Pgxc_fdw Implementation Plan

July, 2025
Koichi Suzuki

1. Introduction

Postgres-XC has been developed and released in 2010, by NTT Open Source (OSSC) to provide a scale-out solution of PostgreSQL database cluster.

Outline of this product was presented in PGCon 2010 and 2012. First presentation information is in this PostgreSQL wiki. Presentation audio is available here. Postgre-XC features and details were presented in a PGCon tutorial in 2012. Presentation material is available here.

At first, Postgres-XC (PGXC hereafter) was based on PostgreSQL 9.2 and was a spinoff of PostgreSQL, meaning that PGXC is a source code level modification of PostgreSQL.

Achievement of PGXC is as follows:

- Provide global consistent visibility for global distributed transactions, avoiding read anomaly caused by different timing of commit/abort.
- Good scale-out for short transactions.

Drawbacks of PGXC are:

- Source code modification of PostgreSQL: PGXC source needs to migrate to PostgreSQL source code upgrade to accommodate major PostgreSQL upgrades.
- Global transaction management was done in a centralized manner. It fits to datacenter level distribution but does not to geo-distributed use case.

These drawbacks made maintenance and enhancement of PGXC very challenging. This is the main reason why PGXC development could not continue for a long time.

Now PostgreSQL provides much more infrastructures such as FDW and parallel execution, which PGXC cannot provide (PGXL, spinoff of PGXC, provide parallel query capability).

2. Key PGXC concepts to achieve good scalability

PGXC allow two types of tables:

2.1 Partitioned table

Partitioned tables are known as “shards” in distributed databases. A single table, called parent table, is divided into multiple partitions to accommodate different rows, based on the constraints defined on each partition. Such partitions can be placed in different databases so that some of the operations on the parent table can be pushed down to the database accommodating the partition and some of them can be executed in parallel, in different databases.

This works to achieve global scalability.

2.2 Replicated table

Replicated table is a table whose copy is accommodated in each remote (and local, possibly) database. This type of table is introduced to increase a change of pushdown of database operation, mainly join.

If we have only partitioned tables, we can push down join operation only when the join key is the partitioning key of both partitions.

Replicated table increases the chance of join pushdown. We can select tables to be replicated which appear many times in join operation with various join keys and which do not have relatively very frequent writes.

With a replicated table, any join between a partitioned table and a replicated table can be pushed down. This significantly enhances the chance of join pushdown and is very essential for pgxc to achieve good scalability.

2.3 Writes to the replicated table

If we allow arbitrary write operations to each replica of a replicated table, data in this table becomes inconsistent very easily even though each writing transaction tries to write consistent data to each replica.

We need a solution to make writes to such replicated tables consistent even when multiple conflicting writes are initiated by different transactions.

PGXC resolved this by introducing a “preferred” node. Preferred node is a database where writes to a replicated table should go first. Writes to other nodes can be done in the same transaction in arbitrary order or in parallel, after the first write to the preferred node is successful.

This protocol is very simple and provides serializable writes to replicated tables successfully. With this protocol, non-conflicting writes can be done in parallel.

3. Component of PGXC

PGXC consists of the following components.

3.1 Query processor

Query processor is a modified query planner and executor to handle distributed operations among partitioned tables and replicated tables.

3.2 Connection pooler

This provides connection pooling to remote databases (server, in fdw) to save overheads to initiate remote sessions.

3.3 Global deadlock detector

In pgxc_fdw, because many transactions are involved with writes to remote databases, there is some risk that such transactions suffer from global deadlock. We may need to implement a deadlock detector for such transactions.

3.4 Global transaction manager (GTM)

This provides consistent transaction management over all the databases in the cluster. This includes providing consistent visibility of the database writes, avoiding read anomaly in distributed transactions.

4. Implementation priority

It is not necessary to implement all the components as shown in the last chapter. First priority is the query processor. This is the central component of the pgxc_fdw feature. As described later, the latest PostgreSQL infrastructure enables the implementation of pgxc_fdw as FDW driver, with some extensions.

Next priority is connection pooler. This is another key component to provide good performance fundamentals for pgxc_fdw distributed transactions. It is apparent that we can implement this as another PostgreSQL extension.

The third priority will be the global deadlock detector. With current PostgreSQL infrastructure, we need to modify the base PostgreSQL code for this capability, especially inside the deadlock.c source module among others.

Last priority will be global transaction manager. Global transaction manager is useful to avoid read anomalies in read transactions. We should note that we can provide write consistency with 2PC protocol and the impact of such read anomalies can be minor. To implement GTM, we need some extension to the current PostgreSQL, as shown later.

5. Query processor basics/design and implementation

In this chapter, we will present the basic ideas/design and implementation of the query processor, including partitioned tables and replicated tables. Schematic idea about DDL and ideas about fdw callback functions will also given here.

