Postgres-XC Concept, Implementation and Achievements

Koichi Suzuki and Masataka Saito

September 10^{th}, 2014

Contents

Preface

Postgres-XC research and development work started in NTT DATA Corporation at around the year of 2002, as RiTaDB, to achieve horizontally scalable database system based on shared-nothing architecture.

Postgres-XC implemented transparent global transaction management from the very beginning, but did not have consistent updating capability for replicated tables.

After several years, RiTaDB was renamed to Postgres-XC as separate open source project derived from the latest version of PostgreSQL. Major achievements of Postgres-XC include horizontally-scalable PostgreSQL database cluster based on shared-nothing architecture, global transaction management over the whole cluster, table sharding and replication, query planner and executor to utilize parallerizm among cluster nodes, and so on.

This book is a summary of the latest Postgres-XC implementation and its achievements.

Acknowledgements

Authors of this book would like to express special thanks to contributors to Postgres-XC implementation. Among them, the following contributors work should be noted.

Mason Sharp

Wrote initial version of backend global transaction management, planner and executor.

Pavan Deolasee

Wrote initial GTM and GTM proxy.

Andrei Martsinchyk

Wrote initial version of the pooler.

Abbas Butt

Wrote online node addition/removal. Also wrote many extensions including `RETURNING`, `LATERAL` and `CURSOR`.

Ashutosh Bapat

Wrote most of the planner extension.

Amit Khandekar

Wrote `TRIGGER`.

Ahsan Hadi

Provided excellent project management in India and Pakistan side.

Michael Paquier

Wrote various code including DDL propagation, DB2 and pgbench benchmark, among others. He also contributed to clean up many bugs and codes.

Takayuki Sudo

Built and maintained the buildfarm environment for Postgres-XC development, test and evaluation.

Nikhil Sontakke

Contributed many GTM-related fixes.

Tetsuo Sakata

Provided initial technical input to the project.

Hitoshi Hemmi

Provided technical input and requirements to the project.

Masaki Hisada

Provided technical input and requirements to the project.

Change History

September 10th, 2014 Initial Document Release

Part I

Postgres-XC Implementation

Chapter 1

Postgres-XC Architecture

1.1 What is Postgres-XC?

Postgres-XC is an open source project to provide horizontal scalability including write-scalability, synchronous multi-master, and transparent PostgreSQL interface. It is a collection of tightly coupled database components which can be installed in more than one hardware or virtual machines.

Write-scalability means Postgres-XC can be configured with as many database servers as you want and handle many more writes (updating SQL statements) than single database server can handle. Multi-master means you can have more than one data base servers which provides single database view. Synchronous means any database update from any database server is immediately visible to any other transactions running in different masters. Transparent means you don't have to worry about how your data is store in more than one database servers internally[1].

You can configure Postgres-XC to run on more than one physical or virtual servers. They store your data in a distributed way, that is, you can configure each table to be either partitioned or replicated[2]. When you issue queries, Postgres-XC determines where the target data is stored and issue appropriate SQL statements to servers which have the target data. This is shown in Figure 1.1.

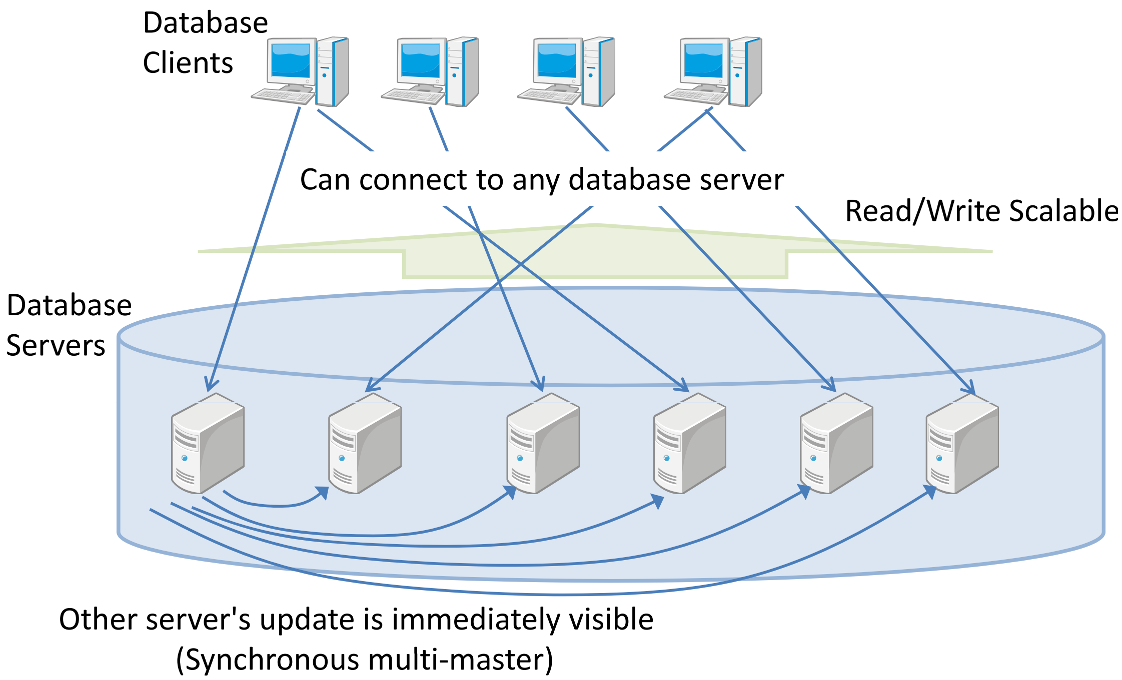

Figure 1.1: Postgres-XC's synchronous multi-master configuration

In typical web applications, you can use any number of web servers or application servers to handle your transactions. In general, you cannot do this for a database server because all the

[1] Of course, in order to get the most from Postgres-XC, you should consider when designing your schema where the table data will be physically stored.

[2] To distinguish from PostgreSQL's partitioning, we call this "**distributed**". In distributed database textbooks, this is often referred to as a "horizontal fragment".

updates to the database have to be visible to all the transactions. Unlike other database cluster solution, Postgres-XC provides this capability. You can install as many database servers as you like. Each database server provides uniform data view to your applications. Any database update from any server is immediately visible to applications connecting the database from other servers. This feature is called "synchronous multi master" capability and this is the most significant feature of Postgres-XC, as illustrated in Figure 1.1.

Postgres-XC is based upon PostgreSQL database system and uses most of existing modules including interface to applications, parser, rewriter, planner and executor. In this way, Postgres-XC's application interface is compatible to existing PostgreSQL. (As described later, at present, Postgres-XC has some restrictions to SQL statements, mainly because of the distributed nature of the architecture. This will be improved in the future).

1.2 Postgres-XC's Goal

Ultimate goal of Postgres-XC is to provide synchronous multi-master PostgreSQL cluster with read/write scalability. That is, Postgres-XC should provide the following features:

Postgres-XC should allow multiple servers to accept transactions and statements from applications, which is know as "master" server in general. In Postgres-XC, these components are called "**coordinator**".

Any coordinator should provide consistent database view to applications. Any updates from any master must be visible in real time manner as if such updates are done in single PostgreSQL server.

Tables should be able to be stored in the database in replicated or distributed way (known as fragment, shard or partition). Replication and distribution should be transparent to applications, that is, such replicated and distributed table are seen as single table and location or number of copies of each record/tuple is managed by Postgres-XC and is not visible to applications.

Postgres-XC should provide compatible PostgreSQL API to applications.

Postgres-XC should provide single and unified view of underlying PostgreSQL database servers so that SQL statements does not depend on how tables are stored in distributed way.

So far, Postgres-XC's achievements are as follows:

1. Transaction management is almost complete. PostgreSQL provides complete "Read Committed" transaction isolation level which behaves exactly the same as single PostgreSQL server. Repeatable read, serializable and savepoint from the client should be added in the future.

2. Major statement features are available. Some features, such as full cursor feature including `WHERE CURRENT OF`, full constraint support in distributed table and savepoint are not supported. Background of some of them is the nature of table distribution and replication.

1.3 How To Scale Out Both Reads And Writes?

Simply put, parallelism is the key of the scalability. For parallelism, transaction control is the key technology.

We'll compare PostgreSQL's transaction control with conventional replication clusters and show how Postgres-XC is safe to run update transactions in multiple nodes first, then shows major Postgres-XC components, and will finally show how to design the database to run transactions in parallel.

1.3.1 Parallelism In Postgres-XC

Parallelism is the key to achieve write scalability in Postgres-XC.

Internally, Postgres-XC analyzes incoming SQL statement and chooses which server can handle it. It is done by a component called "**coordinator**." Actual statement processing is done by a component called "**datanode**." In typical transactional applications, each transaction reads/writes small number of tuples and lots of transactions has to be handled. In this situation, we can design the database so that one or a few datanodes are involved in handling each statement.

In this way, as seen in Figure 1.2, statements are handled in parallel by Postgres-XC servers, which scales transaction throughput. As presented later in this document, with ten servers, the total throughput could be 6.4 compared with single server PostgreSQL. Please note that this is accomplished using conventional DBT-1 benchmark, which includes both read and write operation. Figure 1.2 shows that present Postgres-XC is suitable for transactional use case as described in PostgreSQL Wiki page. By improving supported SQL statements, we expect that Postgres-XC can be suitable for analytic use case.

1.3.2 Star Schema

There's a typical database schema structure called star schema[3]. It is found in many data warehouse and OLTP applications. Star schema consists of a few and big "fact" tables and many smaller "dimension" tables. For example, sales database may include "sales fact" as a fact table, as well as "product dimension" and "store dimension" table as dimension tables. Fact tables are big in size and updated frequently. On the other hand, dimension tables are small in size and more stable compared with fact tables. Figure 1.3 shows typical star schema[4].

Postgres-XC architecture is build to leverage star schema characteristics. Usually, if there is more than one fact tables, they tend to share candidate keys. In Postgres-XC, it is desirable to shard fact tables using one of such common candidate key. In this way, we cay shard one (or few) big table into smaller pieces and store them in different server (datanode). The column used to determine what datanode each row goes is called a **distribution key**.

Then updates by multiple transactions can be performed in more than one datanode in parallel.

[3] `http://www.ciobriefings.com/Publications/WhitePapers/DesigningtheStarSchemaDatabase/tabid/101/Default.aspx` is a typical star schema description. `https://www.youtube.com/watch?v=q77B-G8CA24` is a tutorial how to design start schema.

[4] The chart was taken from `http://support.sas.com/documentation/cdl/en/spdsug/64018/HTML/default/viewer.htm#n0mlj75x9c4dtzn1ves84e1op3jt.htm`

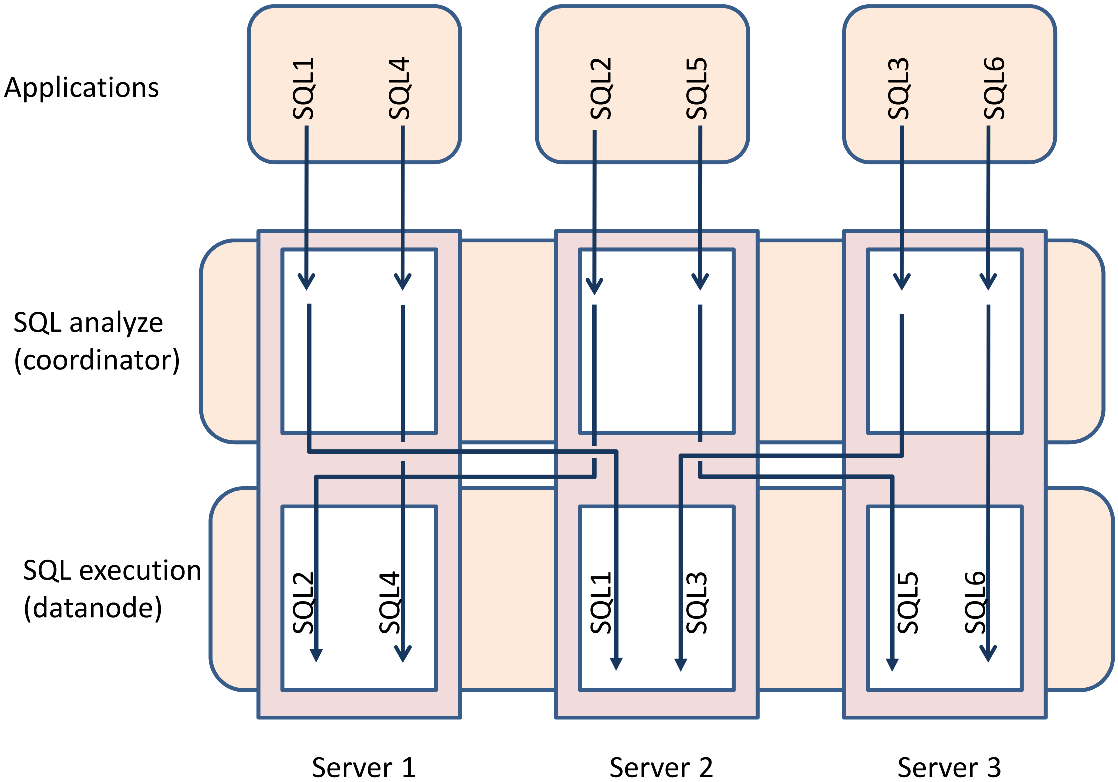

Figure 1.2: Postgres-XC can handle statements in parallel in multiple datanodes.

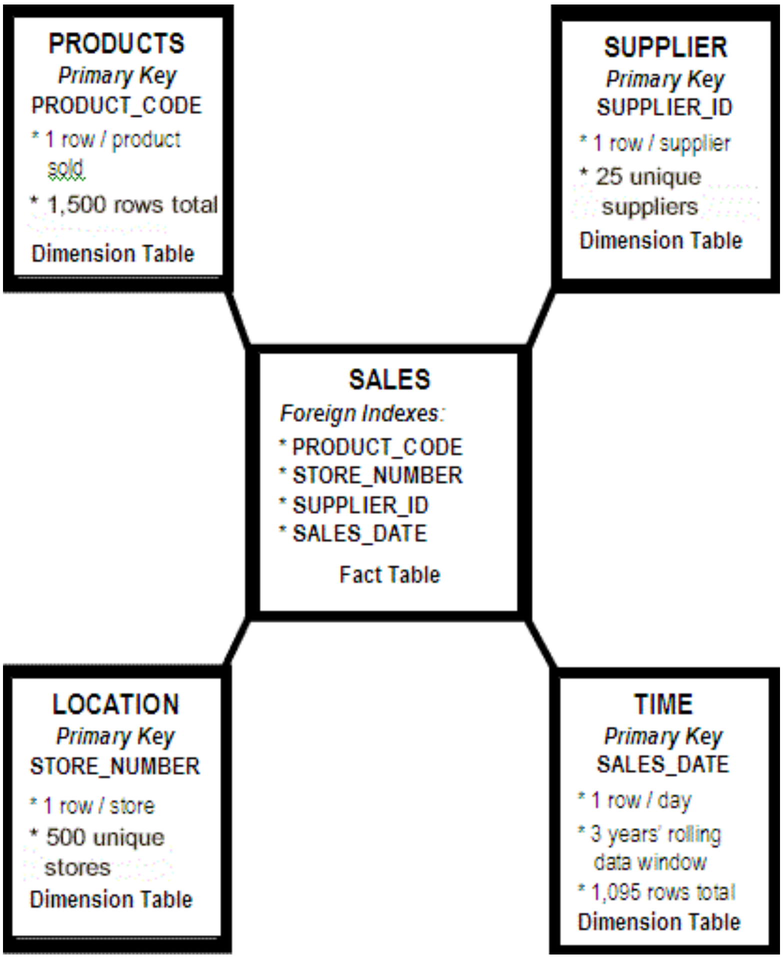

Figure 1.3: Typical star schema (See above for the source)

With more datanode, we can run more updates to fact tables in parallel. This is basically the background that Postgres-XC provides write scalability. Figure 1.4 illustrates this.

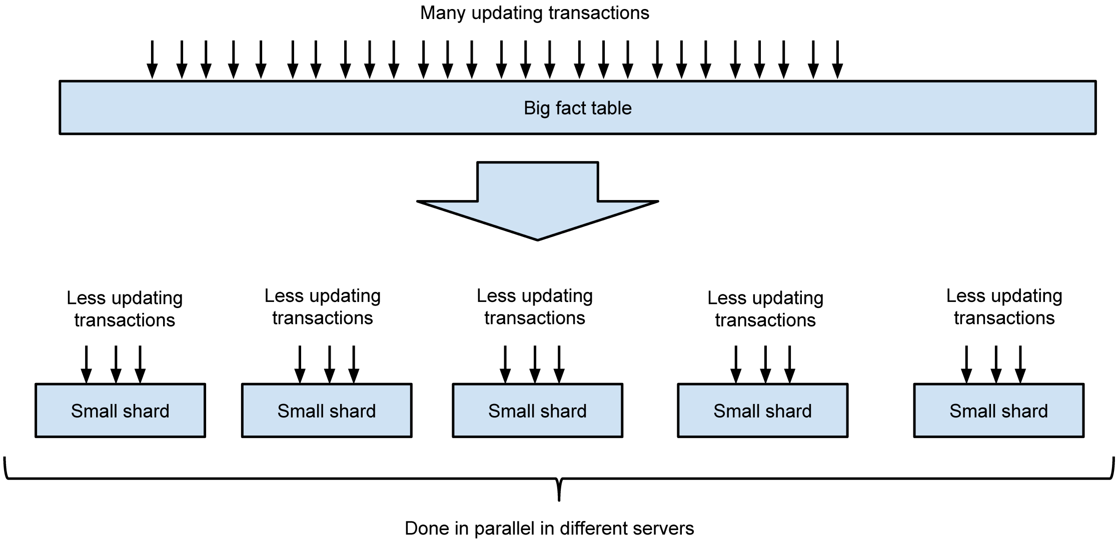

Figure 1.4: Write scalability in Postgres-XC

As shown in Figure 1.5, we replicate all the dimension tables to all the datanodes. Because most of the joins are done between a fact table and dimension tables, or among fact tables with distribution key involved, we can convert a big join to a union of smaller joins between each shard and replicated tables performed locally in each datanode in parallel. This is how Postgres-XC provides read scalability.

If a statement has additional predicates in `WHERE` clause which help to locate a datanode where the target rows are stored, then Postgres-XC can select only a few of datanode to perform such a query. This is found din many of OLTP workloads.

Figure 1.6 illustrates this.

There could be exceptional case where an application needs a join between fact tables without distribution key involved. In this case, Postgres-XC pushes down as many operation as possible to each datanode and performs final join operation at the top level (coordinator).

In other words, if an application cannot utilize this star schema, it is not suited to Postgres-XC.

1.3.3 Replicated table update: Primary node

Because of the delay in log-shipping replication, it is quite challenging to enforce consistent visibility and it is not practical to use log-shipping in replicated table update.

To enforce data integrity in replicated tables, Postgres-XC uses a technique called "**primary node**."

It is done in the following steps.

