name.other_package.func('foo');

  These statements can be used from inside of PL block or in a SELECT statement if the element is not a type declaration or a procedure.

- **SQL Call Statement:**
  Another way is to use the CALL statement. The CALL statement executes a standalone procedure, or a function defined in a type or package.
    - CALL package_name.func('foo');
    - CALL package_name.proc('foo');

4.3.4 Understanding Scope of Visibility

The scope of variables declared in a PL/SQL block is limited to that block. If it has nested blocks, then it will be a global variable to the nested blocks.

Similarly, if both blocks declare the same name variable, then inside of the nested block, its own declared variable is visible and the parent one is invisible. To access the parent variable, that variable must be fully qualified.

Consider the following code snippet.

Example: Visibility and Qualifying Variable Names

```
<<blk_1>>
DECLARE
x INT;
y INT;
BEGIN
-- both blk_1.x and blk_1.y are visible

<<blk_2>>
DECLARE
x INT;
z INT;
BEGIN
-- blk_2.x, y and z are visible
-- to access blk_1.x it has to be a qualified name. blk_1.x := 0;
NULL;
END;
-- both x and y are visible
END;
```

The above example shows how you must fully qualify a variable name in case a nested package contains a variable with the same name.

Variable name qualification helps in resolving possible confusion that gets introduced by scope precedence in the following scenarios:

- Package and nested packages variables: without qualification, nested takes precedence
- Package variable and column names: without qualification, column name takes precedence
- Function or procedure variable and package variable: without qualification, package variable takes precedence.

The fields or methods in the following types need to be type qualified.

- Record Type

Example: Record Type Visibility and Access

```
DECLARE
      x INT;
      TYPE xRec IS RECORD (x char, y INT);

BEGIN
      x := 1; -- will always refer to x(INT) type.
      xRec.x := '2'; -- to refer the CHAR type, it will have to be
qualified name
```

```
END;
```

4.4 Package Examples

4.4.1 Package Specification

```
DROP TABLE log;

CREATE TABLE log( date_of_action DATE,
     user_id VARCHAR2(20),
     package_name VARCHAR2(30) );

-- Package specification:
CREATE OR REPLACE PACKAGE emp_admin AUTHID DEFINER AS
     -- Declare public type, cursor, and exception:
     TYPE EmpRecTyp IS RECORD (emp_id NUMBER, sal NUMBER);
     CURSOR desc_salary RETURN EmpRecTyp;

     invalid_salary EXCEPTION;

     -- Declare public subprograms:
     FUNCTION hire_employee (
          last_name
          first_name
          email
          phone_number VARCHAR2,
          job_id VARCHAR2,
          salary NUMBER,
          commission_pct NUMBER,
          manager_id NUMBER,
          department_id NUMBER
     ) RETURN NUMBER;

     -- Overload preceding public subprogram:
     -- PROCEDURE fire_employee (emp_id NUMBER);
     PROCEDURE fire_employee (emp_email VARCHAR2);
     PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
     FUNCTION nth_highest_salary (n NUMBER) RETURN EmpRecTyp;
END emp_admin;
```

4.4.2 Package Body

```
-- Package body:
CREATE OR REPLACE PACKAGE BODY emp_admin AS
     number_hired NUMBER; -- private variable, visible only in this package

     -- Define cursor declared in package specification:
     CURSOR desc_salary RETURN EmpRecTyp IS SELECT employee_id, salary
          FROM employees
          ORDER BY salary DESC;

     -- Define subprograms declared in package specification:
     FUNCTION hire_employee (
          last_name VARCHAR2,
          first_name VARCHAR2,
          email VARCHAR2,
          phone_number VARCHAR2,
          job_id VARCHAR2,
          salary NUMBER,
          commission_pct NUMBER,
          manager_id NUMBER,
          department_id NUMBER
     ) RETURN NUMBER IS
     new_emp_id NUMBER;

     BEGIN
          RETURN new_emp_id;
     END hire_employee;

     PROCEDURE fire_employee (emp_id NUMBER) IS
     BEGIN
          NULL;
     END fire_employee;

     PROCEDURE fire_employee (emp_email VARCHAR2) IS
     BEGIN
          NULL;
     END fire_employee;

     -- Define private function, available only inside package:
     FUNCTION sal_ok (
          jobid VARCHAR2,
          sal NUMBER
          ) RETURN BOOLEAN IS

          min_sal NUMBER;
          max_sal NUMBER; BEGIN
          RETURN true;
     END sal_ok;

     BEGIN -- initialization part of package body
     INSERT INTO log (date_of_action, user_id, package_name)
          VALUES (SYSDATE, USER, 'EMP_ADMIN');
```

```
    number_hired := 0;
    END emp_admin;
```

4.5 Limitations

Record types are supported as package variables, however they can only be used within package elements i.e., Package function/procedure can utilise them. They can not be accessed outside the package, this limitation will be addressed in the next update of HG-PSQL 2. x.