5.1 General idea

The general idea is to implement “server” as a group of databases. For example, when we have four databases, DB1, DB2, DB3 and DB4 for pgxc_fdw database cluster, we can create a server, namely, pgxc_server to represent a group of databases, db1 to db4. Then, using OPTIONS property of the table, we can specify if the table is replicated or partitioned and which server they actually located, including the local name of these tables.

Please note that even a local tables can be included in this group. In this case, such individual server can be marked as “local”. In this case, current session will be used to issue partial statement as called from core FDW infrastructure. In this case, such “local” replica/partition is handed as FDW resource and pgxc_fdw handles this locally.

To avoid table name duplication, FDW table names need to be different from the actual local table name. We can handle remote tables in the same manner so that we can use the same DDL as much as possible in different local databases, for example, in db1 to db4 as above.

5.2 Partitioned table

Parent tables are defined as local tables and remote tables are added as their partitions. In each remote table definition, the server should be the same and in OPTIONS property, actual location and local table name are given so that the pgxc_fdw driver can handle access to them properly.

5.3 Replicated table

A replicated table is defined as a single table in CREATE FOREIGN TABLE. In OPTIONS property, we specify the set of actual remote tables. Each remote table is defined using another DDL.

5.4 Schematic DDLs

This section gives a general idea how pgxc_fdw resources can be defined. Please note that example statements just illustrates general idea and are not intended to be the final syntax/semantics.

In the following subsections, we assume the following:

- Foreign data wrapper name: `pgxc_fdw`
- Name of representative server: `cluster1`
- Names of individual servers: `db1, db2, db3, db4`

- Individual server for the local host: `db1`
  Please note that we need to run similar DDL in db2, db3 and db4 if we need to run pgxc_fdw queries at these servers.
- User of pgxc_fdw: `foo`
- Partitioned table name: `table_a`
- FDW name of each partition of `table_a: rtab_a_part1, rtab_a_part2, rtab_a_part3, rtab_a_part4`
  Each located at db1 to db4 respectively.
- Actual name of each partition: `table_a_part1, table_a_part2, table_a_part3, table_a_part4`
  Each for `rtab_a_part`*x* respectively.
- FDW name of a replicated table: `table_b`
- FDW name of each replica of table_b: `rtab_b_db1, rtab_b_db2, rtab_b_db3, rtab_b_db4`
  Each located at db1 to db4 respectively.
- Actual name of each replica: `rtab_b`

5.4.1 Servers and user mappings

As described above, a server in pgxc_fdw can be a group of actual servers. From PG core, such a representative server is regarded as a "single" server so that pgxc_fdw can produce its own plan and run it.

The following is a schematic DDL to define representative and actual servers in pgxc_fdw.

Please note that we can combine different versions of postgres, or even different databases for partition and replica.

```
/* Individual Servers */

/* Please note that host1 is local server and db1 server is marked as local */

CREATE SERVER db1 TYPE 'postgres' VERSION '17'
    FOREIGN DATA WRAPPER pgxc_fdw
    OPTIONS (host 'host1', user 'foo', port '5432', local ‘true’);

CREATE SERVER db2 TYPE 'postgres' VERSION '17'
    FOREIGN DATA WRAPPER pgxc_fdw
    OPTIONS (host 'host2', user 'foo', port '5432', local ‘false’);

CREATE SERVER db3 TYPE 'postgres' VERSION '17'
    FOREIGN DATA WRAPPER pgxc_fdw
    OPTIONS (host 'host3', user 'foo', port '5432', local ‘false’);

CREATE SERVER db4 TYPE 'postgres' VERSION '17'
    FOREIGN DATA WRAPPER pgxc_fdw
    OPTIONS (host 'host4', user 'foo', port '5432', local ‘false’);

/* Representative Server */

CREATE SERVER cluster1 TYPE 'group'
    FOREIGN DATA WRAPPER pgxc_fdw
    OPTIONS(servers 'db1 db2 db3 db4',
            /* Pooler can be optional (separate implementation and extension) */
            pooler 'cluster1_pooler',           /* pooler name */
            pool_init '1',                      /* pooler configuratin */
            pool_max  '10',
            pool_limit '15',
            session_string 'SET ....; SET ...;');
```

```
/* User Mapping */

CREATE USER MAPPING FOR USER foo
    SERVER db1;
CREATE USER MAPPING FOR USER foo
    SERVER db2;
CREATE USER MAPPING FOR USER foo
    SERVER db3;
CREATE USER MAPPING FOR USER foo
    SERVER host4;
CREATE USER MAPPING FOR USER foo
    SERVER db4;
```