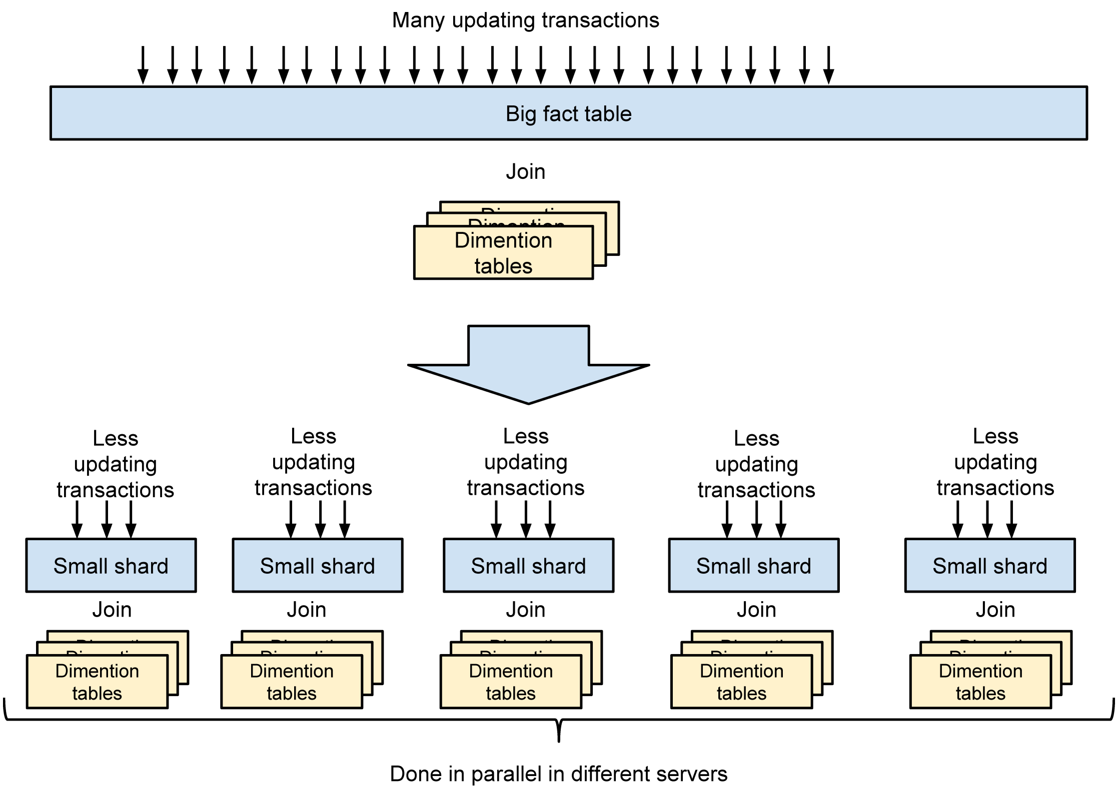

Figure 1.5: Decomposing big statement into smaller shards

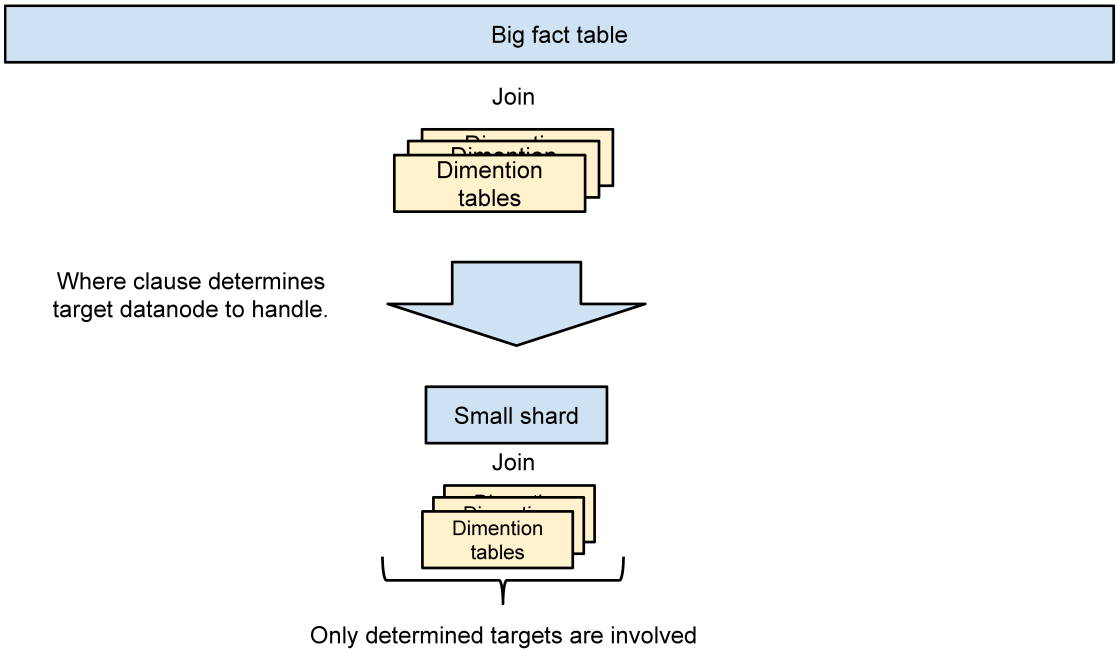

Figure 1.6: Statement can be optimized more if `WHERE` clause determines the target

1. Assign specific datanode as "**primary node**".
2. Any write to replicated tables should go to the primary node first.
3. If there's any conflicting update, such update will be blocked at the primary node and conflicting update does not propagate to other datanode until current updating transaction is committed or aborted.

Please note that this works with statement-based replication. This technique is similar to that used in pgpool-II's parallel mode.

1.3.4 DDL propagation

In Postgres-XC, most DDL should propagate to other coordinator and datanode as well. Exception is for node management DDL, such as `CREATE NODE` and `ALTER NODE`. Node management DDL should run before each coordinator/datanode knows each other, automatic propagation was determined not practical.

This restriction is not from the architecture but just an implementation issue. In the future, there could be an extension that the node management DDL propagates to other node automatically.

1.3.5 System catalog for shard and replica

In Postgres-XC, each table can be defined as "distributed" or "replicated" with `CREATE TABLE` [5]. and `ALTER TABLE` [6] statements Distributed tables correspond to fact tables and replicated tables correspond to dimension tables in the star schema respectively. A distributed table is divided into shards using distribution key and stored in nodes as specified. You can specify how to locate each row, by hash, modulo or round-robin. A replicated table is copied to specified set of nodes and its content is maintained to be logically equivalent.

Postgres-XC uses additional system catalog `pgxc_class`[7] to store sharding and replication information of each table. This can be an extension to `pg_class`. Postgres-XC chose to have this in a separate catalog so that changes in PostgreSQL and Postgres-XC can be maintained as separately as possible.

1.3.6 Limitations coming from sharding and replication

As described in section 1.3.7, Postgres-XC uses SQL statement to instruct other nodes to do something. Because of this, Postgres-XC has following restrictions:

1. Oid value may be different from node to node. For example, you should not expect `pg_class` entry OID and other OID value in system catalogs are the same across the node. If you create a replicated table with OID, OID value will be different from node to node.
2. In replicated tables, CTIDs of given rows may be different from node to node.
3. Each shard of a distributed table has similar characteristics as inherited tables in PostgreSQL. Constraints across the shard is not simply supported. At present, Postgres-XC does not support unique index in a distributed table if the distribution column is not involved in index columns. For the same reason, referential integrity between distributed tables are not supported unless it is guaranteed to be maintained locally.

Restrictions and remarks for specific SQL statement will be given in a separate material.

1.3.7 Postgres-XC's Global Transaction Management

This section describes how transaction update and visibility are enforced in Postgres-XC. You may need to be familiar with internal of PostgreSQL transaction management infrastructure such as `XID`, snapshot, `xmin` and `xmax`. This information is found in "MVCC revealed" available at `http://momjian.us/main/writings/pgsql/mvcc.pdf`

In replication clusters, you can run read transactions in parallel in multiple standby, or slave servers. Replication servers provide read scalability. However, you cannot issue write transactions to standby servers because they don't have means to propagate changes in slaves They cannot maintain consistent view of database to applications for write operations, unless you issue write transactions to single master server.

[5]See `http://postgres-xc.sourceforge.net/docs/1_2_1/sql-createtable.html` for details
[6]See `http://postgres-xc.sourceforge.net/docs/1_2_1/sql-altertable.html` for details.
[7]See `http://postgres-xc.sourceforge.net/docs/1_2_1/catalog-pgxc-class.html` for details.

Postgres-XC is different.

Postgres-XC is equipped with global transaction management capability which provides cluster-wide transaction ordering and cluster-wide transaction status to transactions running on the coordinator (master) and the node which really stores the target data and runs statements, called datanode. This maintains ACID property for distributed transaction and provide atomic visibility[8] to transactions reading more than one node.

1.3.8 Statement based replication and sharding

At present, Postgres-XC sends SQL statements to other nodes to read and write tables. There are many discussions if it is a right choice, or if we should use internal plan tree to transfer to other node.

An internal analysis of dynamic behavior of PostgreSQL shows that around 30% of CPU resource is consumed to parse and plan a statement for typical OLTP workloads. Saving this resource looks nice. On the other hand, serialized plan tree can be very big, which suffers network workload. We also need to maintain all the internal information such as Oids and ctids throughout Postgres-XC cluster, which are not simple. They are the main reason why Postgres-XC chose to send statement from node to node.

Because of this and parallelism of transactions and statements, Postgres-XC does not maintain Oids and ctids of each object and row.

1.4 Postgres-XC Key Components

In this section, major components of Postgres-XC will be described.

Postgres-XC is composed of three major components, called **GTM** (Global Transaction Manager), **Coordinator** and **Datanode** as shown in Figure 1.7. Their features are given in the following sections.

Figure 1.8 outlines how each key component interacts.

1.4.1 GTM (Global Transaction Manager)

GTM is a key component of Postgres-XC to provide consistent transaction management and tuple visibility control. First, we will give how PostgreSQL manages transactions and database update.

[8] "Scalable Atomic Visibility with RAMP Transactions," SIGMOD'14, June 22 – 27, 2014, Snowbird, UT, USA, `http://www.bailis.org/papers/ramp-sigmod2014.pdf`

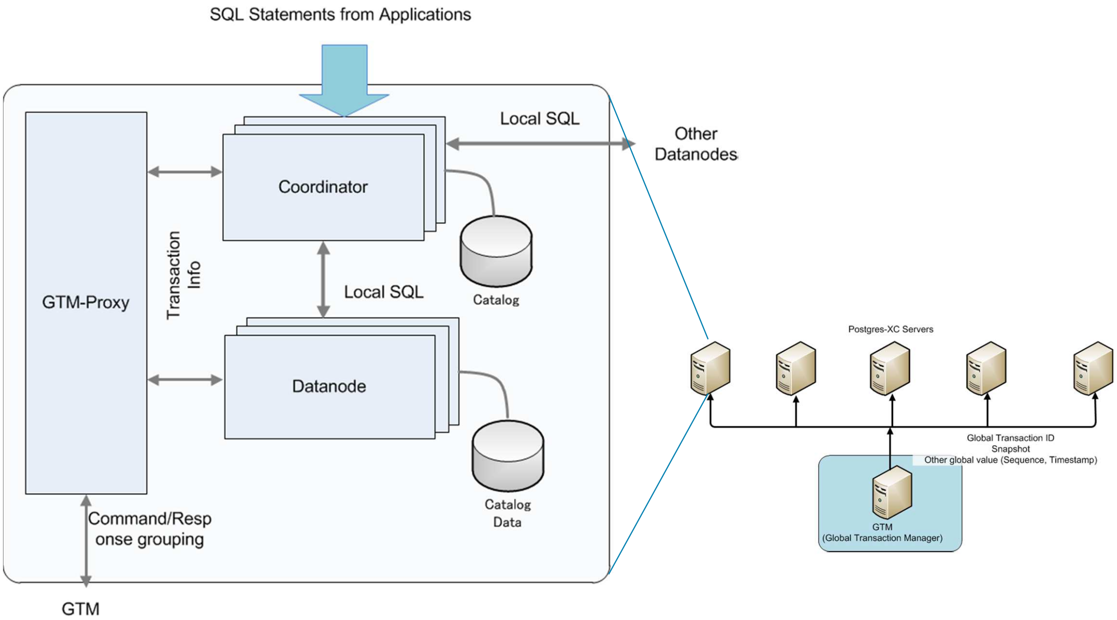

Figure 1.7: Postgres-XC Key Components

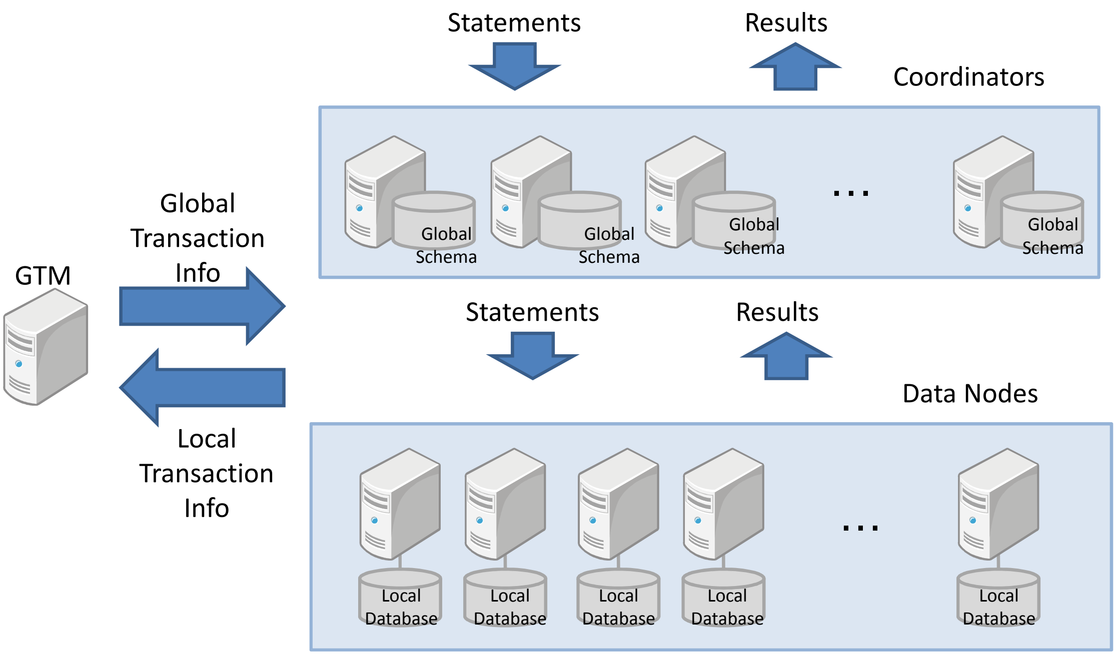

Figure 1.8: Interaction between Postgres-XC key components

1.4.1.1 How PostgreSQL Manages Transactions

In PostgreSQL, each transaction is given unique ID called transaction ID (or XID). XID is given in ascending order to determine which transaction is older/newer[9]. Please let us describe a little in detail how it is done[10].

When a transaction tries to read a tuple, each tuple has a set of XIDs to indicate transactions which created and deleted the tuple. So if the target tuple is created by an active transaction, it is not committed or aborted and reading transaction should ignore such tuple. In such way (in practice, this is done by `tqual.c` module in PostgreSQL core), if we give each transaction a unique transaction Id throughout the system and maintain snapshot what transaction is active, not only in a single server but transaction in all the servers, we can maintain global consistent visibility of each tuple even when a server accepts new statement from other transactions running only on other servers.

These information is stored in "`xmin`" and "`xmax`" fields of each row of table. When we `INSERT` rows, XID of inserting transaction is recorded at xmin field. When we update rows of tables (with `UPDATE` or `DELETE` statement), PostgreSQL does not simply overwrite the old rows. Instead, PostgreSQL "marks" old rows as "deleted" by writing updating transaction's XID to `xmax` field. In the case of `UPDATE` (just like `INSERT`), new rows are created whose `xmin` field is "marked" with XIDs of the creating transaction.

These "`xmin`" and "`xmax`" are used to determine which row is visible to a transaction. To do this, PostgreSQL needs a data to indicate what transactions are running at specific time. This is called "**snapshot**." If a transaction is in a snapshot, it is regarded as **running** even though it has finished.

You should understand that this specific time is not just **now**. If isolation level of a transaction is `read committed`, the transaction needs consistent visibility for some period of time, at least while an SQL statement is being executed. It is not preferable if SQL statements reads sone rows which are committed during this execution. Therefore, in the case of `read committed` isolation level, database should obtain a snapshot before the execution of a statement and continue to use it throughout the execution.

In the case of `repeatable read` and `serializable`, the transaction need consistent visibility throughout the transaction execution. In this case, the transaction should obtain the snapshot before statement execution and should continue to use it throughout the transaction execution, not single statement execution.

If a transaction which created the row is not running, visibility of each row depends upon the fact if the creating transaction was committed or aborted. Suppose a row of a table which was created by some transaction and is not deleted yet. If the creating transaction is running, such row is visible to the transaction which created the row, but not visible to other transactions. If the creating transaction is not running and was committed the row is visible. If the transaction was aborted, this row is not visible.

Therefore, PostgreSQL needs two kinds of information to determine "*which transaction is run-*

[9] More precisely, XID is unsigned 32bit integer. When XID reaches the max value, it wraps around to the lowest value (3, as to the latest definition). PostgreSQL has a means to handle this, as well as Postgres-XC. For simplicity, it will not be described in this document

[10] Please note that this description is somewhat simplified for explanation. You will find the precise rule in `tqual.c` file in PostgreSQL's source code.

ning" and "*if an old transaction was committed or aborted*."

The former information can be obtained as "snapshot." PostgreSQL maintains the latter information as "CLOG."

PostgreSQL uses all these information to determine which row is visible to a given transaction.

1.4.1.2 Making Transaction Management Global

In Postgres-XC, GTM provides the following feature for transaction management:

1. Assigning XID globally to transactions (GXID, Global Transaction ID). With GXID, global transactions can be identified globally. If a transaction writes to more than one node, we can track such writes.'

2. Providing snapshot. GTM collects all the transaction's status (running, committed, aborted etc. to provide snapshot globally (global snapshot). Please note that global snapshot includes GXID given to other servers as shown in Figure 1.8. This is needed because some older transaction may visit new server after a while. In this case, if GXID of such a transaction is not included in the snapshot, this transaction may be regarded as "*old enough*" and uncommitted rows may be read. If GXID of such transaction is included in the snapshot from the beginning, such inconsistency does not occur.

The reason why we need a global snapshot is as follows:

2PC protocol enforces update of each distributed transaction. However, it does not enforce to maintain the consistent visibility of a distributed transaction updates to others. Depending upon the timing of commit at each node, the update may or may not be visible to the same transaction which reads these nodes. To maintain the consistent visibility, we need a global snapshot which includes all the running transaction information (GXID, global transaction id, in this case) in Postgres-XC cluster and use it in the same context of the read operation as found in PostgreSQL.

To do this, Postgres-XC introduced a dedicated component called GTM (Global Transaction Manager). GTM runs as a separate component and provide unique and ordered transaction id to each transaction running on Postgres-XC servers[11]. We call this GXID (Global Transaction Id) because this is globally unique ID,

GTM receives GXID request from transactions and provide GXID. It also keep track of all the transactions when it started and finished to generate snapshot used to control each tuple visibility. Because snapshot here is also global property, it is called Global Snapshot.

As long as each transaction runs with GXID and Global Snapshot, it can maintain consistent visibility throughout the system and it is safe to run transactions in parallel in any servers. On the other hand, a transaction, composed of multiple statements, can be executed using multiple servers maintaining both update and visibility consistently. Outline of this mechanism is illustrated in Figure 1.9. Please note how transactions included in each snapshot changes according to global transaction.

[11] You can configure GTM in the same server as other components such as coordinator and datanode

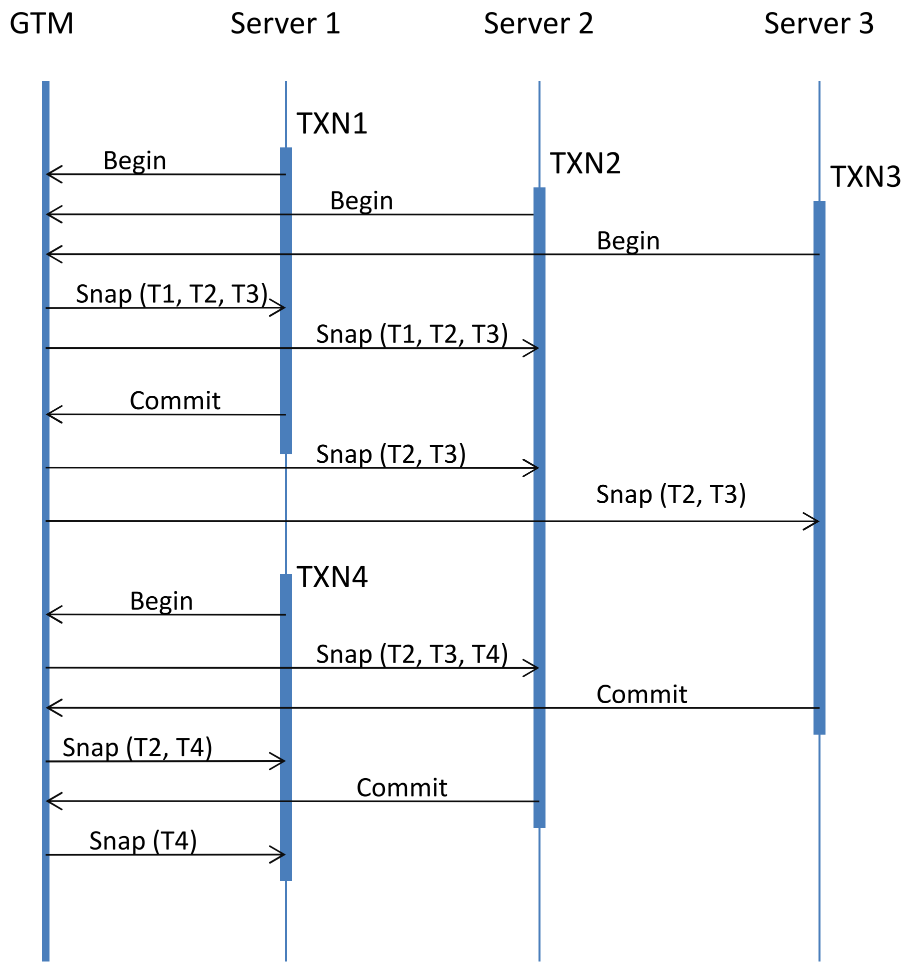

Figure 1.9: Outline of Postgres-XC's Global Transaction Management

GTM provides Global Transaction Id to each transaction and keeps track of the status of all the transactions, whether it is running, committed or aborted, to calculate global snapshot to maintain tuple visibility.

Please note that each transaction reports when it starts and ends, as well as when it issues `PREPARE TRANSACTION` command in two-phase commit protocol.

Please also note that global snapshot provided by GTM includes other transactions running on other components.

Each transaction requests snapshot according to the transaction isolation level as done in PostgreSQL. If the transaction isolation level is "`read committed`", then transaction will request a snapshot for each statementIf it is "`repeatable read`," [12] transaction will request a snapshot at the beginning of transaction and reuse it throughout the transaction.

GTM also provides global value such as sequence. Other global properties such as timestamps and notification will be an extension in the following releases[13].

1.4.2 Coordinator

Coordinator is an interface to applications. It acts like conventional PostgreSQL backend process. However, because tables may be replicated or distributed, coordinator does not store any actual data. Actual data is stored by Datanode as described below. Coordinator receives SQL statements, get Global Transaction Id and Global Snapshot as needed, determine which datanode is involved and ask them to execute (whole or a part of) the statement. When issuing statement to datanodes, coordinator propagates GXID and Global Snapshot to run the statement at datanodes in the same transaction context.