5 Orphaned Prepared Transaction Handling

5.1 Overview

Prepared transaction is one of the key features in PostgreSQL and Its intended purpose is to allow an external transaction manager to perform atomic global transactions across multiple databases or other transactional resources. Understanding this feature's offer and being able to handle any potential pitfalls are critical to maintaining a reliable system.

In a database system, a transaction is a way of processing all or zero statements in a block that generally contains more than one statement. The results of the statements in the block are not visible to other transactions until the entire block is committed. If the transaction fails, or is rolled back, it has no effect on the database.

A regular transaction is normally attached to the current session, but there are occasions where a user is required to perform a transaction that is session independent (For example, across multiple databases). This is where the "prepared transactions" feature can be utilized.

A prepared transaction is a session independent, crash resistant, state-maintained transaction. The state of the transaction is stored on disk which allows the database server to reinstate the transaction even after restarting from a crash. A prepared transaction is maintained until a rollback or a commit action is performed on it.

The PostgreSQL documentation states that a prepared transaction may be created by issuing a PREPARE TRANSACTION 'transaction_id' command within an existing transaction block; it further states that this process prepares a transaction for a two-phase commit.

A prepared transaction can be left unfinished (neither committed nor rollbacked) when the client "disappears". Events such as a client application crash, or a server crash leading to the client's disconnection and it never reconnects. It is up to the transaction manager to ensure that there are no orphaned prepared transactions.

A prepared transaction can also be left unfinished if a backup is restored that carried the preparation steps, but not the steps closing the transaction, therefore ending up with orphaned prepared transactions in the system. Perhaps a DBA creates a prepared transaction and forgets about closing it.

The issue with an orphaned prepared transaction is that it holds key system resources which may include resource locks, or keeping alive a transaction ID that may hold back vacuum from cleaning up dead tuples that are no longer visible to any other transaction except for this orphaned prepared transaction.

For this reason, it is important that there is a way of either notifying the administrators or independently handling the orphaned prepared transactions to allow for a smoother database operations.

5.2 Orphaned Prepared Transactions

An orphaned prepared transaction is defined as:

- Any prepared transaction that has exceeded a predefined age threshold for prepared transactions.

Once the age of a prepared transaction exceeds the threshold, it is assumed that it can be safely assumed that it is no longer needed.

5.3 Reporting Orphaned Prepared Transactions

The reporting functionality is activated during a vacuum process whether initiated through autovacuum or via client initiated vacuum command.

5.3.1 GUCs

5.3.1.1 prepared_xact_warn_max_age

Sets the maximum age after which a prepared transaction is considered an orphan.

Vacuum process will report any prepared transaction that has an age greater than "**prepared_xact_warn_max_age**" as an orphaned prepared transaction. The default value is -1 which disables this feature. This parameter can only be set in the **postgresql.conf** file or via the server command line.

If you are planning to use prepared transactions, this parameter may be set to a value that defines maximum age a prepared transaction can take in your environment. This parameter must be used in conjunction with "**prepared_xact_warn_min_duration**".

*"**prepared_xact_warn_max_age**" only applies when prepared transactions are enabled. The age for a transaction is calculated from the time it was created to current time. If this value is specified without units, it is taken as milliseconds.*

5.3.1.2 prepared_xact_warn_min_duration

Sets timeout after which vacuum starts throwing warnings for every prepared transaction that has exceeded maximum age defined by "**prepared_xact_warn_max_age**". If this value is specified without units, it is taken as milliseconds. The default value of -1 will disable this warning mechanism. This parameter can only be set in the **postgresql.conf** file or via the server command line.

This GUC along with "**prepared_xact_warn_max_age**", if enabled, help you better manage prepared transactions. Warnings are emitted to log when an autovacuum worker encounters orphaned prepared transactions, or to a client which has issued a vacuum command. This parameter defines how frequently a vacuum process should throw a warning when it encounters a prepared transaction with an age exceeding "**prepared_xact_warn_max_age**".

The warnings are not thrown when the vacuum command is run with relations, or when vacuumdb command is executed.

Setting a value too small could potentially fill up the log with orphaned prepared transaction warnings, so this parameter must be set to a value that is reasonably large to not fill up a log file, but small enough to notify of long running and potential orphaned prepared transactions

5.3.2 Reporting Orphaned Prepared Transactions

By default, any orphaned prepared transactions found by the background vacuum workers are reported to log file. However, when a vacuum command is run without specifying any relations, then the client will see warnings for any orphaned prepared transactions.