5.4.2 Partitioned Table

```
/* Parent table */

CREATE TABLE tableA
    (column ....)
    PARTITIONED BY ...;

/* Each partition */

CREATE FOREIGN TABLE rtab_a_part1
    (columns ...)
    PARTITION OF TableA (... partitioning details ...)
    SERVER cluster1
    OPTIONS (server 'host1', partition 'true', server 'db1', real_name 'table_a_part1');

CREATE FOREIGN TABLE rtab_a_part2
    (columns ...)
    PARTITION OF TableA (... partitioning details ...)
    SERVER cluster1
    OPTIONS (server 'host2', partition 'true', server 'db2', real_name 'table_a_part2');

CREATE FOREIGN TABLE rtab_a_part3
    (columns ...)
    PARTITION OF TableA (... partitioning details ...)
    SERVER cluster1
    OPTIONS (server 'host3', partition 'true', server 'db3', real_name 'table_a_part3');

CREATE FOREIGN TABLE rtab_a_part4
    (columns ...)
    PARTITION OF TableA (... partitioning details ...)
    SERVER cluster1
    OPTIONS (server 'host4', partition 'true', server 'db4', real_name 'table_a_part4');
```

5.4.3 Replicated Table

```
/* Each Replica */

CREATE FOREIGN TABLE rtab_b_db1
    (column ....)
    SERVER db1
    OPTIONS (repl 'true', is_parent 'false', real_name 'rtab_b');

CREATE FOREIGN TABLE rtab_b_db2
    (column ....)
    SERVER db2
    OPTIONS (repl 'true', is_parent 'false', real_name 'rtab_b');

CREATE FOREIGN TABLE rtab_b_db3
    (column ....)
    SERVER db3
    OPTIONS (repl 'true', is_parent 'false', real_name 'rtab_b');

CREATE FOREIGN TABLE rtab_b_db4
    (column ....)
    SERVER db4
    OPTIONS (repl 'true', is_parent 'false', real_name 'rtab_b');

/* Parent Table */
/* If we define this at different server too, we need to configure same preferred property */

CREATE FOREIGN TABLE tabB
    (column ...)
    SERVER  cluster1
    OPTIONS(repl 'true',
             is_parent 'true',
            elements 'rtab_b_db1 rtab_b_db2 rtab_b_db3 rtab_b_db4'
            preferred 'rtab_b_db1');
```

5.5 General Remarks

- With the above definition, all the partitions and (virtually single) replicated tables are regarded to be in the server `cluster1`.
- `Cluster1` is actually a group of servers. Pgxc_fdw understands this and it can tell the core about join/aggregate pushdown and other operations, where to go.
- Also pgxc_fdw can manage writes to the replicated table, to write to all the nodes, considering the preferred node as well.
- In the callback functions, we should handle the server as representative one and use actual server to return the requested information and to do actual operation.
- We may be able to (and should) have dedicated catalog for pgxc_fdw for performance and maintenance. We need to implement a validator for this, to check OPTIONS properties and maintain pgsc_fdw catalog table.

6. Connection pooler

Connection pooler can be implemented as a separate extension. Connection pooler provides various functions for pgxc_fdw to initialize/attach/release/finalize connection. As given in 5.4.1, connection pooler can belong to a representative server. Multiple representative servers can share a connection pooler if session parameters and other connection properties are shared among them.

Connection pooler is database specific (not a global object) and is identified by its name. It can have the following properties but the list is not comprehensive.

- pooler name,
- representative server to use this pooler,
- initial number of connections per non-representative server,
- max number of connections per non-representative server,

- limit number of connections per non-representative server. If connection is needed beyond max number, pooler can create more connections for temporary use, Once this is released, then the pooler will terminate the connection to save system resource.

Please note that connection pooler on one database can have more than one connection pooler, identified by its name. Assignment of the pooler to the representative server can be done in CREATE SERVER.

Connection pooler should provide functions, among others:

- Attach a connection to a given (non-prepresentative) server,
- Detach a connection.
- Start and stop pooler.

7. Global transaction manager

As stated in many external materials, two-phase commit protocol is the basic infrastructure to provide transaction write consistency for distributed transactions. However, because of the difference of the timestamp of commit/abort of each transaction, we have a chance that visibility of such transaction results is different from database to database, which is called read anomalies.

We need a separate infrastructure to avoid such read anomalies. Various different kinds of algorithms are proposed for this and pgxc_fdw can choose one of such algorithms. In PGXC/PGXL, we modified the database core source and this is not a good idea, considering the ease of code merge and getting along with PostgreSQL upgrades.

Here's some idea to allow external/individual global transaction managers as PostgreSQL extensions.

- Have a new hook so that we can replace `TransactionIdGetStatus()` function in `transam.c` source module. This function returns the status of given XID. To avoid read anomalies, we need to look into the consistent status of a given distributed transaction. The backend is that even though the local transaction has been committed/aborted, other pieces of transactions consisting of the global transaction might be running. In this case, we need to return the status as **running**.
- New event trigger for each transaction statement: commit/abort/prepare transaction/savepoint/release savepoint/rollback savepoint.
  - These event triggers can be replaced with corresponding hooks.

8. Global deadlock detector

Global deadlock detector has already been implemented and is available in the git hub as a folk from PostgreSQL 14. We need to port this to the latest version of the PostgreSQL and (hopefully) make this a part of the core code.

Presentation material is available here.

[End of the document]