1.4.3 Datanode

Datanode actually stores user data. Tables may be distributed among datanodes, or replicated to all the datanodes. Datanode does not handle global view of the whole database and just takes care of local data. Coordinator builds the statement to run in the datanode locally. Incoming statement is examined by the coordinator as described next, and rebuilt to execute at each datanode involved. It is then transferred to each datanodes involved together with GXID and Global Snapshot as needed. Datanode may receive request from various coordinators. However, because each the transaction is identified uniquely and associated with consistent (global) snapshot, datanode doesn't have to worry what coordinator each transaction or statement came from.

Overall diagram of transaction control and query processing is shown in Figure 1.10.

[12] PostgreSQL has another isolation level "`serializable`", based upon SSI. Although it seems that global snapshot works well with SSI, this may need further discussion and study.

[13] GTM provides timestamp and is used to some extent at present.

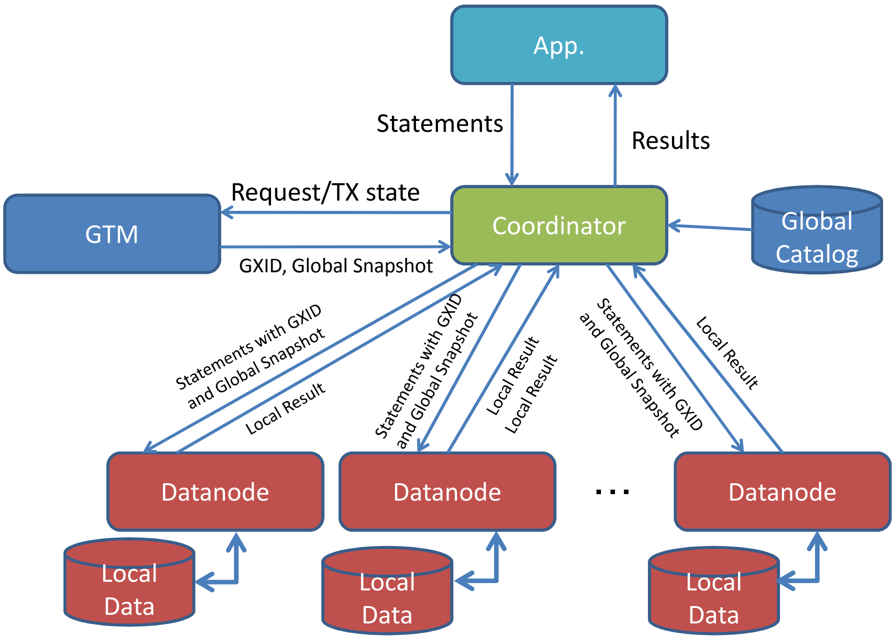

Figure 1.10: Interaction among Postgres-XC components

1.4.4 Interaction Between Key Components

As explained in the previous section, Postgres-XC has three major components to provide global consistency of both multi-node reads and writes and to determine which datanode each statement should go and to handle the statement.

Sequence of global transaction control and interaction among Postgres-XC components are given in Figure 1.11.

As shown in the figure, when a coordinator begins a new transaction, it requests GTM for new transaction ID (GXID, global transaction id). GTM keeps track of such requirement to calculate global snapshot.

If the transaction isolation mode is `REPEATED READ`, snapshot will be obtained and used throughout the transaction. When the coordinator accepts a statement from an application and the isolation mode is `READ COMMITTED`, snapshot will be obtained from the GTM. Then the statement is analyzed, determined what datanode to go, and converted for each datanode if necessary.

Please note that statements will be passed to appropriate datanodes with GXID and global snapshot to maintain global transaction Identity and visibility of each rows of tables. Each result is be collected and calculated into the response to the application.

At the end of the transaction, if multiple datanodes are involved in the update in the transaction, the coordinator issues `PREPARE TRANSACTION` for 2PC, then issue `COMMIT`. These steps will be reported to GTM as well to keeps track of each transaction status for the calculation of subsequent global snapshots.

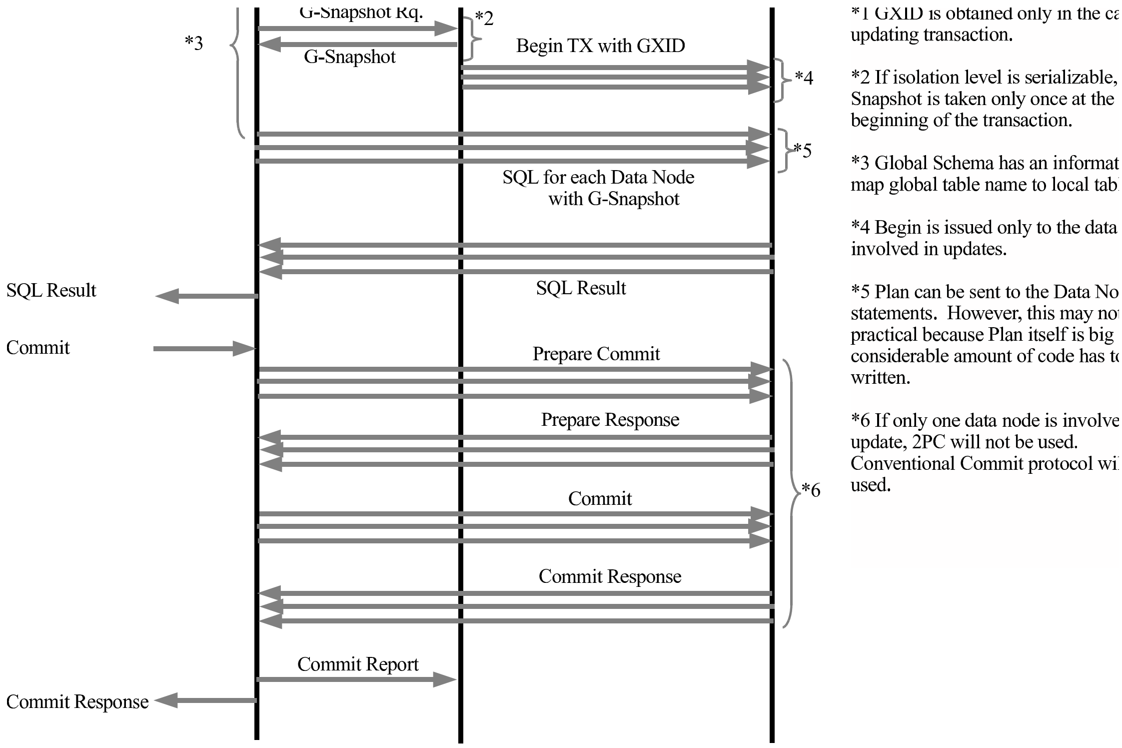

Figure 1.11: Sequence of Global Transaction Control

Please see the section 1.4.1 for details of this background.

1.5 Isn't GTM A Performance Bottleneck

Because GTM can be regarded as "serializing" all the transaction processing, it could be a performance bottleneck.

In fact, GTM can limit the whole scalability. GTM should not be used in very slow network environment such as wide area network. GTM architecture is intended to be used with Gigabit local network. For the network workload, please see section 1.7.3. Latency to send each packet may be a problem. We encourage to install Postgres-XC with local Gigabit network with minimum latency that is, use as fewer switches involved in the connection among GTM, coordinator and datanodes. Typical configuration is shown in Figure 1.12.

This chapter describes general performance issue of GTM in Postgres-XC along with GTM internal structure alternatives.

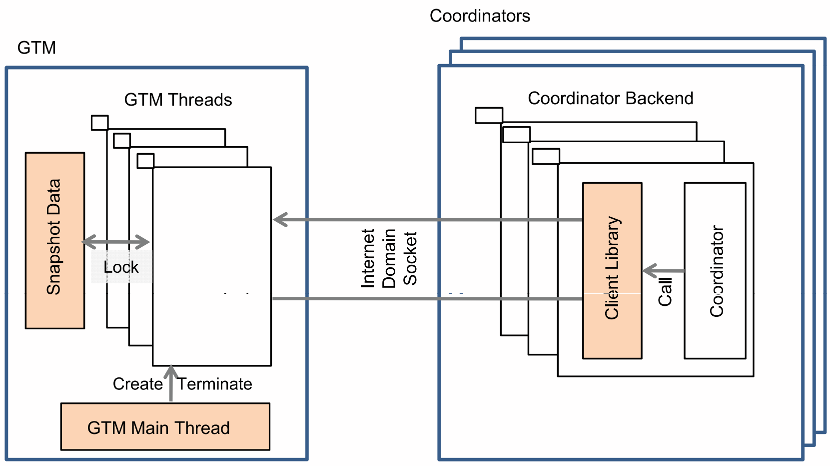

Figure 1.12: GTM Configuration without GTM Proxy

1.5.1 GTM Implementation without proxy

Sequence in Figure 1.11, can be implemented as shown in Figure 1.12. Coordinator backend corresponds to PostgreSQL's backend process which handles a database connection from an application and handles each transaction.

The outline of the structure and algorithm are as follows:

1. Coordinator backend is provided with GTM client library to obtain GXID and snapshot and to report the transaction status.
2. GTM opens a port to accept connection from each coordinator backend. When GTM accepts a connection, it creates a thread (GTM Thread) to handle request to GTM from the connected coordinator backend.
3. GTM Thread receives each request, record it and returns GXID, snapshot and other response to the coordinator backend.
4. The above sequence is repeated until the coordinator backend requests disconnect.

Each of the above interaction is done separately. For example, if the number of coordinator is ten and each coordinator has one hundred connection from applications, which is quite reasonable in single PostgreSQL in transactional applications, GTM has to have one thousand of GTM Threads. If each backend issues 25 transaction in a second and each transaction includes five statements and each coordinator runs one hundred backends, then the total number of the interaction between GTM and ten coordinators to provide global snapshot can be estimated as: $10 \times 100 \times 25 \times 5 = 125,000$. Because we have one hundred backends in each coordinator, the length of snapshot (GXID is 32bit integer, as defined in PostgreSQL) will be $4 \times 100 \times 10 = 4,000$Bytes. Therefore, GTM has to send about 600Megabytes of data in a second to support this scale. It is quite larger than Gigabit network can support[14]. In fact, the order of the amount of data sent from GTM is $O(N^2)$ where N is the number of coordinators.

Not only the amount of data is the issue. The number of interaction is an issue. Very simple test will show that Gigabit network provides up to $100,000$ interactions for each server.

Network workload measurement in later section shows the amount of data is not that large, but it is obvious that we need some means to reduce both interaction and amount of data.

The next section will explain how to reduce both the number of interaction and amount of data in GTM.

1.5.2 GTM Proxy Implementation

You may have been noticed that each transaction is issuing request to GTM so frequently and we can collect them into single block of requests in each coordinator to reduce the amount of interaction.

This is the idea of GTM Proxy Implementation as shown in Figure 1.13.

[14] In later section, you will see this estimation is too large. However, this can be a bottleneck anyway.

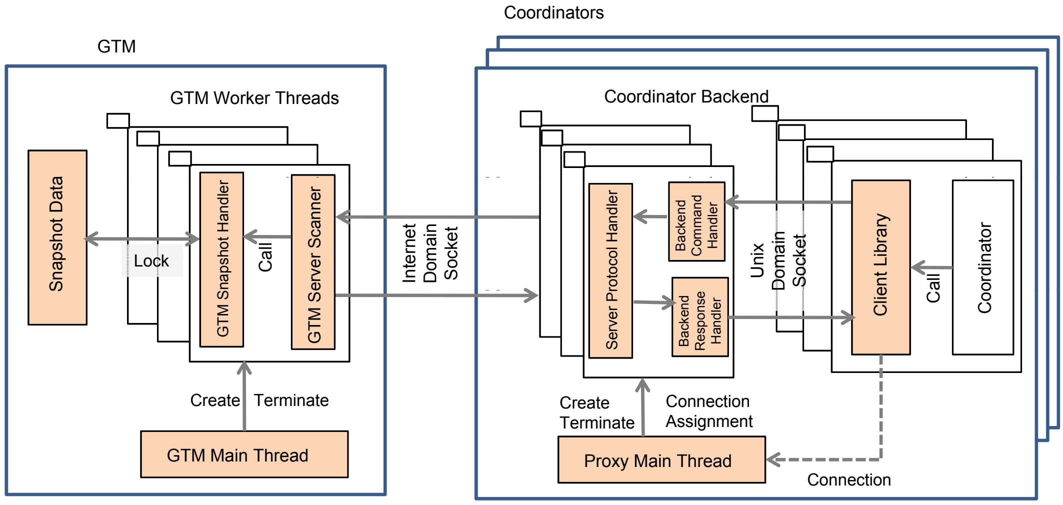

Figure 1.13: GTM Configuration with GTM Proxy

In this configuration, each coordinator backend does not connect to GTM directly. Instead, we have GTM Proxy between GTM and coordinator backend to group multiple requests and responses. GTM Proxy, like GTM explained in Section 1.5.1, accepts connection from the coordinator backend. However, it does not create new thread. The following paragraphs explains how GTM Proxy is initialized and how it handles requests from coordinator backends.

GTM Proxy, as well as GTM, is initialized as follows:

1. GTM starts up just as described in section 1.5.1. Now GTM can accept connections from GTM Proxies.
2. GTM Proxy starts up. GTM Proxy creates GTM Proxy Threads. Each GTM Proxy Threads connect to the GTM in advance. The number of GTM Proxy Threads can be specified at the startup. Typical number of threads is one or two so it can save the number of connections between GTM and Coordinators.
3. GTM Main Thread waits for the request connection from each backend.

When each coordinator backend requests for connection, Proxy Main Thread assigns a GTM Proxy Thread to handle request. Therefore, one GTM Proxy Thread takes care of multiple coordinator backends. If a coordinator has one hundred coordinator backends and one GTM Proxy Thread, this thread takes care of one hundred coordinator backends.

Then GTM Proxy Thread scans all the requests from coordinator backend. If coordinator is busier, it is expected to capture more requests in a single scan. Therefore, the proxy can group many requests into single block of requests, to reduce the number of interaction between GTM and the coordinator.

Furthermore, in a single scan, we may have multiple request for snapshots. Because these requests can be regarded as received at the same time, we can represent multiple snapshots with single one. This will reduce the amount of data which GTM provides.

Test result will be presented later but it is observed that the GTM Proxy is applicable to twenty coordinators at least in short transactional application such as DBT-1.

It is not simple to estimate the order of interaction and amount of data in GTM Proxy structure. When the workload to Postgres-XC is quite light, the interaction will be as same as the case in Section 1.5.1. On the other hand, when the workload is heavier, the amount of data is expected to be smaller than $O(N^2)$ and the number of interaction will be smaller than $O(N)$.

1.6 Performance And Stability

1.6.1 DBT-1-Based Benchmark

DBT-1 benchmark is used as a basis of performance and stability evaluation. We chose DBT-1 benchmark for the test because

- It is a typical OLTP workload benchmark available in public for transactional use case.
- Tables cannot be partitioned simply with single common distribution key. We need more than one distribution key.

The following describes how DBT-1 was modified to best tune to Postgres-XC. To localize each statement target, we modified DBT-1 tables as follows[15].

1. Customer-ID is added to `ADDRESS` table because it is practically obvious that personal information belongs to each customer and it is common practice not to share such information among different customers.
2. Stock table is divided into two tables, item and stock, as in the latest TPC-W specification.

Also, we changed connection from ODBC to libpq[16].

Table configuration of DBT-1 is illustrated in Figure 1.14 with modification for Postgres-XC. Tables with blue frame are distributed using customer ID, green with shopping cart ID and red with item ID. Tables with black frame are replicated over all the datanodes.

Please note the distribution of `SHOPPPING_CART` cart and `SHOPPING_CART_LINE` tables. It is more favorable if shopping cart and shopping card ID can be distributed using customer ID. However, DBT-1 uses these table even while a customer is not assigned to the shopping cart. This is the reason they are distributed using shopping cart ID.

We also found that current DBT-1 is not suitable for long-period test. DBT-1 does not maintain order and order line tables. From time to time, number of outstanding order increases while some of the transaction displays all such orders. Number of displayed order could be thousands and could reduce the throughput as we measure it for a long period as one week.

[15]Tables are designed so that it cannot simply be partitioned. We need more than one partitioning key.
[16]This is because early Postgres-XC implementation did not support ODBC

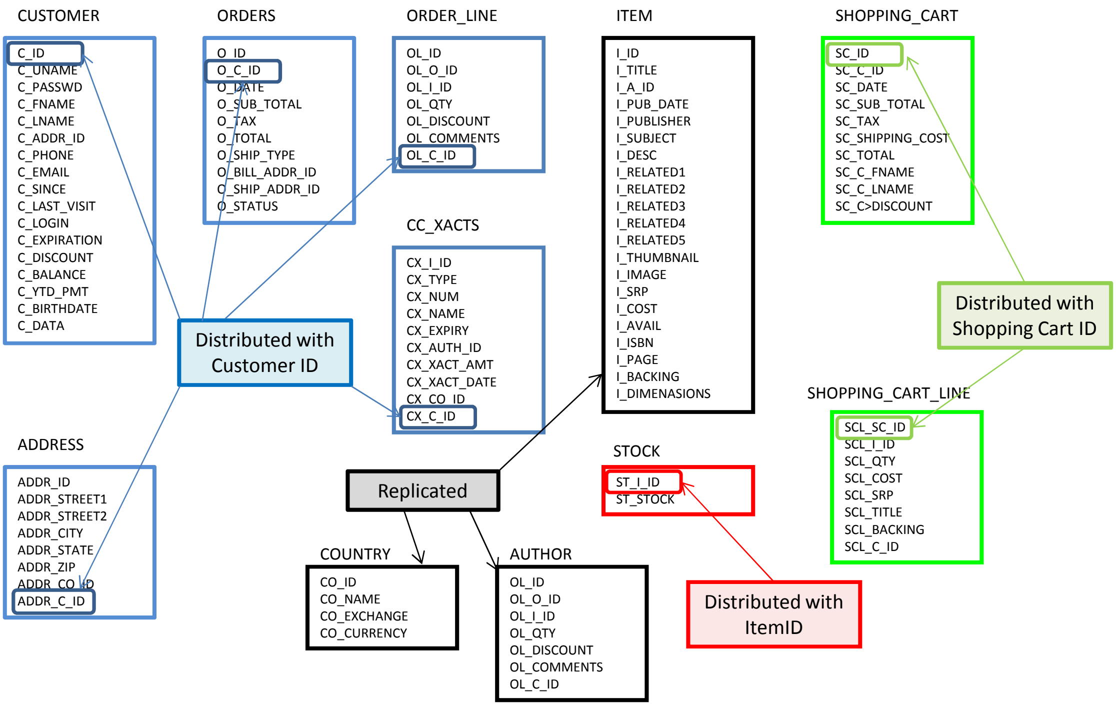

Figure 1.14: DBT-1-Based Table Structure Used in the Benchmark

To improve this, we modified the code to limit the number of displayed order. (the modification is available as a part of Postgres-XC release material).

Test environment is shown in Figure 1.15. We have one GTM, and up to ten database servers. Each server is configured with one coordinator and one datanode. Although we can install coordinator and datanode in separate servers, we used this configuration because it is simpler to balance the workload of coordinator and datanode.

Additional four servers were used to generate DBT-1 workloads.

Each servers are equipped with two NICs (1Gbps each). GTM and some of coordinator are equipped with Infiniband connection to be used when Gigabit network is not sufficient. Preliminary experiment showed that Infiniband is not required for this case. Infiniband is suitable for the workload to use giant packet. DBT-1 workload in Postgres-XC does not use giant packet.

1.7 Test Result

This section describes the benchmark test using DBT-1 based benchmark program and environment described in previous sections.

We ran the benchmark program in the following configuration.

1. Vanilla PostgreSQL for reference.

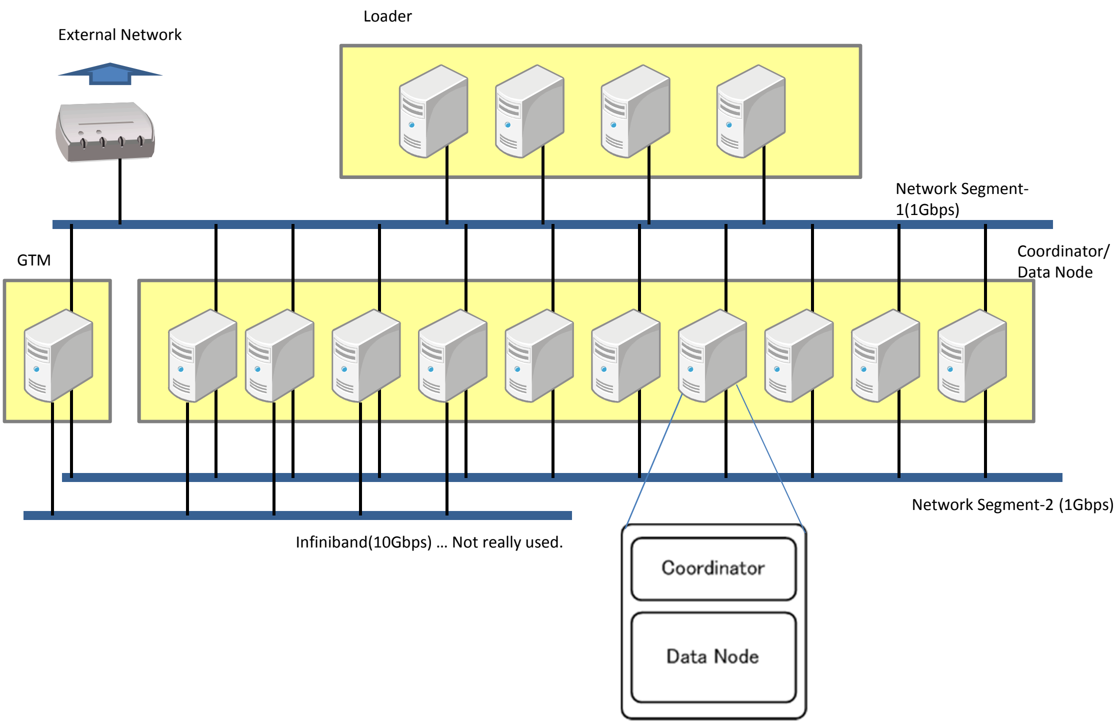

Figure 1.15: Postgres-XC Test Environment

2. Postgres-XC with one server.
3. Postgres-XC with two servers.
4. Postgres-XC with three servers.
5. Postgres-XC with five servers.
6. Postgres-XC with ten servers.