Example: Reporting an orphaned prepared transaction

```
postgres=# BEGIN;
BEGIN

postgres=# CREATE TABLE foo(firstCol INT);
CREATE TABLE

postgres=# INSERT INTO foo VALUES(27);
INSERT 0 1

postgres=# PREPARE TRANSACTION 'foo_insert';
PREPARE TRANSACTION

postgres=# SELECT * FROM pg_prepared_xacts;
 transaction |    gid     |           prepared            | owner  | database
-------------+------------+-------------------------------+--------+----------
       48219 | foo_insert | 2020-01-27 11:36:18.522511+00 | highgo | postgres

postgres=# vacuum;
WARNING:  prepared transaction with identifier "foo_insert" created on "2020-01-27
11:36:18.522511+00" is overage.
WARNING:  1 orphaned prepared transactions found.
```

6 Parallel Foreign Scan

6.1 Overview

Sharding in database is the ability to horizontally partition data across one more database shards. It is the mechanism to partition a table across one or more foreign servers. While the declarative partitioning feature allows users to partition tables into multiple partitioned tables living on the same database server, sharding allows tables to be partitioned in a way that the partitions live on external foreign servers and the parent table lives on the primary node where the user is creating the distributed table. The built-in sharding feature in PostgreSQL will use a FDW-based approach. FDW's are based on the SQL/MED specification that defines how an external data source can be accessed. PostgreSQL provides a number of foreign data wrappers (FDW's) that are used for accessing external data sources. Using the FDW-based sharding, the data is partitioned to the shards in order to optimize the query for the sharded table. Various parts of the query e.g., aggregates, joins, are pushed down to the shards. This enables the heavy query processing to be done on the shards and only results of the query are sent back to the primary node.

When a query is querying multiple foreign scans in a single query, all the foreign scans are being executed in a sequential manner, one after another. Parallel foreign scan functionality will allow executing multiple foreign scans in parallel. This feature is particularly important for the OLAP use cases, for example if you are running a query on large volume sharded table with multiple partitions residing on multiple foreign servers, the query will be sent to each foreign server sequentially and results from each server are sent to the parent node. The important point to note is that without parallel foreign scan feature, each sub-partition will be scanned sequentially and with this feature all the sub-partitions residing on different foreign servers will be scanned in parallel and result will be sent in parallel to parent node. The parent node will process the data from all the foreign server nodes and sent the results back to the client.

HG-PGSQL 2.1 organizes sharding structure in the way of external partition tables. Before adding Parallel Foreign Scan function, Append node is used in the code to maintain the structural relationship between multiple external partition tables. When executing Append node, the child nodes of each Append node will be processed in turn. Suppose partition table A has three foreign sub-partitions A1,A2 and A3. When querying Table A, the executor will scan A1 and return data first, then scan A2 and return data, and then scan A3 and return data.

This function changes the execution logic of the append node, sends a scan request to the three foreign subpartitions, and then polls to receive the data returned by the three nodes. This can greatly improve query performance with less data returned.

6.2 Usage

This feature is transparent to the users, they can use the parallel foreign scan function without setting any configuration parameter. The users can view the foreign scan getting executed in parallel using the EXPLAIN plan command.

For example we have this partition:

```
postgres=# \d+ t3
                        Partitioned table "public.t3"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 i      | integer |           |          |         | plain   |              |
 j      | integer |           |          |         | plain   |              |
Partition key: RANGE (i)
Partitions: t3_p1_f FOR VALUES FROM (0) TO (4),
            t3_p2_f FOR VALUES FROM (4) TO (8)
```

The execution plan without parallel foreign scan feature:

```
postgres=# explain select * from t3;
                          QUERY PLAN
-------------------------------------------------------------------------------
 Append  (cost=100.00..399.20 rows=5120 width=8)
   ->  Foreign Scan on t3_p1_f t3_1  (cost=100.00..186.80 rows=2560 width=8)
   ->  Foreign Scan on t3_p2_f t3_2  (cost=100.00..186.80 rows=2560 width=8)
```

The execution plan with parallel foreign scan feature:

```
postgres=# explain select * from t3;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Append  (cost=0.00..223.70 rows=7380 width=8)
   Async subplans: 2
   ->  Async Foreign Scan on t3_p1_f t3_1  (cost=100.00..186.80 rows=2560 width=8)
   ->  Async Foreign Scan on t3_p2_f t3_2  (cost=100.00..186.80 rows=2560 width=8)
```

6.3 Limitation

While this feature is clearly very useful for OLAP use-cases, there is a limitation to this functionality. Once the data is returned by the sub-nodes to the main parent node, the overhead of processing all the results is shifted to the parent node which can impact the performance of the parent node. However for OLAP queries, the performance will increase significantly as the sub-nodes will process the query operation in parallel and result will be sent to the parent node which will process the result and send it to the client.