One coordinator and datanode were installed in each server. GTM was installed in a separate server. GTM-Proxy was optional to measure its effort to network workload. We ran the test with two kinds of workload as follows:

1. Full load. Measured throughput and resource consumption with full workload, which is the maximum throughput available.
2. 90% load. Arranged workload to get 90% throughput of the full load.

The following sections will explain the throughput, scale factor, resource consumption and network workload of the benchmark.

1.7.1 Throughput and Scalability

This subsection describes the measurement result of throughput and scale factor.

Table 1.1 shows the result of full load throughput for various configurations. Figure 1.16 is the chart of Postgres-XC scale factor vs. number of servers, based on the result in Table 1.1.

Table 1.1: Summary of measurement (Full load)

Database	Num.of Servers	Throughput (TPS)	Scale Factor
PostgreSQL	1	2,500	1.0
Postgres-XC	1	1,900	0.72
Postgres-XC	2	3,630	1.45
Postgres-XC	3	5,568	2.3
Postgres-XC	5	8,500	3.4
Postgres-XC	10	16,000	6.4

From these table and figure, scale factor is quite reasonable, considering that each statements parsed and analyzed twice, by coordinator and datanode.

We also ran Postgres-XC with five coordinators/datanodes for a week with 90% workload of the full load with five coordinator/datanodes. In this period, GTM, coordinators and datanodes handled GXID wraparound and vacuum freeze successfully.[17] [18]

[17]Because GXID, as well as TransactionID in PostgreSQL, is defined as 32bit unsigned integer, it reaches the maximum value some time (in this case, on 6th or 7th day) and GXID value has to return to the initial valid value. Until then, all the tuples marked with the first half value of GXID has to be frozen to special XID value defined as `FrozenTransactionId`.

[18]Please note that this measurement is a bit different from the official scaling chart used today. The original

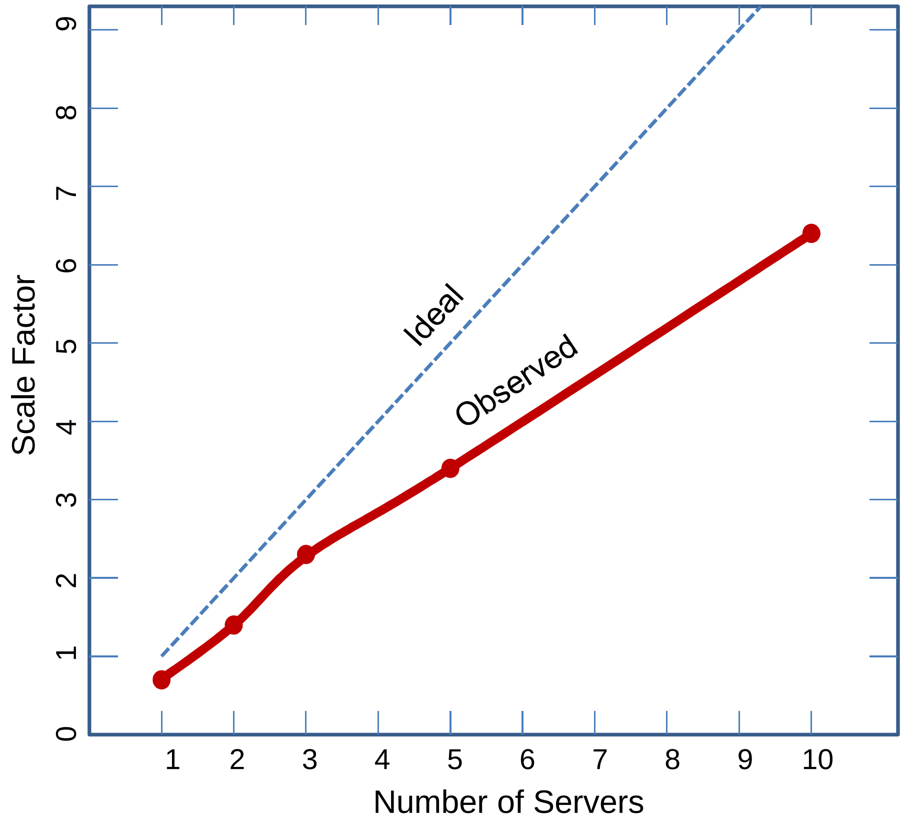

Figure 1.16: Postgres-XC Full Load Throughput

Figure 1.17 shows the throughput chart. At average, the throughput is quite stable, except that spikes are observed periodically and spike grows with time.

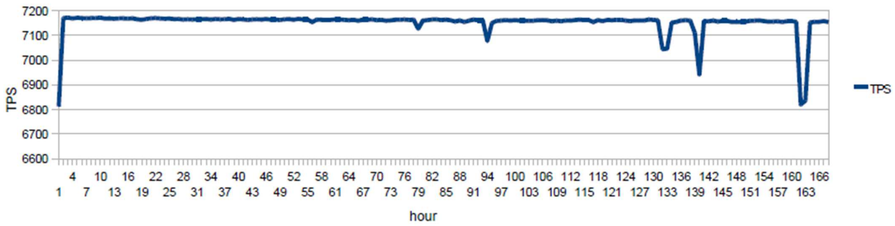

Figure 1.17: Postgres-XC 90% Load Throughput in One Week Test

1.7.2 CPU Consumption

We've measured CPU consumption in the benchmark test above to see if Postgres-XC reasonably uses hardware resource. Table 1.2 shows CPU usage (100% − idle) for various configuration and nodes with full workload.

Table 1.2: Postgres-XC CPU Usage

Configuration	GTM	CO/DN*(Av.)	Loader(Av.)
PostgreSQL**	N/A	99.2%	5.6%
Postgres-XC(1,2)***	1.9%	91.5%	5.1%
Postgres-XC(2,2)***	3.9&	95.6%	11.7%
Postgres-XC(3,2)***	6.5%	96.4%	19.3%
Postgres-XC(5,2)***	14.3%	96.4%	38.0%
Postgres-XC(10,4)***	42.2%	95.7%	34.4%

* Coordinator/Datanode
** 2 loaders were used.
*** Indicates number of Coordinator/Datanode and loader respectively.

1.7.3 Network Workload

We measured the network workload as well.

Figure 1.18 summarizes the data transfer rate among each component.

This is also summarized in Table 1.3.

This measurement indicates the following:

1. GTM Proxy drastically reduces the amount of data transfer between GTM and coordinator. Considering the network transfer rate about 100GB/s (Gigabit network), GTM can take care of at least twenty coordinators at full load.

document is based upon older measurement. For more precise analysis, we need to rerun the benchmark again with the current hardware.

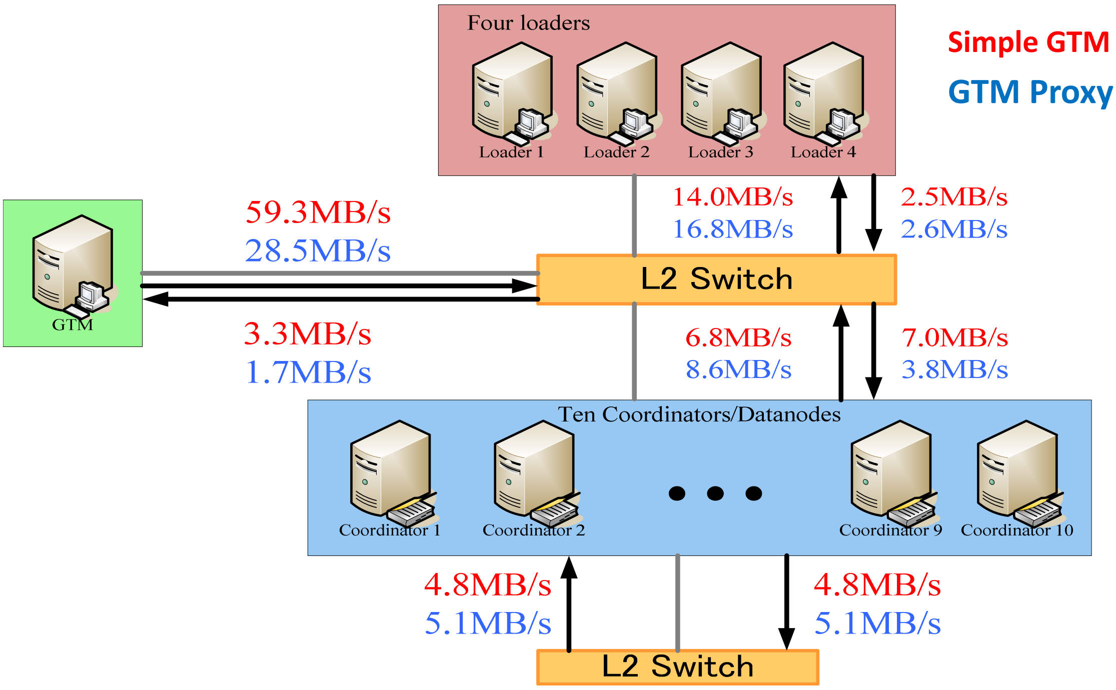

Figure 1.18: Network Data Transfer Rate of Each Server

Table 1.3: Postgres-XC Network Workload

Server	Read(simple)	Read(proxy)	Write(simple)	Write(proxy)
GTM ⇔ Coordinator	3.3MB/s	1.7MB/s	59.3MB/s	28.6MB/s
Loader ⇔ Coordinator	14.0MB/s	16.8MB/s	2.5MB/s	2.6MB/s
Coordinator/Datanode ⇔ Loader/GTM	7.0MB/s	3.8MB/s	6.8MB/s	8.6MB/s
Coordinator/Datanode ⇔ Coordinator/Datanode	4.8MB/s	5.1MB/s	4.8MB/s	4.8MB/s

2. Other server's network workload is very light. Conventional Gigabit network is sufficient.

1.7.4 Connection Handling

We want to avoid involving datanodes in a transaction that do not have target data. That is, we do not simply obtain a connection to every datanode for every client session connected to a coordinator. The connections are managed via a connection pooler process.

For example, assume we have two tables each distributed across 10 nodes via a hash on one of their respective columns. If we have a two statement transaction (each an `UPDATE`) where the `WHERE` clause of each `UPDATE` contains the hash column being compared to a literal, for each of those statements we can determine the single node to execute on. When each of those statements is executed, we only send it down to the datanode involved.

Assume in our example, only 2 datanodes were involved, one for the first `UPDATE` statement, and one for the second one. This frees up more connections that can remain available in the pool. In addition, at commit time, we commit on only those nodes involved.

Again using this example, we implicitly commit the transaction in a two-phase commit transaction since more than one datanode is involved. Note that if both of the `UPDATE`s went to the same data node, at commit time we detect this and do not bother using two phase commit, using a simple commit instead.

1.7.5 High-Availability Consideration

In High-Availability (HA, afterwords) solution, we need to integrate automatic failover system not only for Postgres-XC components but also for other system components such as server hardware, storage system and network.

Because this integration deeply depends upon specific cases, it is determined that such HA integration/solution is outside Postgres-XC scope, as in the case of vanilla PostgreSQL.

Instead, Postgres-XC's component provides each slave which is available in integrating into system-wide HA solution.

So far, gtm proxy does not have any permanent data in it and it does not need specific consideration for HA.

Coordinator and datanode uses vanilla PostgreSQL's log-shipping replication. To minimize transaction loss chance, it is highly recommended to connect with slave using synchronous replication.

GTM needs separate slave feature. In this case, GTM can copy all its status changes such as GXID and sequence values to the slave.

Chapter 2

Postgres-XC source code tree structure

The rest of this part describes major implementation details of Postgres-XC. It is based upon the source code of `REL1_2_STABLE` branch of Postgres-XC git repository.

In referring source file, directofy path may not be given or only a part of it may be given if there's no ambiguity.

2.1 Additional source directories

Postgres-XC source file tree structure follows PostgreSQL. Additional directories are shown in table 2.1.

Table 2.1: Additional source directories for Postgres-XC

Directory	Description
`contrib/pgxc_clean`	Additional module to cleanup errors.
`contrib/pgxc_ctl`	Additional module for Postgres-XC cluster configuration and operation.
`contrib/pgxc_ddl`	Additional module to propagate DDL execution to other nodes.
`contrib/pgxc_monitor`	Additional module to monitor if each node is running.
`doc-xc`	Postgres-XC reference document. See chapter 3 at page 37 for details.
`src/backend/pgxc/barrier`	Barrier module. See chapter 1 for details.
`src/backend/pgxc/copy`	Copy command module.
`src/backend/pgxc/locator`	Module to locate target datanode of given row and table.
`src/backend/pgxc/nodemgr`	Module to read/write `pgxc_node` catalog.
`src/backend/pgxc/pool`	Connection pooling module between a coordinator and other nodes.
`src/backend/pgxc/xc_ maintenance_mode`	Module to handle `xc_maintenance_mode` GUC parameter.
`src/backend/pgxc`	Postgres-XC specific modules for coordinator/datanode which cannot be classified into existing directory.
`src/bin/gtm_ctl`	GTM and GTM proxy launcher.
`src/gtm`	Global transaction manager
`src/include/gtm`	Header files for gtm.
`src/include/pgxc`	Additional header files which cannot be classified into existing directory.
`src/pgxc/tools/makesgml`	SGML generator. See 3 for details.
`src/pgxc`	Commonly used Postgres-XC-specific modules.

2.2 Additional source file for Postgres-XC

Additional Postgres-XC source files at existing PostgreSQL source directory are shown in table 2.2. Files sown in section2.2.1 are not listed in this table.

2.3 Modification to existing files

Modification of existing PostgreSQL source files are made as follows:

Table 2.2: Additional source files for Postgres-XC

File	Description
src/backend/access/rmgrdesc/pgxcdesc.c	Additional resource manager routines specific to Postgres-XC.
src/backend/access/transam/gtm.c	Global transaction manager at coordinator/datanode side.
src/backend/catalog/pgxc_class.c	pgxc_class system catalog handler.
src/backend/optimizer/path/pgxcpath.c	Postgres-XC additional module to find possible remote query paths.
src/backend/optimizer/plan/pgxcplan.c	Postgres-XC additional module of distributed query planner.
src/backend/optimizer/util/pgxcship.c	Postgres-XC additional module to evaluate expression shippability to remote nodes.
src/include/catalog/pgxc_class.h	Definition of pgxc_class system catalog.
src/include/catalog/pgxc_group.h	Definition of pgxc_group system catalog.
src/include/catalog/pgxc_node.h	Definition of pgxc_node system catalog.
src/include/optimizer/pgxcplan.h	Additional definition for the planner.
src/include/optimizer/pgxcship.h	Additional definition for evaluation of expression shippability to datanodes.

- For `*.c` and `*.h` files, Postgres-XC-specific lines are indicated by C-pre-compiler directive using `PGXC` label. An example is given in Figure 2.1.
- For `*.l` and `*.y` files, flex and bison does not provide directive as used in C source file. Modifications were done directly to these files.
- For document files, all existing sgml files are renamed into sgmlin files to include directives to indicate Postgres-XC specific description. See Chapter 3 at page 37for details.

```
#ifdef PGXC
    while ((c = getopt_long(argc, argv,
                            "ih:knvp:dSNc:j:Crs:t:T:U:lf:D:F:M:",
                            long_options, &optindex)) != -1)
#else
    while ((c = getopt_long(argc, argv,
                            "ih:nvp:dqSNc:j:Crs:t:T:U:lf:D:F:M:",
                            long_options, &optindex)) != -1)
#endif
```

Figure 2.1: Example of C source code modification

2.4 Number of lines of the source

Source code size of Postgres-XC was estimated by counting lines of additional souce code of Postgres-XC to corresponding PostgreSQL source by using `diff`.

In counting the lines of the code, the following directories and files are excluded:

- Copy from PostgreSQL, such as libpq at gtm.
- Reference documents.
- Regression test source/expected result.

Postgres-XC source code commit is `REL1_2_STABLE`, while PostgreSQL source code commit is `f5f21315d25ffcbfe7c6a3fa6ffaad54d31bcde0`.

As the result, additional source code lines from correponding PostgreSQL is about ninety-thousand lines.

Chapter 3

Postgres-XC Reference Document

This chapter describes how Postgres-XC reference document is built.

3.1 Postgres-XC Reference Document Source Structure

Because Postgres-XC inherits most of the spec from PostgreSQL, it is reasonable to prepare the reference manual using original PostgreSQL SGML files. With PostgreSQL document resource, we can build the following documents:

1. PDF (A4 and US Legal size)
2. man pages
3. html online document
4. epub file (version 1.2 or later)

Target should be as follows:

1. `postgres-A4.pdf` or `postgres-US.pdf`
2. `man`
3. `html`
4. `postgres.epub`

In the case of `html`, current Postgres-XC document handling framework allows to embed international characters like Japanese.

We should be careful to make it easier to merge with later version of PostgreSQL SGML files. To achieve this, Postgres-XC reference document source allows to embed special "TAG" to distinguish what is common, what is PostgreSQL-specific and what is Postgres-XC-specific. Also, we may want to allow translations to different languages. To make it easier to handle as an external tool, Postgres-XC built dedicated (but somewhat general) tool to select what tags to be included. Tag is used to indicate what part of the original file should be taken or thrown away. Lines not enclosed with any such tags are common to all. So SGML file may look like...

```
...
<para>
<!## PG>
...
<!## end>
<!## XC>
...
<!## end>
</para>
```

You can nest this tag. With the nest, you can include different translations in a single file.

This can be handled by a command `makesgml`, which will be placed at `src/pgxc/tools/makesgml`.

Makesgml can be invoked as follows:

```
makesgml -i inf -o outf -I include_tag ... -E exclude_tag ...
```

Each argument is optional and order of the argument is arbitrary. If you omit `-i` option, it will read from stdin. If `-o` is omitted, it will write to stdout. If input file include unspecified tags in the arguments, it will be treated as specified `-E`.

All the sgml files from original PostgreSQL tarball will be renamed to sgmlin. Then it will be filtered by `makesgml` and fed to original document build scripts.

3.2 Postgres-XC Reference Documents

Postgres-XC reference document will be found in a separate PDF file.

Chapter 4

Node Structure for Parser and Planner

In PostgreSQL, internal information used in the parser, planner and executor is called **node**. PostgreSQL defined many internal nodes for specific use to share internal status or information among various internal modules.

This chapter describes additional node definition in Postgres-XC.

4.1 New nodes

Additional nodes specific to Postgres-XC is listed in Table 4.1.

Table 4.1: Additional Internal Node Structures

Structure Name	Source File[1]	Description
AlterNodeStmt	parsenodes.h	Parsed ALTER NODE statement.
BarrierStmt	parsenodes.h	Parsed CREATE BARRIER statement.
CreateGroupStmt	parsenodes.h	Parsed CREATE NODE GROUP statement.
CreateNodeStmt	parsenodes.h	Parsed CREATE NODE statement.
DistributeBy	primnodes.h	Represents DISTRIBUTE BY clause.
DropGroupStmt	parsenodes.h	Parsed DROP NODE GROUP statement.
DropNodeStmt	parsenodes.h	Parsed DROP NODE statement.
ExecNodes	locator.h	A set of node to execute.
RemoteQueryPath	relation.h	Represents queries to be sent to datanodes.
RemoteQueryState	execRemote.h	Status of remote query execution.
RemoteQuery	pgxcplan.h	Represents whole remote query. Output of the planner.
SimpleSort	pgxcplan.h	Represents remote sort.

Outline of each table is also given as follows.

AlterNodeStmt structure members

Member Name	Description
type	Type: NodeTag; Value T_AlterNodeStmt is used.
node_name	Type: char*; Node name to change attributes
options	Type: List*; List of options in the statement.

BarrierStmt structure members

Member Name	Description
type	Type: NodeTag; Value T_BarrierStmt is used.
id	Type: char*; Name of the supplied barrier is set.

CreateGroupStmt structure members

Member Name	Description
type	Type: NodeTag; Value T_CreateGroupStmt is used.
group_name	Type: char*; Name of the node group.
nodes	Type: List*; List of the nodes in the group

`CreateNodeStmt` structure members

Member Name	Description
`type`	Type: `NodeTag`; Value `T_CreateNodeStmt` is used.
`node_name`	Type: `char*`; Node name.
`options`	Type: `List*`; List of the properties of the node.

`DistributeBy` structure members

Member Name	Description
`type`	Type: `NodeTag`; Value `T_DistributeBy` is used.
`disttype`	Type: `DistributionType`; Type of distribution. `DISTTYPE_REPLICATION`, `DISTTYPE_HASH`, `DISTTYPE_ROUNDROBIN` or `DISTTYPE_MODULO` is used.
`colname`	Type: `char*`; Distribution column name.

`DropGroupStmt` structure members

Member Name	Description
`type`	Type: `NodeTag`; Value `T_DropGroupStmt` is used.
`group_name`	Type: `char*`; Name of the group to drop.

ExecNodes structure members

Member Name	Description
type	Type: NodeTag; Value T_ExecNodes is used.
primarynodelist	Type: List*; Primary node list. Set to NULL if operation is not replicated write.
nodelist	Type: List*; List of the target nodes.
baselocatortype	Type: char; Locator type of the target relation. Definitions will be found in locator.h. 'R' for replicated type. 'H' for hash distribution type. 'G' for range distribution type. This is for extension and not used at present. 'N' for round-robin distribution type. 'C' for custom distribution type. This is for extension and not used at present. 'M' for modulo distribution type. 'O' for non-distribution type. 'D' for distribution type without specific scheme. It is used as a result of JOIN of replicated and distributed table.
en_expr	Type: Expr; Expression used to determine the target node at execution time. It is used when the planner cannot determine the execution nodes.
en_relid	Type: Oid; Relation used to determine execution nodes using en_expr.
accesstype	Type: RelationAccessType; Access type used to determine execution nodes. The type RelationAccessType is defined in locator.h.
en_dist_vars	Type: List*; This is a list of Var nodes defined in primnodes.h indicating a list of columns by which the relations or the result of query is distributed. If the distribution type is other than Hash or Modulo, this is ignored. Var structure has no Postgres-XC-specific modification.

RemoteQueryPath structure members

Member Name	Description
path	Type: Path; T_RemoteQueryPath should be set as the type member of this structure. No Postgres-XC-specific modification was made to Path structure.
rqpath_en	Type: ExecNodes; List of datanodes to execute the query on.
leftpath	Type: RemoteQueryPath*; Outer relation when this represents a join relation.
rightpath	Type: RemoteQueryPath*; Inner relation when this represents a join relation.
jointype	Type: JoinType; Join type. Defined in nodes.h.
join_restrictlist	Type: List*; Restrict list correspond to JOINs. Effective only the rest of join information is available.
rqhas_unshippable_qual	Type: bool; Indicates if there is at least one qual which cannot be shipped to the datanodes.
rqhas_temp_rel	Type: bool; Indicates if at least one of the base relations involved in this path is a temporary table.
rqhas_unshippable_tlist	Type: bool; Indicates if at least one target list entry is not completely shippable.

RemoteQueryState structure members

Member Name	Description
ss	Type: ScanState; ScanState structure of vanilla PostgreSQL. There is no Postgres-XC-specific modification to this structure. type member of this structure has to be set to T_RemoteQueryState.
node_count	Type: int; Total count of participating nodes.
connections	Type: PGXCNodeHandle**; Datanode connections being combined.
conn_count	Type: int; Count of active connections.
combine_type	Type: CombineType; Type of combining each datanode result. Definition of CombineType is given in pgxcplan.h as follows: COMBINE_TYPE_NONE: known that no row count. Do not parse. COMBINE_TYPE_SUM: Sum row counts. In the case of distributed relation. COMBINE_TYPE_SAME: All the counts should be the same. In the case of replicated write.
command_complete_count	Type: int; Count of received CommandComplete messages.
request_type	Type: RequestType; Type of the response from the datanode. The enum RequestType is defined in execRemote.h as follows: REQUEST_TYPE_COMMAND: OK or no row count response. REQUEST_TYPE_QUERY: Row description response. REQUEST_TYPE_COPY_IN: Copy In response. REQUEST_TYPE_COPY_OUT: Copy Out response.
tuple_desc	Type: TupleDesc; Tuple descriptor of emitted tuples. There is no Postgres-XC-specific modification to TupleDesc structure.
description_count	Type: int; Count of received RowDescription messages.
copy_in_count	Type: int; Count of received CopyIn messages.
copy_out_count	Type: int; Count of received CopyOut messages.
errorCode	Type: char[5]; Error code sent back to the client.
errorMessage	Type: char*; Error message sent back to the client.
errorDetail	Type: char*; Error detail sent back to the client.
currentRow	Type: RemoteDataRowData; Next data ro to be wrapped into a tuple. Definition of this structure is given in execRemote.h.
rowBuffer	Type: List*; Buffer storing rows. Used when the connection should be cleaned for reuse by other remote query.

`RemoteQuery` structure members

Member Name	Description
scan	Type: Scan; Scan structure for this remote query. T_RemoteQuery must be set to the node tag of this structure.
exec_direct_type	Type: ExecDirectType; Type of EXECUTEDIRECT if the remote query is execute direct. ExecDirectType is an enum defined in pgxcplan.h.
combine_type	Type: CombineType; See RemoteQueryState description for details.
read_only	Type: bool; Indicates not to use 2PC when committing read only steps.
force_autocommit	Type: bool; Enforces autocommit. Some commands like VACUUM require autocommit mode.
statement	Type: char*; If specified, it is used as a parsed statement name on the remote node.
cursor	Type: char*; If specified, it is used as a portal name on the remote node.
rq_num_params	Type: int; Number of parameters present in the remote statement.
rq_param_types	Type: Oid; Parameter types for the remote statement.
rq_params_internal	Type: bool; Indicates to refer to the source data plan, as against user-supplied parameters.
exec_type	Type: RemoteQueryExecType; Indicates the type of nodes where this remote query should run. RemoteQueryExecType is an enum defined in pgxcplan.h. It can take one of EXEC_ON_DATANODES, EXEC_ON_COORDS, EXEC_ON_ALL_NODES, or EXEC_ON_NONE.
is_temp	Type: bool; Indicates this remote node is based on a temporary objects.
rq_finalise_aggs	Type: bool; Indicates that the aggregate should be finalized at the datanode.
rq_sortgroup_colno	Type: bool; Indicates to use resno for sort group references instead of expressions.
remote_query	Type: Query*; Query structure representing the query to be sent to the remote node.
base_tlist	Type: List*; The target list representing the result of the query sent to the remote node.
coord_var_tlist	Type: List*; Reference target list of Vars in the plan node on coordinator.
query_var_tlist	Type: List*; Reference target list of Vars in the plan node on query.
has_row_marks	Type: bool; Indicates if SELECT has FORUPDATE or FORSHARE.
rq_save_command_id	Type: bool; Indicates to save the command ID used in some special cases.
rq_use_pk_for_rep_change	Type: bool; Indicates if primary key or unique index is used to perform update/delete on a replicated table.
rq_max_param_num	Type: int; Indicates the maximum number of parameters added in an delete operation on a replicated table.

SimpleSort structure members

Member Name	Description
type	Type: NodeTag; Value T_SimpleSort is used.
numCols	Type: int; Number of sort-key columns.
sortColIdx	Type: AttrNumber; Index of sort-key columns.
SortOperations	Type: Oid; Oid if operators used to sort.
sortCollations	Type: Oid;
nullsFirst	Type: Oid; determine to make FIRST or LAST directions effective.

4.2 Modified Nodes

Table 4.2 shows existing internal node structures with Postgres-XC-specific modification.

Table 4.2: Existing Internal Node Structures with Postgres-XC Modification

Structure Name	Source File [2]	Description
AggState	execnodes.h	Status of aggregate execution.
AlterSeqStmt	parsenodes.h	Represents ALTERSEQUENCE statement.
CreateSeqStmt	parsenodes.h	Represents CREATESEQUENCE statement.
CreateStmt	parsenodes.h	Represents CREATE statement.
EState	execnodes.h	Master working state for an Executor invocation.
IntoClause	primnodes.h	Represents INTO clause used in SELECTINTO, CREATETABLEAS and CREATEMATERIALIZEDVIEW commands.
JunkFilter	execnodes.h	Store junk attributes, an attribute in a tuple that is needed only for storing intermediate information in the executor, and does not belong in emitted tuples.
ModifyTableState	execnodes.h	Represent table modification status.
PlannerInfo	relation.h	Per-query information for planning/optimization
Query	parsenodes.h	Parsed statement.
RangeTblEntry	parsenodes.h	Relation or clause(s) representing a kind of relation which can be materialized afterwords.
TupleTableSlot	tuptable.h	Tuples stored by the executor.

Additional members of AggState structure

Member Name	Description
skip_trans	Type: bool; Indicate to skip transition step for aggregates.

Additional members of `AlterSeqStmt` structure

Member Name	Description
`is_serial`	Type: `bool`; Indicate if this sequence is a part of SERIAL process.

Additional members of `CreateSeqStmt` structure

Member Name	Description
`distributeby`	Type: `DistributeBy*`; Distribution to use, or NULL.
`subcluster`	Type: `PGXCSubCluster*`; Subcluster of the table.

Additional members of `EState` structure

Member Name	Description
`es_result_remoterel`	Type: `PlanState*`; Currently active remote relation.

Additional members of `IntoClause` structure

Member Name	Description
`distributeby`	Type: `DistributeBy*`; Distribution to use, or NULL.
`subcluster`	Type: `PGXCSubCluster*`; Subcluster of the table.

Additional members of `JunkFilter` structure

Member Name	Description
`jf_xc_node_id`	Type: `AttrNumber`; Indicates nodeid when `jf_xc_wholerow` is used as ctid.
`jf_xc_wholerow`	Type: `AttrNumber`; ctid or whole row, just like `jf_junkAttNo`.

Additional members of `ModifyTableState` structure

Member Name	Description
`mt_remoterels`	Type: `PlanState**`; Remote query node per target.

Additional members of `PlannerInfo` structure

Member Name	Description
`rs_alias_index`	Type: `int`; Used to build the alias reference.
`xc_rowMarks`	Type: `List*`; List of `PlanRowMarks` of type `ROW_MARK_EXCLUSIVE` and `ROW_MARK_SHARE`.

Additional members of `Query` structure

Member Name	Description
`sql_statement`	Type: `char*`; Original statement.
`is_local`	Type: `bool`; Indicates to enforce query execution on local node. This is used by `EXECUTEDIRECT`.
`has_to_save_cmd_id`	Type: `bool`; Indicates that command id has to be maintained. This is used when a statement is divided into more than one statements to be performed. For example, `INSERTSELECT` which inserts into a child by selecting from its parent, or a `WITH` clause that updates a table in main query and inserts a row to the same table in `WITH` clause.

Additional members of `RangeTblEntry` structure

Member Name	Description
`relname`	Type: `char*`; Table name.

Additional members of `TupleTableSlot` structure

Member Name	Description
`tts_dataRow`	Type: `char*`; Tuple data in `DataRow` format.
`tts_dataLen`	Type: `int`; Actual length of the data row.
`tts_shouldFreeRow`	Type: `bool`; Indicates to free this `tts_dataRow`.
`tts_attinmeta`	Type: `structAttInMetadata*`; Store info to extract values from `DataRow` here.
`tts_xcnodeoid`	Type: `Oid`; Oid of the node to fetch datarow.

4.3 Additional structure used in nodes

Table 4.3 shows additional structure used in the nodes as described in sections 4.1 and 4.2.

Table 4.3: Additional New Structures used in Nodes

`StructureName`	Source File [3]	Description
`PGXCSubCluster`	`primnodes.h`	Represents subcluster on which a table can be created .
`PGXCNodeHandle`	`pgxcnode.h`	Represent each node (coordinator or datanode) and status of I/O to it.
`RemoteDataRowData`	`execRemote.h`	Represent DataRow message received from a remote node.

Details of them are as follows:

`PGXCSubCluster` structure members

Member Name	Description
`type`	Type: `NodeTag`; `T_PGXCSubCluster` should be set.
`clustertype`	Type: `PGXCSubClusterType`; Indicates the type of `members`. `SUBCLUSTER_NODE` is for individual nodes and `SUBCLUSTER_GROUP` is for node group.
`member`	Type: `List*`; List of nodes or node groups

PGXCNodeHandle structure members

Member Name	Description
nodeoid	Type: Oid; Oid of the node.
sock	Type: int; Connection file descriptor.
transaction_status	Type: char; Transaction state. 'I' indicates initialized status. 'E' indicates error status.
state	Type: DNConnectionState; Connection status to remote node. This type is an enum defined in pgxcnode.h. DN_CONNECTION_STATE_IDLE: idle status. Remote node is ready for query. DN_CONNECTION_STATE_QUERY: query is sent and waiting for response. DN_CONNECTION_STATE_ERROR_FATAL: fatal error. DN_CONNECTION_STATE_COPY_IN: copy in state. DN_CONNECTION_STATE_COPY_OUT: copy out state.
combiner	Type: RemoteQueryState; Remote query state. See above for details.
have_row_desc	Type: bool; For debug.
error	Type: char*; Error message.
outBuffer	Type: char*; Output buffer.
outSize	Type: size_t; Output buffer size.
outEnd	Type: size_t; Used size of the output buffer.
inBuffer	Type: char*; Input buffer.
inSize	Type: size_t; Input buffer size.
inStart	Type: size_t; Index at the start of current input message.
inEnd	Type: size_t; Index at the end of current input message.
inCursor	Type: size_t; Cursor position of the current input message.
ck_resp_rollback	Type: RESP_ROLLBACK; Indicates response handling. Description of the enum RESP_ROLLBACK is given in pgxcnode.h. RESP_ROLLBACK_IGNORE: ignore response checking. RESP_ROLLBACK_CHECK: check whether the response was ROLLBACK. RESP_ROLLBACK_RECEIVED: response is ROLLBACK. RESP_ROLLBACK_NOT_RECEIVED: response is NOT ROLLBACK.

`RemoteDataRowData` structure members

Member Name	Description
`msg`	Type: `char*`; Last data row message.
`msglen`	Type: `int`; Length of the data row message.
`msgnode`	Type: `int`; Node number of the data row message.

4.4 Query Explanation

To explain the built plan, Postgres-XC has to know how to convert Postgres-XC specific planner node into human readable expression. Postgres-XC specific planner nodes are shown in Table 4.1 and 4.2. Postgres-XC query explain is added handling two nodes; `RemoteQuery` and `ModifyTable`.

Converting logic is implemented in `src/backend/commands/explain.c`. We have no need to change the planner for explaining, what we have to do here is just formatting the given planned statement tree. Postgres-XC extended `ExplainNode()` function for `RemoteQuery` node, and modified it for `ModifyTable` to support remote table modification. `ExplainTargetRel()` is also extended for `RemoteQuery`.

Postgres-XC introduced two options to EXPLAIN command. One is `NODES`, the other is `NUM_NODES`. Both options control whether the command displays involved nodes' information or not. These options are interpreted in `ExplainQuery()`. This change needless to touch the grammar. Please note that these additional options are documented at present, although it is very useful to build potable regression test suites.

Chapter 5

Additional Core Modules

This chapter describes implementation of Postgres-XC specific additional core modules found in `src/backend` and `src/gtm`.

5.1 Locator

Locator is a new module to Postgres-XC to determine target datanode for each statement, row or whole table. The source code is found at `src/backend/pgxc/locator`. The module consists of two source files, `locator.c` and `redistrib.c`. The former provide fundamental features to determine nodes over which given table is replicated or distributed. The latter handles row redistribution as a part of `ALTER TABLE` command.

5.1.1 `locator.c`

This module is quite independent and does not conflict with other module. Implementation is straightforward and is comprehensible.

Features of the module are:

`GetPreferredReplicationNode()`

Finds preferred node. Preferred node is a datanode which provides better performance. Preferred node can be configured using `CREATE NODE` or `ALTER NODE` statement. Please note that more than one preferred node can be configured. At the read from replicated tables, planner looks for a preferred nodes and set one of them as the remote node to run the query.

This function is called from the following codes:

Caller	File & Description
`CopyTo()`	`backend/commands/copy.c` Used to find a preferred node when `COPY` command is reading from a replicated table.
`pgxc_FQS_find_datanodes`	`pgxcship.c` Used to find the target datanode when the whole statement reading from a replicated table can be shipped.
`create_remotequery_plan()`	`pgxcplan.c` Used to find the target datanode in the standard planner for a replicated table.
`RemoteCop_GetRelationLo()`	`remotecopy.c` Used to find the target datanode in `COPY TO` operation.

`GetRelationDistribColumn()`

Finds distribution column of the given relation.

This is called from the following codes:

Caller	File & Description
`pgxc_FQS_get_relation_nodes()`	`pgxcship.c` Used to find the target datanode in handling `INSERT` command.
`distrib_delete_hash()`	`redistrib.c` Used in table redistribution by `ALTER TABLE` command.

`IsDistribColumn()`

Determines if the given column is the distribution column of the given table.

This is called from the following codes:

Caller	File & Description
`ATCheckCmd()`	`tablecmds.c` Used to check `ALTER TABLE` command if it drops distribution column. This operation is not allowed.
`transformFKConstraints()`	`parse_utilcmd.c` Used to check if distribution column is involved in a foreign key constraint.

`IsTypeDistributable()`

Determines if the given column can be the distribution column.

This is called from the following codes:

Caller	File & Description
`GetRelationDistributionItems()`	`heap.c` Used to find a distribution column in handling `DISTRIBUTE BY` clause.

`GetRoundRobinNode()`

Determines the target datanode for the next row to go for a table with round robin distribution.

This is called from the following codes:

Caller	File & Description
`GetRelationNodes()`	`locator.c` Used in finding a list of the target datanode.

`IsTableDistOnPrimary()`

Checks if the given node list include the primary node.

This is called from the following codes:

Caller	File & Description
`GetRelationNodes()`	`locator.c` Used in finding a list of primary node at update or insert operation.

`IsLocatorInfoEqual()`

Checks if given two locator information is equivalent.

This is called from the following codes:

Caller	File & Description
`pgxc_check_fk_shippability()`	`pgxcship.c` Used to check if parent and child relation refers to the same node.
`pgxc_redist_build_entry()`	`redistrib.c` Used to check if there's anything to do in row redistribution in `ALTER TABLE` command.

`GetRelationNodes()`

Obtains the list of relation nodes where the relation is defined over.

This is called from the following codes:

Caller	File & Description
`CopyTo()`	`backend/commands/copy.c` Used to determine the target node to read a table in `COPY TO` statement handling.
`CopyFrom()`	`backend/commands/copy.c` Used to determine the target node list to write to a table in `COPY FROM` statement handling.
`pgxc_FQS_get_relation_nodes`	`pgxcship.c` Used to determine the target node list to execute a statement on.
`create_remotedml_plan()`	`pgxcplan.c` Used to construct remote query plan for each node.
`get_exec_connections()`	`execRemote.c` Used to get connections to the target datanodes.
`distrib_copy_from()`	`redistrib.c` Used in performing `COPY FROM` in redistributing rows.
`GetRelationNodesByQuals()`	`locator.c` Used to reduce the node list by looking at the quals.

`GetRelationNodesByQuals()`

Obtain the list of relation nodes which satisfy given quals. This functions tries to reduce the target node list.

This is a wrapper around `GetRelationNodes()` and is called from the following codes:

Caller	File & Description
`create_plainrel_rqpath()`	`pgxcpath.c` Used to determine a target node list for plain relation path.
`pgxc_FQS_get_relation_nodes()`	`pgxcship.c` Used to obtain a target node list in fast query shipping (see Section 5.2, page 60).

`GetLocatorType()`

Gets the type of the locator of the given table (type of distribution or replication).

This function is for future usage and is not used at present.

5.1.2 `redistrib.c`

This module handles row redistribution as a part of `ALTER TABLE` operation.

External functions defined in this module are:

`PGXCRedistribTable()`

Top level function to perform row redistribution.

This function is called from the following codes:

Caller	File & Description
`ATController()`	`tablecmds.c` Used to perform `ALTER TABLE` command to redistribute table rows.

`PGXCRedistribCreateCommandList()`

Looks for the list of necessary commands to perform table redistribution.

This function is called from the following codes:

Caller	File & Description
`BuildRedistribCommands()`	`tablecmds.c` Used to build operations needed to redistribute rows of the given table in `ALTER TABLE` execution.

`makeRedistribState()`

Makes redistribution state operator.

This function is called from the following codes:

Caller	File & Description
`BuildRedistribCommands()`	`tablecmds.c` Used to build redistribution state used in `ALTER TABLE` execution.

`FreeRedistribState()`

Frees redistribution state operator.

This function is called from the following codes:

Caller	File & Description
`ATController()`	`tablecmds.c` Used to clean table redistribution work resource.

`makeRedistribCommand()`

Builds a redistribution command structure. So far, this function is called only internally within `redistrib.c` module.

`FreeRedistribCommand()`

Frees a redistribution command structure. So far, this function is called only internally within `redistrib.c` module.

5.2 FQS: Fast query shipping module

Fast query shipment became a topic when adding Postgres-XC query feature to the planner caused to slow down benchmark results. It is essentially a short cut to avoid all the planning on the coordinator (creating different join paths esp.), if we find that the query is completely shippable. The shippability is judged using the same techniques as above methods.

FQS module is implemented at `pgxcship.c` and `pgxcplan.c` together with other planner utility functions used in Postgres-XC.

In this section, major external functions defined in this module are listed and then outline of FQS handling is described.

5.2.1 `pgxcship.c` file

`pgxc_is_query_shippable()`

This function analyze the query and determine if the whole statement can be shipped to datanodes. If so, then it creates a plan and returns to the caller. If not, it just returns NULL.

This function is called by `pgxc_FQS_planner()` in `pgxcplan.c` and by other functions internally in `pgxcship.c` file.

`pgxc_is_expr_shippable()`

This function checks if the given expression can be shipped to datanodes. This is a utility function and is called by the following codes:

Caller	File & Description
BeginCopyFrom()	backend/commands/copy.c Used in preparing COPY command execution to check if a default value can be shipped to datanode.
create_remotequery_path()	pgxcpath.c Used to build a remote query path.
create_joinrel_rqpath()	pgxcpath.c Used to build a remote query path for the join if it is shippable.
pgxc_separate_quals()	pgxcplan.c Used to separate quals into shippable and non-shippable ones.
pgxc_build_shippable_tlist()	pgxcplan.c Used to build shippable target list.
pgxc_build_shippable_query_jointree()	pgxcplan.c Used to build shippable join tree.
pgxc_process_grouping_targetlist()	pgxcplan.c Used to build target list of RemoteQuery plan. Checks for aggregate shipping to datanodes.
pgxc_process_having_clause()	pgxcplan.c Used to handle HAVING clause.
create_remotelimit_plan()	pgxcplan.c Used to handle LIMIT clause.
RemoteCopy_BuildStatement()	remotecopy.c Used to build COPY query for remote management.

This is also used internally in `pgxcship.c` file.

`pgxc_is_func_shippable()`

This function determines if a given function is shippable. This is a utility function and is used only within `pgxcship.c` so far.

`pgxc_find_dist_equijoin_qual()`

This function checks equijoin condition on given relations. This is a utility function and is used only within `pgxcship.c` so far.

`pgxc_merge_exec_nodes()`

This function combines the two `exec_node` if resultant `exec_node` corresponds to the join of respective relations. This is a utility function and is used only within `pgxcship.c` so far.

`pgxc_query_has_distcolgrouping()`

This function determines if grouping clause is grouped with the distribution column.

This function is a utility function and is called by `create_remotegrouping_plan()` function in `pgxcplan.c` to generate remote grouping plan with aggregate, group by or having clause. It is also called internally from other functions in `pgxcship.c`.

`pgxc_check_index_shippability()`

This function checks index shippability described by given conditions. This is a utility function and is called by the following codes:

Caller	File & Description
`DefineIndex()`	`indexcmds.c` Used in creating new index.
`BuildRedistribCommands()`	`tablecmds.c` Used to build commands to redistribute rows.

This is also used internally in `pgxcship.c` file.

`pgxc_check_fk_shippability()`

This function checks a shippability of a parent and a child relation. This is based upon the distribution of each relation and the columns used to reference to parent and child relation. Used for inheritance or foreign key shippability evaluation.

This function is called by `ATAddForeignKeyConstraint()` in `tablecmds.c` to add foreign key constraint to a table.

`pgxc_check_triggers_shippability()`

This function checks if none of the triggers prevents the query from being FQS-ed. This is a utility function and is used only within `pgxcship.c` so far.

`pgxc_find_nonshippable_row_trig()`

This function searches for a non-shippable ROW trigger for a given type. This is a utility function and is called by the following codes:

Caller	File & Description
`pgxc_should_exec_triggers()`	`trigger.c` Used in determining where the trigger should be fired.
`pgxc_should_exec_br_trigger()`	`trigger.c` Used in determining if `BEFORE ROW` trigger should be executed in this node.
`BeginCopyFrom()`	`trigger.c` Used in the beginning of `COPY FROM` command execution.

This is also used internally in `pgxcship.c` file.

Fast query shipping (FQS) works as follows:

1. When `planner()` is invoked, it invokes `pgxc_planner()` in `pgxcplan.c`.
2. `pgxc_planner()` invokes `pgxc_FQS_planner()` in `pgxcplan.c` which invokes `pgxc_is_query_shippable()` in `pgxcship.c`.
3. If the query is shippable, `pgxc_is_query_shippable()` returns the list of the node where the query should go. `NULL` otherwise.
4. If the query is shippable, `pgxc_FQS_planner` builds and returns the plan. If not, it returns `NULL`
5. `pgxc_planner` checks the return value of the last step. If the query is shippable, returns this plan to `planner()`. Otherwise, call `standard_planner()` to build the plan and returns the plan to `planner()`.

5.3 Postgres-XC specific planner module

Planner module specific to Postgres-XC will be found at `pgxcpath.c` and `pgxcplan.c`. Provided functions are described in the following subsections.

5.3.1 `pgxcpath.c` module

This module provides following functions.

`create_plainrel_rqpath()`

This function builds a `RemoteQuery` path for a plain relation and is called from `set_plain_rel_pathlist()` at `allpaths.c` to build access paths for a plain relation without subquery or inheritance.

`create_joinrel_rqpath()`

This function builds a `RemoteQuery` path for joined relations. This function also checks if the join is shippable to the datanodes. If shippable, creates `RemoteQuery` path for this join.

This is called from `add_paths_to_joinrel()` at `joinpath.c` to build the path for join operation, including deciding inner and outer relation.

5.3.2 `pgxcplan.c` module

This module provides following functions.

`pgxc_planner`

This function is the entry point to Postgres-XC's planner. It tests if entire query is shippable to datanodes with FQS module. If not, it invokes `standard_planner`.

`AddRemoteQueryNode`

This function adds a Remote Query node to launch on datanodes. This is called from the following codes:

Caller	File & Description
`ProcessUtilitySlow()`	`utility.c` Used in handling utility statements which may associate triggers.

`pgxc_query_contains_temp_tables`

This function checks if there is any temporary object used in given list of queries. This is called from the following codes:

Caller	File & Description
`fmgr_sql_validator()`	`pg_proc.c` Used to check if the list of queries contains temporary objects.

`pgxc_query_contains_utility()`

This function checks if there is any utility statement used in given list of queries. This is called from the following codes:

Caller	File & Description
`fmgr_sql_validator()`	`pg_proc.c` Used to check if the list of queries contains utility statements.

`pgxc_rqplan_adjust_tlist()`

This function adjusts the target list of `remote_query` in `RemoteQuery` node according to the plan's target list. This is called from the following codes:

Caller	File & Description
`grouping_planner()`	`planner.c` Used in planning related to grouping or aggregate to add new top-level query plan.
`pgxc_set_agg_references()`	`setrefs.c` Used in planning aggregates to adjust `Aggref` nodes.

This function is also called internally in `pgxcplan.c` module.

5.4 Connection pooler

Connection pooler maintains connections from a coordinator to datanodes and assign this to a coordinator's backend when needed. This module saves the cost to establish connections to datanode backend each time coordinator backend needs such connection.

Connection pooler runs as a separate backend as a child process of the postmaster.

This module is implemented at `src/backend/pgxc/pool` directory, which consists of the following files: `poolmgr.c`, `poolutils.c`, `poolcomm.c` [1], `pgxcnode.c` and `execRemote.c`.

Sections below describes important functions defined in this module in file-by-file manner.

5.4.1 poolmgr.c

This file contains main module to run as pooler process. Pooler process is spawned by coordinator's postmaster. The sequence is as follows.

`PostmasterMain()`
↓
`StartPoolManageri()`
↓
`StartChildProcess(PoolerProcess)`
↓
(child process folked, child process flow follows)
↓
`PGXC_StartChildProcess()`
↓
`AuxiliaryProcessMain()`[2]
↓
`PoolManagerInit()`

`PoolManagerInit()`

`PoolManagerInit()` is the entry point of the pooler process. It initializes memory context, signal mask and signal handlers, allocates pool agents (`poolAgent` structure [3]) which accommodates each connection from the coordinator to a datanode. Then it calls `PoolerLoop()`, which is the main connection handler for coordinator backends.

[1] This module handles message read/write between a coordinator backend and pooler process. It is very similar to libpq in PostgreSQL and the details will not be given in this report. Information on libpq protocol is available at `http://www.pgcon.org/2014/schedule/attachments/330_postgres-for-the-wire.pdf`.

[3] `poolmgr.h`

`PoolerLoop()`

`PoolerLoop()` handles connection and disconnection request from coordinator backends. The logic is very similar to postmaster's main loop and the detail is not provided here.

When `PoolerLoop()` detects new connection request and establishes it, it then calls `agent_handle_input()` to handle messages from the coordinator backend.

`agent_handle_input()`

This function receives message from a coordinator backend and handle it. Functions to send and receive messages are provided by `poolcomm.c` module.

According to the input request, this function will do the following:

- **Abort**
  Send `SIGTERM` signal to all the coordinator backends which is connected to the same database with the same user. Please note that the sender of the message will not be killed.
- **Set transaction local command**
  Send transaction local `SET` command to specified coordinators (optional) and datanodes.
- **Connect**
  Initialize the connection. Set senders information and options to pooler agent information (`poolAgent`).
- **Disconnect**
  Handles disconnect request.
- **Clean connection**
  Cleanup connection for specified database with specified user.
- **Get connections**
  Transfer sockets for specified database and user back to the sender.
- **Cancel**
  Cancel SQL command in progress on specified connections.
- **Lock/Unlock**
  Lock and unlock the pooler.
- **Reload**
  Reinitialize all the pooled connections. This is needed to deal with the change in the target node list, for example, by slave promotion and update in `pgxc_node` catalog.
- **Check connection**
  Check if the connection is consistent with the system catalog.
- **Release connection**
  Release the connection and make it available to another coordinator backend.

- **Session command**
  Send session command. Session command is accumulated in the pooler to send this later to handle get connection. Sending these messages from a coordinator backend is handled by `execRemote.c` module.

5.4.2 `execRemote.c`

This module sends statement to other nodes. External functions used by other modules are as follows:

`HandleCmdComplete()`

This function combines all the results for deparsed SQL statement execution results from different nodes.

This is called from the following codes:

Caller	File & Description
`PortalRunMulti()`	`pquery.c` Used in `INSERT` command handling.

`BufferConnection()`

Buffers all the unread result from the remote node. This is needed because the same connection can be used in multi-step statement execution and in such case, following step may read outstanding result of earlier statement. This function avoids such case.

At present, this function is called only inside `execRemote.c` module.

`FetchTuple()`

Fetch the next data from the combiner's buffer. If the statement was targeted to more than one node, all the results are combined by the combiner before this fetch.

At present, this function is called only inside `execRemote.c` module.

`handle_response()`

Handles response from the remote node.

This is called from the following codes:

Caller	File & Description
`CheckBarrierCommandStatus()`	`barrier.c` Used to check the result of `CREATE BARRIER` command.

This function is also called internally in `execRemote.c` module.

`is_data_node_ready()`

Checks if the datanode is ready to accept SQL statements.

This is called from the following codes:

Caller	File & Description
`pgxc_node_flush_read()`	`pgxcnode.c` Used to drain all the outstanding messages from the datanode to prepare for sending the next command.

`pgxcNodeCopyBegin()`

Begins `COPY` command.

This is called from the following codes:

Caller	File & Description
`pgxc_node_copybegin()`	`backend/commands/copy.c` Used in a wrapper around `DataNodeCopyBegin()`.
`pgxc_send_matview_data()`	`matview.c` Used to send rows to be stored in the materialized view.
`distrib_copy_to()`	`redistrib.c` Used to collect all the data to be redistributed.
`distrib_copy_from()`	`redistrib.c` Used to distribute all the collected data to the target table.

`DataNodeCopyIn()`

Send a data row to the specified nodes.

This is called from the following codes:

Caller	File & Description
`CopyFrom()`	`backend/commands/copy.c` Used to handle `COPY FROM` command.
`pgxc_send_matview_data()`	`matview.c` Used to send rows in materialized view.
`distrib_copy_from()`	`redistrib.c` Used to distribute all the collected data to the target table.

`DataNodeCopyOut`

Receives a data row from the specified nodes.

This is called from the following codes:

Caller	File & Description
`CopyTo()`	`backend/commands/copy.c` Used to handle `COPY TO` command.
`distrib_copy_to()`	`redistrib.c` Used to collect all the data to be redistributed.

`pgxcNodeCopyFinish()`

Finishes copy process on all connections.

This is called from the following codes:

Caller	File & Description
`CopyFrom()`	`backend/commands/copy.c` Used to handle `COPY FROM` command.
`pgxc_send_matview_data()`	`matview.c` Used to send rows for materialized view.
`distrib_copy_from()`	`redistrib.c` Used to distribute all the collected data to the target table.

This function is also used internally in `execRemote.c` module.

`DataNodeCopyEnd()`

Ends a copy process on a connection.

This is called from the following codes:

Caller	File & Description
`cancel_query()`	`pgxcnode.c` Used to cancel a query due to error while processing rows.

This function is also used internally in `execRemote.c` module.

`ExecInitRemoteQuery()`

Initializes for executing remote query (`remoteQuery` node).

This is called from the following codes:

Caller	File & Description
`ExecInitNode()`	`execProcnode.c` Used in initializing all the nodes in the plan tree. Handles `RemoteQuery` plan tree.

`do_query()`

Execute the remote query.

At present, this function is called only inside `execRemote.c` module.

`ExecRemoteQuery()`

This is a wrapper for `RemoteQueryNext()`. The result is materialized at the coordinator.

This is called from the following codes:

Caller	File & Description
`ExecProcNode()`	`execProcnode.c` Used to execute `RemoteQueryState` node to return tuples.

`RemoteQueryNext()`

This is the executtion step of PGXC plan.

At present, this function is called only inside `execRemote.c` module.

`ExecEndRemoteQuery()`

Ends the remote query execution.

This is called from the following codes:

Caller	File & Description
`ExecEndNode()`	`execProcnode.c` Used to clean up all the nodes in the plan. Handles `RemoteQueryState` node.

`SetDataRowForExtParams()`

Encodes parameter values to format of DataRow message to prepare for sending down to datanodes. The data row is copied to `RemoteQueryState.paramval_data`.

At present, this function is called only inside `execRemote.c` module.

`ExecRemoteQueryReScan()`

Rescans the relation.

This is called from the following codes:

Caller	File & Description
`ExecReScan()`	`execAmi.c` Used to reset a plan node so that its output can be re-scanned. Handles `RemoteQueryState` node.

`ExecRemoteUtility()`

Executes utility statements on multipole datanodes.

This is called from the following codes:

Caller	File & Description
`ExecuteTruncate()`	`tablecmds.c` Used in handling `TRUNCATE` command.
`standard_ProcessUtility()`	`utility.c` Used to handle DDL commands in `standard_ProcessUtility()` function.
`distrib_execute_query()`	`redistrib.c` Used to redistribute rows.
`ExecUtilityStmtOnNodes()`	`utility.c` Used to propagate DDLs to coordinators and datanodes.

`PGXCNodeCleanAndRelease()`

This is called at the end of the backend. This is called from the following codes:

Caller	File & Description
`PostgresMain()`	`postgres.c` Used at coordinator backend exit.

`ExecCloseRemoteStatement()`

Closes the remote statement.

This is called from the following codes:

Caller	File & Description
`DropDatanodeStatement()`	`prepare.c` Used in handling `DROP NODE` command.

`DataNodeCopyInBinaryForAll()`

In a `COPY TO`, send to all datanodes `PG_HEADER` for a `COPY TO` in binary mode.

This is called from the following codes:

Caller	File & Description
`CopyFrom()`	`backend/commands/copy.c` Used in handling `COPY FROM` command.

`ExecSetTempObjectIncluded()`

Remembers that we have accessed a temporary object.

This is called from the following codes:

Caller	File & Description
`DefineView()`	`view.c` Used in handling `CREATE VIEW` command to record if the view is temporary.
`DoCopy()`	`backend/commands/copy.c` Used in handling `COPY` command to record if the target table is temporary.
`doDeletion()`	`dependency.c` Used in handling `DROP` command to record if the target object is temporary.
`fmgr_sql_validator()`	`pg_proc.c` Used in SQL language validator to record if the statement contains temporary object.
`transformTableLikeClause()`	`parse_utilcmd.c` Used in parsing `LIKE` clause in `CREATE TABLE` statement to record if the relation in `LIKE` clause is temporary.
`CommitTransaction()`	`xact.c` Used to record if the transaction involves temporary object at commit. It is done only when 2PC is not enforced or the transaction is implicit 2PC.

This function is also used internally in `execRemote.c` module.

`ExecIsTempObjectIncluded()`

Check if a temporary object has been accessed.

This is called from the following codes:

Caller	File & Description
`CommitTransaction()`	`xact.c` Used to record if the transaction involves temporary object at commit. It is done only when 2PC is not enforced or the transaction is implicit 2PC.
`ProcessUtilitySlow()`	`utility.c` Used in handling `CREATE VIEW` command. This command is propagated to all the other coordinators only when it is not temporary.

This function is also used internally in `execRemote.c` module.

`ExecProcNodeDMLInXC()`

This function is used to execute `Insert`/`Update`/`Delete` on the datanode using `RemoteQuery` plan.

This is called from the following codes:

Caller	File & Description
`ExecInsert()`	`nodeModifyTable.c` Used to handle `INSERT` command.
`ExecDelete()`	`nodeModifyTable.c` Used to handle `DELETE` command.
`ExecUpdate()`	`nodeModifyTable.c` Used to handle `UPDATE` command.

`RegisterTransactionNodes()`

Adds a node to the set of nodes involved in the current transaction.

At present, this function is called only inside `execRemote.c` module.

`ForgetTransactionNodes()`

Frees a list of nodes involved in the transaction.

At present, this function is called only inside `execRemote.c` module.

`AtEOXact_Remote()`

Clears per transaction remote information.

This is called from the following codes:

Caller	File & Description
`CommitTransaction()`	`xact.c` Used to at the last step of `COMMIT` command handling.
`AbortTransaction()`	`xact.c` Used to at the last step of `ABORT` command handling.

`PreCommit_Remote()`

Performs pre-commit processing for remote nodes which includes datanodes and coordinators. If more than one node are involved, 2PC commit protocol will be performed.

This is called from the following codes:

Caller	File & Description
`CommitTransaction()`	`xact.c` Used to perform implicit 2PC in handling `COMMIT` command.

`PreAbort_Remote()`

Performs abort processing for the transaction. Internally this function performs abort on all the involved nodes.

This is called from the following codes:

Caller	File & Description
`AbortTransaction()`	`xact.c` Used to abort transaction on all the involved nodes in handling `ABORT` command.

`PrePrepare_Remote()`

Perform `PREPARE` command on all the nodes involved.

This is called from the following codes:

Caller	File & Description
`PrepareTransaction()`	`xact.c` Used to prepare transaction on all the involved nodes in handling `PREPARE` command.

`PostPrepare_Remote()`

After the prepare operation, if it is explicit `PREPARE` statement, this function reports involved nodes to GTM for later use.

This is called from the following codes:

Caller	File & Description
`PrepareTransaction()`	`xact.c` Used to report 2PC status to GTM in handling `PREPARE` command.

`IsTwoPhaseCommitRequired()`

Checks if more than one node are involved in the transaction.

This is called from the following codes:

Caller	File & Description
`CommitTransaction()`	`xact.c` Used to check if implicit 2PC is needed in `COMMIT` command handling.

`FinishRemotePreparedTransaction()`

This is a step to finish prepared transaction at all the remote node involved.

This is called from the following codes:

Caller	File & Description
`standard_ProcessUtility()`	`utility.c` Used to finish prepared transaction at remote node in handling `COMMIT PREPARED` command.
`standard_ProcessUtility()`	`utility.c` Used to finish prepared transaction at remote node in handling `ROLLBACK PREPARED` command.

`pgxc_all_success_nodes()`

Collects user-friendly message from coordinators and datanodes.

This is called from the following codes:

Caller	File & Description
‘ `ExecUtilityWithMessage()`	`utility.c` Used in running a utility statement on remote nodes in a transaction block.

`set_dbcleanup_callback()`

Register a callback function which does some non-critical cleanup tasks on transaction success or abort.

This is called from the following codes:

Caller	File & Description
`CreateTableSpace()`	`tablespace.c` Used to register `createtbspc_abort_callback()` function as cleanup in handling `CREATE TABLESPACE` command.
`createdb()`	`dbcommands.c` Used to register `createdb_xact_callback()` function as cleanup in handling `CREATE DATABASE` command.
`movedb()`	`dbcommands.c` Used to register `movedb_xact_callback()` function as cleanup in handling `ALTER DATABASE SET TABLESPACE` command.

`AtEOXact_DBCleanup()`

This is called at post-commit or pre-abort from the following codes:

Caller	File & Description
`CommitTransaction()`	`xact.c` Used as post-commit handling in `COMMIT` command.
`AbortTransaction()`	`xact.c` Used as pre-abort handling in `COMMIT` command.

5.4.3 `pgxcnode.c`

This module provides functions for the coordinator to communicate with datanodes and other coordinators.

`InitMultinodeExecutor()`

Allocates and initializes memory to store datanode and coordinator handles. This function is a global initializer of Postgres-XC executor for a process.

This is called from the following codes:

Caller	File & Description
`pgxc_pool_reload()`	`poolutils.c` Used in `pgxc_poo_reload()` function. This function initializes whole executor because it refreshes all the node information from `pgxc_node` catalog.
`HandlePoolerReload()`	`poolutils.c` Used in reinitialize all the pooler status when `PROCSIG_PGXCPOOL_RELOAD` is activated.
`PostgresMain()`	`postgres.c` Called at coordinator backend initialization.

`PGXCNodeConnStr()`

Builds up a connection string to the target node.

This is called from the following codes:

Caller	File & Description
`build_node_conn_str()`	`poolmgr.c` Used to build a connection string for give node.

`PGXCNodeConnect()`

Connects to the target node.

This is called from the following codes:

Caller	File & Description
`grow_pool()`	`poolmgr.c` Used in growing pooled connection size.

`PGXCNodeClose()`

Close the connection to the target node.

This is called from the following codes:

Caller	File & Description
`destroy_slot()`	`poolmgr.c` Used in growing pooled connection size.

`PGXCNodeSendSetQuery()`

Sends `SET` statement to the given connection. Session parameters set here are reused within the pooler to restore SET option setups later.

This is called from the following codes:

Caller	File & Description
`agent_set_command()`	`poolmgr.c` Used to save a `SET` command and distribute it to the agent connections already in use.
`agent_acquire_connections()`	`poolmgr.c` Used to propagate in-use `SET` commands to newly acquired connection pool.
`send_local_commands()`	`poolmgr.c` Used to send transaction local commands.
`agent_reset_session()`	`poolmgr.c` Used to reset the session status.

`PGXCNodeConnected()`

Checks if the connection is active.

This is called from the following codes:

Caller	File & Description
`grow_pool()`	`poolmgr.c` Used in growing pooled connection size.

`pgxc_node_receive()`

Waits until at least one of specified connections has data available and read the data into the buffer.

This is called from the following codes:

Caller	File & Description
`CheckBarrierCommandStatus()`	`barrier.c` Used to check response from the nodes in `CREATE BARRIER` command handling.
`BufferConnection()`	`execRemote.c` Used in buffering all the unread result from the remote node.
`FetchTuple()`	`execRemote.c` Used in getting next row from the combiner's buffer.
`pgxc_node_receive_responses()`	`execRemote.c` Used in handling responses from PGXC node connections.
`do_query()`	`execRemote.c` Used in executing remote query in `RemoteQuery` node.
`ExecEndRemoteQuery()`	`execRemote.c` Used in finishing the remote query.
`close_node_cursors()`	`execRemote.c` Used in closing the node cursors.
`ExecRemoteUtility()`	`execRemote.c` Used in executing a utility statement on multiple PGXC nodes.
`ExecCloseRemoteStatement()`	`execRemote.c` Used in closing remote statement.

`pgxc_node_is_data_enqueued()`

Checks if any data in the TCP input buffer from PGXC node connection is waiting to be read.

This function is for future use. It is not used in Postgres-XC so far.

`pgxc_node_read_data()`

Reads up incoming messages from the PGXC node connection.

This is called from the following codes:

Caller	File & Description
`DataNodeCopyIn()`	`execRemote.c` Used in sending data row to the datanode to check if the datanode sent error messages.
`DataNodeCopyOut()`	`execRemote.c` Used in receiving data row from the datanode to consume all the extra data.

This function is used within `pgxcnode.c` module as well.

`get_char()`

Gets one character from the connection buffer and advance the cursor.

So far, this function is used only within `pgxcnode.c` module.

`get_int()`

Reads an integer from the connection buffer and advance the cursor.

So far, this function is used only within `pgxcnode.c` module.

`get_message()`

Gets a message from the connection.

This is called from the following codes:

Caller	File & Description
`handle_response()`	`execRemote.c` Used in handling response from PGXC node.
`is_data_node_ready()`	`execRemote.c` Used in checking if the datanode is ready to receive commands.

`release_handles()`

Releases all datanode and coordinator connections back to the pooler and release occupied memory.

This is called from the following codes:

Caller	File & Description
`DataNodeCopyOut()`	`execRemote.c` Used at the last step in handling response from PGXC node.
`PGXCNodeCleanAndRelease()`	`execRemote.c` Used when the coordinator backend is ending.
`PreAbort_Remote()`	`execRemote.c` Used in handling abort.

`cancel_query()`

Cancels a running query due to error while processing rows.

This is called from the following codes:

Caller	File & Description
`PreAbort_Remote()`	`execRemote.c` Used in handling abort.

`clear_all_data()`

Cleans all the data around all the communication handles.

This is called from the following codes:

Caller	File & Description
`PreAbort_Remote()`	`execRemote.c` Used in handling abort.

`ensure_in_buffer_capacity()`

Ensures specified amount of data can fit to the incoming buffer and increase it if necessary.

So far, this is used only within `pgxcnode.c` module.

`send_some()`

Sends specified amount of data from the outgoing buffer through the connection.

This is called from the following codes:

Caller	File & Description
`DataNodeCopyIn()`	`execRemote.c` Used in sending data row to the datanode to check if the datanode sent error messages.
`flushPGXCNodeHandleData()`	`execRemote.c` Used in flushing the cashed data to the datanode.

This function is used within `pgxcnode.c` module as well.

`pgxc_node_send_parse()`

Sends **PARSE** message with specified statement down to the target node.

So far, this is used only within `pgxcnode.c` module.

`pgxc_node_send_bind()`

Sends **BIND** message down to the target node.

So far, this is used only within `pgxcnode.c` module.

`pgxc_node_send_describe()`

Sends **DESCRIBE** message (portal or statement) down to the target node.

So far, this is used only within `pgxcnode.c` module.

`pgxc_node_send_close()`

Sends **CLOSE** message (portal or statement) down to the target node.

This is called from the following codes:

Caller	File & Description
`close_node_cursors()`	`execRemote.c` Used in closing the node cursor.
`ExecCloseRemoteStatement()`	`execRemote.c` Used in closing the remote statement.

`pgxc_node_send_execute()`

Sends **EXECUTE** message down to the target node.

This is called from the following codes:

Caller	File & Description
`FetchTuple()`	`execRemote.c` Used to execute a query when no tuple is available.

This function is used within `pgxcnode.c` module as well.

`pgxc_node_send_flush()`

Sends **FLUSH** message down to the target node.

This function is for future use. It is not used in Postgres-XC so far.

`pgxc_node_send_sync()`

Sends **SYNC** message down to the target node.

This is called from the following codes:

Caller	File & Description
`FetchTuple()`	`execRemote.c` Used to execute a query when no tuple is available.
`close_node_cursors()`	`execRemote.c` Used in closing a node cursor.
`ExecCloseRemoteStatement()`	`execRemote.c` Used in closing remote statement.

This function is used within `pgxcnode.c` module as well.

`pgxc_node_send_query_extended()`

Sends **GXID** with query down to the target node.

This is called from the following codes:

Caller	File & Description
`pgxc_start_command_on_connection()`	`execRemote.c` Used in sending GTM together with a statement.

`pgxc_node_flush()`

Sends all the outstanding data in the buffer to the target node.

This is called from the following codes:

Caller	File & Description
`SendBarrierPrepareRequest()`	`barrier.c` Used in sending barrier request to PGXC node.
`SendBarrierEndRequest()`	`barrier.c` Used in finishing barrier request to PGXC node.
`DataNodeCopyEnd()`	`execRemote.c` Used in finishing a copy process.

This function is used within `pgxcnode.c` module as well.

`pgxc_node_flush_read()`

Reads all the available data and waits until the target node is ready.

So far, this is used only within `pgxcnode.c` module.

`pgxc_node_send_query()`

Sends specified statement down to the PGXC node. This is called from the following codes:

Caller	File & Description
`pgxc_node_begin()`	`execRemote.c` Used in sending `BEGIN` command and GXCID to PGXC node.
`pgxc_node_remote_prepare()`	`execRemote.c` Used in preparing all nodes involved in the current transaction.
`pgxc_node_remote_commit()`	`execRemote.c` Used in committing all the PGXC nodes involved.
`pgxc_node_remote_abort()`	`execRemote.c` Used in aborting all the PGXC nodes involved.
`pgxcNodeCopyBegin()`	`execRemote.c` Used in beginning `COPY` command.
`pgxc_start_command_on_connection()`	`execRemote.c` Used in sending command to PGXC node.
`ExecRemoteUtility()`	`execRemote.c` Used in executing utility command at remote node.

`pgxc_node_send_gxid()`

Sends **GXID** down to the PGXC node. This is called from the following codes:

Caller	File & Description
`pgxc_node_begin()`	`execRemote.c` Used in sending `BEGIN` command and GXCID to PGXC node.
`pgxc_node_remote_commit()`	`execRemote.c` Used in committing all the PGXC nodes involved.
`pgxc_node_remote_abort()`	`execRemote.c` Used in aborting all the PGXC nodes involved.

`pgxc_node_send_cmd_id()`

Sends the Command ID down to the PGXC node. This is called from the following codes:

Caller	File & Description
`pgxc_start_command_on_connection()`	`execRemote.c` Used in sending command to PGXC node.

`pgxc_node_send_snapshot()`

Sends the snapshot down to the PGXC node. This is called from the following codes:

Caller	File & Description
`pgxcNodeCopyBegin()`	`execRemote.c` Used to push snapshot down to the remote PGXC node.
`pgxc_start_command_on_connection()`	`execRemote.c` Used to push snapshot down to the remote PGXC node.
`ExecRemoteUtility()`	`execRemote.c` Used to push snapshot down to the remote PGXC node.

`pgxc_node_send_timestamp()`

Sends the timestamp down to the PGXC node. This is called from the following codes:

Caller	File & Description
`pgxc_node_begin()`	`execRemote.c` Used to send coordinator's timestamp down to remote PGXC node.

`add_error_message()`

Adds another message to the list of errors to be returned back to the client at a convenient time.

`get_handles()`

Gets handles of all the nodes specified. This is called within `pgxcnode.c` module and `execRemote.c` module. The usage is straightforward and description of reference to this function is not given here.

`pfree_pgxc_all_handles()`

Frees `PGXCNodeAllHandles` structure. This is called from the following codes:

Caller	File & Description
`ExecuteBarrier()`	`barrier.c`
`RequestBarrier()`	`barrier.c`
`pgxcNodeCopyBegin()`	`execRemote.c`
`RemoteQueryNext()`	`execRemote.c` Used to execute Postgres-XC plan.
`ExecEndRemoteQuery()`	`execRemote.c`
`ExecCloseRemoteStatement()`	`execRemote.c`

`PGXCNodeGetNodeId()`

Looks at the data cached for handles and return node position. This is called from the following codes:

Caller	File & Description
`BuildRedistribCommands()`	`tablecmds.c` Used to build commands to redistribute rows in `ALTER TABLE` command.
`CopyFrom()`	`backend/commands/copy.c` Used to handle `COPY FROM` command.
`transformExecDirectStmt()`	`backend/parser/analyze.c` Used in transforming `EXECUTE DIRECT` command.
`CleanConnection()`	`poolutils.c` Used to execute `CLEAN CONNECTION` statement.
`pgxc_node_begin()`	`execRemote.c`
`pgxc_start_command_on_connection()`	`execRemote.c`
`FinishRemotePreparedTransaction()`	`execRemote.c`
`get_success_nodes()`	`execRemote.c` Used in print user-friendly message which nodes the query failed.
`distrib_copy_from()`	`redistrib.c` Used in `COPY FROM` operation in row redistribution.
`GetPreferredReplicationNode()`	`locator.c`
`IsTableDistOnPrimary()`	`locator.c`
`GetRelationNodes()`	`locator.c`
`RelationBuildLocator()`	`locator.c`

This function is also used internally in `pgxcnode.c` as well.

`PGXCNodeGetNodeOid()`

Looks at the data cached for handles and return node Oid. This is called from the following codes:

Caller	File & Description
`ExplainRemoteQuery()`	`explain.c` Used in explaining remote query.
`GetRelationDistributionNodes()`	`heap.c` Used in transforming subcluster information into sorted array of node OIDs.

`pgxc_node_str()`

Gets the name of the node. This is called from the following codes:

Caller	File & Description
`fetch_ctid_of()`	`pgxcplan.c` Used in adding ctid used in `WHERE CURRENT OF` clause.
(*Function entry*)	`fmgrtab.c` Defined as a builtin function.

`PGXCNodeGetNodeIdFromName()`

Returns node position in handles array. This function is for future use. It is not used in Postgres-XC so far.

5.4.4 `poolutils.c`

This module is utilities for Postgres-XC pooler.

Provided functions are as follows:

`pgxc_pool_check()`

Checks if pooler information in catalog is consistent with cached one.

This is called from the following codes:

Caller	File & Description
(*Function entry*)	`fmgrtab.c` Defined as a builtin function.

`pgxc_pool_reload()`

Reloads data cached with pooler and reload node connection information from the catalog for reinitialization. Needed to reflect node configuration changes brought by `CREATE NODE`, `ALTER NODE`, or `DROP NODE`.

This is called from the following codes:

Caller	File & Description
(*Function entry*)	`fmgrtab.c` Defined as a builtin function.

`CleanConnection()`

Executes `CLEAN CONNECTION` statement. This is called from the following codes:

Caller	File & Description
`standard_ProcessUtility()`	`utility.c` Used to handle `CLEAN CONNECTION` statement.

`DropDBCleanConnection()`

Cleans connection for given database before dropping it. This is called from the following codes:

Caller	File & Description
`standard_ProcessUtility()`	`utility.c` Used to handle `DROP DATABASE` statement.

`HandlePoolerReload()`

This function is called when `PROCSIG_PGXCPOOL_RELOAD` is activated. It aborts the current transaction if any, then reconnects to the pooler and reinitializes the session connection information.

This is called from the following codes:

Caller	File & Description
`procsignal_sigusr1_handler()`	`procsignal.c` Used in `SIGUSR1` signal handler.

5.5 GTM: Global Transaction Manager

This module supplies external transaction management and cluster-wide sequence infrastructure. Please refer to section 1.4.1 for the functional details.

GTM is implemented as a independent process to the postmaster. It means that GTM has own binary, configuration file, log file and pid file, and we need to start the process separately.

5.5.1 Source tree structure

Table 5.1 shows source code tree of GTM.

Table 5.1: Utility Functions Forked From PostgreSQL

`Directory`	Description
`src/include/gtm/`	Header files for GTM functions
`src/gtm/client/`	Library functions for GTM client
`src/gtm/config/`	Scanner for the configuration file
`src/gtm/main/`	GTM main program
`src/gtm/proxy/`	GTM Proxy main program described in section 5.6.
`src/gtm/recovery/`	PGXC node register functions on GTM and GTM Proxy, and utility functions for global variable of GTM standby
`src/gtm/common/`	Common functions shared among GTM and GTM Proxy and GTM clients. A part of files in this directory is forked from PostgreSQL.
`src/gtm/libpq/`	libpq protocol functions forked from PostgreSQL.
`src/gtm/path/`	Portable path handling routines forked from PostgreSQL.

5.5.2 Utility functions

5.5.2.1 Forked Utility Functions

GTM uses utility functions of PostgreSQL, while GTM is based upon the thread model, in constrast to PostgreSQL. Because GTM can't use PostgreSQL's utility functions implicitly using process-wide global variables, GTM forked them and customized. They are found in `src/gtm/libpq/`, `src/gtm/common/` and `src/gtm/path/`.

Table 5.2 lists utility functions forked from PostgreSQL and describes major changes.

Table 5.2: Utility functions forked from PostgreSQL

Function Name	Source File	
`*`	`src/gtm/libpq/ip.c`	Removed support for UNIX domain socket, SSL and Win32.
`*`	`src/gtm/libpq/pqcomm.c`	Removed support of UNIX domain socket.
`StreamServerPort`	`src/gtm/libpq/pqcomm.c`	Changed # of backlogs to fixed value.
`pq_*`	`src/gtm/libpq/pqcomm.c`	Changed to take target `Port` as argument.
`pq_*`	`src/gtm/libpq/pqcomm.c`	Removed codes for complexed usecase.
`pq_*`	`src/gtm/libpq/pqformat.c`	Removed encoding conversion.
`*`	`src/gtm/common/elog.c`	Removed support for syslog, csv and Win32.
`*`	`src/gtm/common/elog.c`	Removed PostmasterEnvironment and interrupt related code.
`*`	`src/gtm/common/elog.c`	Removed many unused error information structure field and related functions.
`*`	`src/gtm/common/elog.c`	Changed to use `pthread_exit()` if it is not the main thread.
`EmitErrorReport`	`src/gtm/common/elog.c`	Changed to take target `Port` as argument.
`send_message_to_server_log`	`src/gtm/common/elog.c`	Changed to add detailed information always.
`send_message_to_frontend`	`src/gtm/common/elog.c`	Changed to take target `Port` as argument.
`send_message_to_frontend`	`src/gtm/common/elog.c`	Shrinked to just propagate a message severity and text. Other information, for example source code function name, SQL status and so on, are omitted.
`log_line_prefix`	`src/gtm/common/elog.c`	Changed to include the thread information into the lvog prefix.
`*`	`src/gtm/common/aset.c`	Introduced a lock into the memory context shared among threads.
`*`	`src/gtm/common/mcxt.c`	Removed `NodeTag` from the memory context and introduced a lock into the memory context shared among threads.
`*`	`src/gtm/path/path.c`	Removed support for Win32 and unused functions.
`*`	`src/gtm/common/gtm_list.c`	This file is forked from `list.c`. Almost or function names, arguments and variables appended prefix `gtm_`, but they are essentially the same function of `list.c`.
`pq_send_ascii_string`	`src/gtm/libpq/pqformat.c`	`pq_send_ascii_string` appends a null-terminated text string with bypassesing encoding conversion, instead just silently replacing any non-7-bit-ASCII characters with question marks.
`pq_getmsgunreadlen`	`src/gtm/libpq/pqformat.c`	`pq_getmsgunreadlen` returns the length of the unread data in the message buffer
`current_memory_context`	`src/gtm/common/mcxt.c`	Alias to `CurrentMemoryContext`.
`allocTopMemContext`	`src/gtm/common/mcxt.c`	Allocate new memory context in `TopMemoryContext`.

make_absolute_path	src/gtm/path/path.c	If the given path name isn't absolute, make it so, assuming it is relative to the current working directory.
join_path_components	src/gtm/path/path.c	Join two path components, inserting a slash.
dupStringInfo	src/gtm/common/stringinfo.c	Get new `StringInfo` and copy the original to it
copyStringInfo	src/gtm/common/stringinfo.c	Copy `StringIinto` target `StringInfo`. Cursor of the destination is initialized.

5.5.2.2 gtm_serialize.c

This module is for the serialization and deserialization of GTM data

`gtm_get_snapshotdata_size()`

Gets a serialized size of `GTM_SnapshotData` structure. This function is used by only the functions in the same file.

`gtm_serialize_snapshotdata()`

Serializes a `GTM_SnapshotData` structure This function is used by only the functions in the same file.

`gtm_deserialize_snapshotdata()`

Deserializes a `GTM_SnapshotData` structure This function is used by only the functions in the same file.

`gtm_get_transactioninfo_size()`

Gets a serialized size of `GTM_TransactionInfo` structure This function is used by only the functions in the same file.

`gtm_serialize_transactioninfo()`

Serializes a `GTM_TransactionInfo` structure This function is used by only the functions in the same file.

`gtm_deserialize_transactioninfo()`

Deserializes a `GTM_TransactionInfo` structure This function is used only the functions in the same file.

`gtm_get_transactions_size()`

Gets a serialized size of `GTM_Transactions` structure This is called from the following codes:

Caller	File & Description
gtm_serialize_transactions()	gtm_serialize.c
ProcessGXIDListCommand()	gtm_txn.c Used in processing MSG_TXN_GXID_LIST message.

gtm_serialize_transactions()

This is called from the following codes:

Caller	File & Description
ProcessGXIDListCommand()	gtm_txn.c Used in processing MSG_TXN_GXID_LIST message.

gtm_deserialize_transactions()

Returns a number of deserialized transactions. This is called from the following codes:

Caller	File & Description
get_txn_gxid_list()	gtm_client.c Used in restoring the GXID list from the GTM active.

gtm_get_pgxcnodeinfo_size()

Returns size of PGXC node information. This is called from the following codes:

Caller	File & Description
gtm_serialize_pgxcnodeinfo	gtm_serialize.c
ProcessPGXCNodeList()	register_gtm.c Used in processing MSG_NODE_LIST message.

gtm_serialize_pgxcnodeinfo()

Returns a serialize number of PGXC node information. This is called from the following codes:

Caller	File & Description
ProcessPGXCNodeList()	register_gtm.c Used in processing MSG_NODE_LIST message.

gtm_deserialize_pgxcnodeinfo()

Returns a deserialize number of PGXC node information This is called from the following codes:

Caller	File & Description
gtmpqParseSuccess()	fe-protocol.c Used to parse NODE_LIST_RESULT type result from GTM.

gtm_get_sequence_size()

Returns size of sequence information. This is called from the following codes:

Caller	File & Description
gtm_serialize_sequence()	gtm_serialize.c
ProcessSequenceListCommand()	gtm_seq.c Used in processing MSG_SEQUENCE_LIST message.

gtm_serialize_sequence()

Returns number of serialized sequence information. This is called from the following codes:

Caller	File & Description
ProcessSequenceListCommand()	gtm_seq.c Used in processing MSG_SEQUENCE_LIST message.

gtm_deserialize_sequence()

Returns number of deserialized sequence information This is called from the following codes:

Caller	File & Description
gtmpqParseSuccess()	fe-protocol.c Used to parse SEQUENCE_LIST_RESULT type result from GTM.

5.5.2.3 gtm_serialize_debug.c

This module provides debug functionalities of serialization management

dump_transactions_elog()

Dumps GTM_TransactionInfo structure to elog. This is a debug supporting function and not called by the release code.

dump_transactioninfo_elog()

Dumps GTM_Transactions structure to elog. This is a debug supporting function and not called by the release code.

5.5.2.4 gtm_utils.c

This module supplies utility functions of GTM

`gtm_util_init_nametabs()`

Initializes mapping table value to value name table. This function is used only by the functions in the same file.

`gtm_util_message_name()`

Returns mapped type name. This is called from the following codes:

Caller	File & Description
`ProcessCommand`	`main.c` Output received message name for debug.

`gtm_util_result_name()`

Returns mapped result name. This function never used.

5.5.2.5 gtm_lock.c

This module provide locking infrastructure for GTM

`GTM_RWLockAcquire()`

Requests to acquire read-write lock in specified mode.

This function is implemented with POSIX-thread functions.

This function is called from various functions that requires exclusive operations. The caller list is not shown here because it is too large and does not look useful.

`GTM_RWLockRelease()`

Releases a read-write lock previously acquired using `GTM_RWLockAcquire`.

This function is implemented with POSIX-thread functions.

This function is called from various functions that requires exclusive operations. The caller list is not shown here because it is too large and does not look useful.

`GTM_RWLockInit()`

Initializes a read-write lock object.

This function is implemented with POSIX-thread functions.

This function is called from various functions that initializes objects which requires exclusive operations. The caller list is not shown here because it is too large and does not look useful.

`GTM_RWLockDestroy()`

Destroys a read-write lock object initialized using `GTM_RWLockDestroy`.

This function is implemented with POSIX-thread functions.

This function is called from various functions that releases objects which requires exclusive operations. The caller list is not shown here because it is too large and does not look useful.

`GTM_RWLockConditionalAcquire()`

Conditionally acquires a read-write lock without blocking.

This function is implemented with POSIX-thread functions.

This function is not used at present.

`GTM_MutexLockAcquire()`

Acquires a mutex lock.

This is called from the following codes:

Caller	File & Description
`GTMProxy_ThreadMain`	`proxy_main.c` Locking thread information object.
`GTMProxy_ThreadAddConnection`	`proxy_thread.c` Locking thread information object.
`GTMProxy_ThreadRemoveConnection`	`proxy_thread.c` Locking thread information object.

`GTM_MutexLockRelease()`

Releases previously acquired lock. This is called from the following codes:

Caller	File & Description
`GTMProxy_ThreadMain`	`proxy_main.c` Releasing a thread information object
`GTMProxy_ThreadAddConnection`	`proxy_thread.c` Releasing a thread information object
`GTMProxy_ThreadAddConnection`	`proxy_thread.c` Releasing a thread information object
`GTMProxy_ThreadAddConnection`	`proxy_thread.c` Releasing a thread information object
`GTMProxy_ThreadRemoveConnection`	`proxy_thread.c` Releasing a thread information object
`GTMProxy_ThreadRemoveConnection`	`proxy_thread.c` Releasing a thread information object

`GTM_MutexLockInit()`

Initializes a mutex lock This is called from the following codes:

Caller	File & Description
`GTMProxy_ThreadCreate`	`proxy_thread.c` Creating a thread information object

`GTM_MutexLockDestroy()`

Destroys a mutex lock. This function is not called at present.

`GTM_MutexLockConditionalAcquire()`

Conditionally acquires a lock without locking.

This function is not called at present.

5.5.3 Common modules

Some modules are shared with GTM and GTM Proxy. This subsection describes about them.

5.5.3.1 `register_common.c`

This module supplies PGXC Node Register on GTM and GTM Proxy, node registering functions

`pgxcnode_get_all()`

Returns all node information into target array. This is called from the following codes:

Caller	File & Description
`ProcessPGXCNodeList`	`register_gtm.c` Used in processing `MSG_NODE_LIST` message.

`pgxcnode_find_by_type()`

Returns node information into target array filterd by specified node type. This is called from the following codes:

Caller	File & Description
`find_standby_node_info`	`gtm_standby.c` Used to find standby node information.

`Recovery_PGXCNodeRegister()`

Adds node information to registered node list.

This function also saves node information to “register.node” file if it is not in recovery mode.

This is called from the following codes:

Caller	File & Description
gtm_standby_restore_node	gtm_standby.c Used in the process of recovery node information from the GTM active.
ProcessPGXCNodeRegister	register_gtm.c Used in processing MSG_NODE_REGISTER / MSG_BKUP_NODE_REGISTER message
ProcessPGXCNodeCommand	proxy_main.c Used in processing MSG_NODE_REGISTER message
Recovery_RestoreRegisterInfo	register_common.c Used in the process of recovery node information from “register.node” file.

Recovery_PGXCNodeUnregister()

Unregisters the given node. This is called from the following codes:

Caller	File & Description
ProcessPGXCNodeRegister	register_gtm.c Used to remove existing standby node information in processing MSG_NODE_REGISTER message.
ProcessPGXCNodeUnregister	register_gtm.c Used in processing MSG_NODE_UNREGISTER / MSG_BKUP_NODE_UNREGISTER message
ProcessPGXCNodeCommand	proxy_main.c Used in processing MSG_NODE_UNREGISTER message
Recovery_RestoreRegisterInfo	register_common.c Used in the process of recovery node information from “register.node” file.

Recovery_PGXCNodeBackendDisconnect()

Updates node information to be disconnected. This is called from the following codes:

Caller	File & Description
ProcessPGXCNodeBackendDisconnect	register_common.c Used in processing MSG_BACKEND_DISCONNECT message.

Recovery_RecordRegisterInfo()

Adds a Register or Unregister record on PGXC Node file on disk. This is called from the following codes:

Caller	File & Description
`Recovery_PGXCNodeUnregister`	`register_common.c` Used in unregistering node information.
`Recovery_PGXCNodeRegister`	`register_common.c` Used in registering node information.

`Recovery_RestoreRegisterInfo()`

Restores node information from "`register.node`" file.

This is called from the following codes:

Caller	File & Description
`main`	`main.c` Used in start up sequence to restore registered node information if it is not standby mode.
`main`	`proxy_main.c` Used in start up sequence to restore registered node information.

`Recovery_SaveRegisterInfo()`

Rewrites only data of currently registered nodes stored in on-disk register information. This is called from the following codes:

Caller	File & Description
`GTM_SigleHandler`	`main.c` Used in a part of shutdown sequence
`GTMProxy_SigleHandler`	`proxy_main.c` Used in a part of shutdown sequence

`Recovery_PGXCNodeDisconnect()`

Disconnects node whose master connection has been disconnected with GTM. This is called from the following codes:

Caller	File & Description
`GTM_ThreadMain`	`main.c` Used in the message loop when it detects EOF or unknown query type.
`GTMProxy_HandleDisconnect`	`proxy_main.c` Used when the message loop detects EOF or unknown query type.

`Recovery_SaveRegisterFileName()`

Returns the path of the file stored registered node information.

This is called from the following codes:

Caller	File & Description
`BaseInit`	`main.c` Used in start up sequence
`BaseInit`	`proxy_main.c` Used in start up sequence

5.5.4 Main program

5.5.4.1 main.c

This file contains main module of GTM process.

The GTM active is initialized as following sequence in `main()`. It's very simple: Setup configuration, initialize the main thread, restore information from files and accept connections.

`InitializeGTMOptions()`
↓
(Parse command line options and load configuration file)
↓
`BaseInit()`
↓
`GTM_RestoreTxnInfo()`
↓
`GTM_RestoreSeqInfo()`
↓
`GTM_Recovery_RestoreRegisterInfo()`
↓
(Establish input sockets)
↓
(Setup signal handlers)
↓
`ServerLoop()`

The GTM standby is initialized as following sequence in `main()`. It's made so complexed to restore information from GTM active, but please notice that basically these sequence similar to active.

```
InitializeGTMOptions()
↓
(Parse command line options and load configuration file)
↓
BaseInit()
↓
gtm_standby_start_startup()
↓
gtm_standby_begin_backup()
↓
gtm_standby_restore_next_gxid()
↓
gtm_standby_restore_gxid()
↓
gtm_standby_restore_sequence()
↓
GTM_SetNeedBackup()
↓
GTM_WriteRestorePoint()
↓
gtm_standby_register_self()
↓
gtm_standby_restore_node()
↓
gtm_standby_end_backup()
↓
(Establish input sockets)
↓
(Setup signal handlers)
↓
gtm_standby_activate_self()
↓
gtm_standby_finish_startup()
↓
ServerLoop()
```

`ServerLoop()` is brought from same name function of `postmaster.c`. It waits for new connection, connects to standby if needed, then calls `GTMAddConnection()` to create a thread to process the connection. If it detects a signal to abort, it saves information to the control file and exit.

`GTM_ThreadMain()` is a main function with the message loop. This function seems to be copied from `postmaster.c:BackendRun()`. First it initialized many things; for example a memory context, the connection, a message buffer and an exception stack. As in the case of PostgreSQL, a signal and an exception in child functions is propagted with long jump, so an exception handler is registered using `sigsetjmp()`.

`GTM_ThreadMain()` calls `ProcessCommand()` to process the "command" message from GTM clients. `ProcessCommand()` dispatches the message as shown in Table 5.3. It also saves transaction and sequence information after it process a message if needed.

The processing function dispatches a command message while processing the message, sends a backup message to GTM slave and sends a response message to assigned GTM client. If the client is a backend, the response message is sent immediately. But if it is GTM proxy, the function just places the response message to libpq buffer, and flushes the buffer when it receives a message with type "F" (flush). This basis are common to all the processing function, so these functions are not explained unless it has anything significant.

Table 5.3: GTM Message Dispatcher

Message	Dispatcher	Processing function
MSG_SYNC_STANDBY	-	ProcessSyncStandbyCommand()
MSG_NODE_REGISTER	ProcessPGXCNodeCommand()	ProcessPGXCNodeRegister()
MSG_BKUP_NODE_REGISTER		ProcessPGXCNodeRegister()
MSG_NODE_UNREGISTER		ProcessPGXCNodeUnregister()
MSG_BKUP_NODE_UNREGISTER		ProcessPGXCNodeUnregister()
MSG_NODE_LIST		ProcessPGXCNodeList()
MSG_BEGIN_BACKUP	-	ProcessGTMBeginBackup()
MSG_END_BACKUP	-	ProcessGTMEndBackup()
MSG_NODE_BEGIN_REPLICATION_INIT	ProcessTransactionCommand()	ProcessBeginReplicationInitialSyncRequest()
MSG_NODE_END_REPLICATION_INIT		ProcessEndReplicationInitialSyncRequest()
MSG_TXN_BEGIN		ProcessBeginTransactionCommand()
MSG_BKUP_TXN_BEGIN		ProcessBkupBeginTransactionCommand()
MSG_TXN_BEGIN_GETGXID		ProcessBeginTransactionGetGXIDCommand()
MSG_BKUP_TXN_BEGIN_GETGXID		ProcessBkupBeginTransactionGetGXIDCommand()
MSG_TXN_BEGIN_GETGXID_AUTOVACUUM		ProcessBeginTransactionGetGXIDAutovacuumCommand()
MSG_BKUP_TXN_BEGIN_GETGXID_AUTOVACUUM		ProcessBkupBeginTransactionGetGXIDAutovacuumCommand()
MSG_TXN_BEGIN_GETGXID_MULTI		ProcessBeginTransactionGetGXIDCommandMulti()
MSG_BKUP_TXN_BEGIN_GETGXID_MULTI		ProcessBkupBeginTransactionGetGXIDCommandMulti()
MSG_TXN_START_PREPARED		ProcessStartPreparedTransactionCommand()
MSG_BKUP_TXN_START_PREPARED		ProcessStartPreparedTransactionCommand()
MSG_TXN_PREPARE		ProcessPrepareTransactionCommand()
MSG_BKUP_TXN_PREPARE		ProcessPrepareTransactionCommand()
MSG_TXN_COMMIT		ProcessCommitTransactionCommand()
MSG_BKUP_TXN_COMMIT		ProcessCommitTransactionCommand()
MSG_TXN_COMMIT_PREPARED		ProcessCommitPreparedTransactionCommand()
MSG_BKUP_TXN_COMMIT_PREPARED		ProcessCommitPreparedTransactionCommand()
MSG_TXN_ROLLBACK		ProcessRollbackTransactionCommand()
MSG_BKUP_TXN_ROLLBACK		ProcessRollbackTransactionCommand()
MSG_TXN_COMMIT_MULTI		ProcessCommitTransactionCommandMulti()
MSG_BKUP_TXN_COMMIT_MULTI		ProcessCommitTransactionCommandMulti()
MSG_TXN_ROLLBACK_MULTI		ProcessRollbackTransactionCommandMulti()
MSG_BKUP_TXN_ROLLBACK_MULTI		ProcessRollbackTransactionCommandMulti()
MSG_TXN_GET_GXID		ProcessGetGXIDTransactionCommand()
MSG_TXN_GET_GID_DATA		ProcessGetGIDDataTransactionCommand()
MSG_TXN_GET_NEXT_GXID		ProcessGetNextGXIDTransactionCommand()
MSG_TXN_GXID_LIST		ProcessGXIDListCommand()
MSG_SNAPSHOT_GET	ProcessSnapshotCommand()	ProcessGetSnapshotCommand()
MSG_SNAPSHOT_GXID_GET		ProcessGetSnapshotCommandMulti()
MSG_SNAPSHOT_GET_MULTI		ProcessGetSnapshotCommand()
MSG_SEQUENCE_INIT	ProcessSequenceCommand()	ProcessSequenceInitCommand()
MSG_BKUP_SEQUENCE_INIT		ProcessSequenceInitCommand()
MSG_SEQUENCE_ALTER		ProcessSequenceAlterCommand()
MSG_BKUP_SEQUENCE_ALTER		ProcessSequenceAlterCommand()
MSG_SEQUENCE_GET_NEXT		ProcessSequenceGetNextCommand()
MSG_BKUP_SEQUENCE_GET_NEXT		ProcessSequenceGetNextCommand()
MSG_SEQUENCE_SET_VAL		ProcessSequenceSetValCommand()
MSG_BKUP_SEQUENCE_SET_VAL		ProcessSequenceSetValCommand()
MSG_SEQUENCE_RESET		ProcessSequenceResetCommand()
MSG_BKUP_SEQUENCE_RESET		ProcessSequenceResetCommand()
MSG_SEQUENCE_CLOSE		ProcessSequenceCloseCommand()
MSG_BKUP_SEQUENCE_CLOSE		ProcessSequenceCloseCommand()
MSG_SEQUENCE_RENAME		ProcessSequenceRenameCommand()
MSG_BKUP_SEQUENCE_RENAME		ProcessSequenceRenameCommand()

MSG_SEQUENCE_LIST		ProcessSequenceListCommand()
MSG_TXN_GET_STATUS	ProcessQueryCommand()	*Not implemented*
MSG_TXN_GET_ALL_PREPARED		*Not implemented*
MSG_BARRIER	-	ProcessBarrierCommand()
MSG_BKUP_BARRIER	-	ProcessBarrierCommand()
MSG_BACKEND_DISCONNECT	-	GTM_RemoveAllTransInfos() ProcessPGXCNodeBackendDisconnect()

5.5.4.2 `gtm_txn.c`

This module supplies transaction handling functionality, this is one of the most importarnt part of GTM.

Figure 5.1 shows how transaction information is held within GTM. As the figure shows, each transaction information is refered by two pointers, one is an array and the other is a linked list. And the index of the array is called "handle." This will help your understanding.

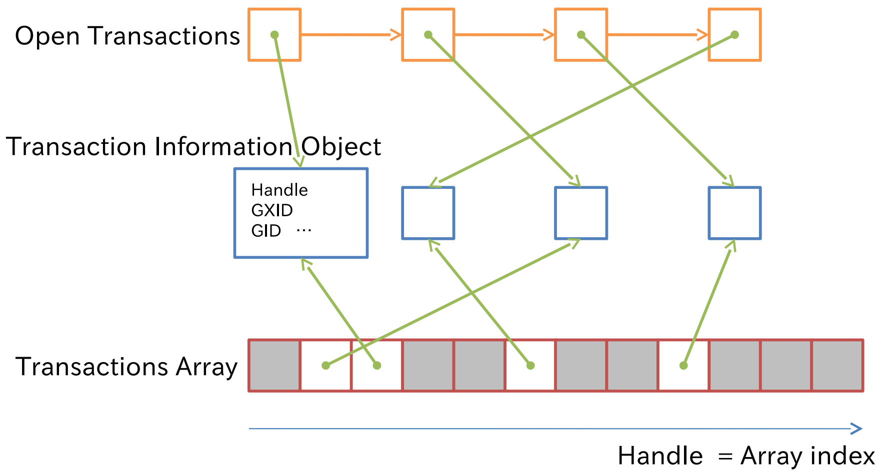

Figure 5.1: Transaction management in GTM

Note: Some codes seems to be copied from `transam.c`.

`GTM\_HandleToTransactionInfo()`

Finds the corresponding transaction info structure by the transaction handle. This is called from the following codes:

Caller	File & Description
-	`gtm_txn.c`
`GTM_GetTransactionSnapshot`	`gtm_snap.c` Used in taking the transaction snapshot to check and store information with given handles.

`GTM_GXIDToHandle()`

Finds the transaction handle corresponding to given GXID. This is called from the following codes:

Caller	File & Description
-	`gtm_txn.c`
`ProcessGetSnapshotCommand`	`gtm_snap.c` Used in processing `MSG_SNAPSHOT_GET` command to obtain the handle from given GXID for querying the transaction information.
`ProcessGetSnapshotCommandMulti`	`gtm_snap.c` Used in processing `MSG_SNAPSHOT_GET_MULTI` command to obtain the handle from given GXID for querying the transaction information.

`GTM_GIDToHandle()`

Finds the transaction handle corresponding to the given the GID (for a prepared transaction). This function is used by only the functions in the same file.

`GTM_InitTxnManager()`

Initializes global variables of the transaction manager. This is called from the following codes:

Caller	File & Description
`BaseInit`	`main.c` Used in start up sequence.

`GTM_BeginTransaction()`

Begins a new transaction and return assigned handle.

This function creates new transaction object and register it into opend transaction list.

This is called from the following codes:

Caller	File & Description
`ProcessBeginTransactionCommand`	`gtm_txn.c` Used in processing `MSG_TXN_BEGIN` message
`ProcessBeginTransactionGetGXIDCommand`	`gtm_txn.c` Used in processing `MSG_TXN_BEGIN_GETGXID` message
`ProcessBeginTransactionGetGXIDAutovacuumCommand`	`gtm_txn.c` Used in processing `MSG_TXN_BEGIN_GETGXID_AUTOVACUUM` message
`ProcessGetGIDDataTransactionCommand`	`gtm_txn.c` Used in processing `MSG_TXN_GET_GID_DATA` message

`GTM_BeginTransactionMulti()`

Begins a new transactions and returns number of new transactions began.

This function delegates internal logic to `GTM_BeginTransaction`.

This is called from the following codes:

Caller	File & Description
`GTM_BeginTransaction`	`gtm_txn.c` Delegates internal logic
`ProcessBeginTransactionGetGXIDCommandMulti`	`gtm_txn.c` Used in processing `MSG_TXN_BEGIN_GETGXID_MULTI` message

`GTM_RollbackTransaction()`

Rolls back a transaction by handle.

This function delegates internal logic to `GTM_RollbackTransactionMulti`.

This is called from the following codes:

Caller	File & Description
`GTM_RollbackTransactionGXID`	`gtm_txn.c` Convert GXID to handle and delegates internal logic
`ProcessRollbackTransactionCommand`	`gtm_txn.c` Used in processing `MSG_TXN_ROLLBACK` / `MSG_BKUP_TXN_ROLLBACK` message

`GTM_RollbackTransactionMulti()`

Rolls back transactions by handles.

This function marks transactions as "rollback in progress" and removes them from opend transaction list.

This is called from the following codes:

Caller	File & Description
`GTM_RollbackTransaction`	`gtm_txn.c` Delegates internal logic
`ProcessRollbackTransactionCommandMulti`	`gtm_txn.c` Used in processing `MSG_TXN_ROLLBACK_MULTI` or `MSG_BKUP_TXN_ROLLBACK_MULTI` message

`GTM_RollbackTransactionGXID()`

Rolls back transaction by GXID

This function converts GXID to the handle and delegates internal logic to `GTM_RollbackTransaction`.

This function is never used.

`GTM_CommitTransaction()`

Commits a transaction by handle.

This function delegates internal logic to `GTM_CommitTransactionMulti`.

This is called from the following codes:

Caller	File & Description
`GTM_CommitTransactionGXID`	`gtm_txn.c` Convert GXID to handle and delegates internal logic
`ProcessCommitTransactionCommand`	`gtm_txn.c` Used in processing `MSG_TXN_COMMIT` message

`GTM_CommitTransactionMulti()`

Commits transactions by handles.

This function marks transactions as "commit in progress" and removes them from opend transaction list.

This is called from the following codes:

Caller	File & Description
`GTM_CommitTransaction`	`gtm_txn.c` Delegates internal logic
`ProcessCommitPreparedTransactionCommand`	`gtm_txn.c` Used in processing `MSG_TXN_COMMIT_PREPARED` / `MSG_BKUP_TXN_COMMIT_PREPARED` message
`ProcessCommitTransactionCommandMulti`	`gtm_txn.c` Used in processing `MSG_TXN_COMMIT_MULTI` / `MSG_BKUP_TXN_COMMIT_MULTI` message

`GTM_CommitTransactionGXID()`

Commits a transaction by GXID.

This function converts GXID to the handle and delegates internal logic to `GTM_CommitTransaction`.

This function is not used at present.

`GTM_PrepareTransaction()`

Prepares a transaction by the handle.

This function marks transaction as "prepared".

This is called from the following codes:

Caller	File & Description
`ProcessPrepareTransactionCommand`	`gtm_txn.c` Used in processing `MSG_TXN_PREPARE` / `MSG_BKUP_TXN_PREPARE` message

`GTM_StartPreparedTransaction()`

Starts to prepare a transaction by the handle.

This function save GID and marks transactions as "prepare in progress".

This is called from the following codes:

Caller	File & Description
`GTM_StartPreparedTransactionGXID`	`gtm_txn.c` Converts GXID to the handle and delegates internal logic
`ProcessStartPreparedTransactionCommand`	`gtm_txn.c` Used in processing `MSG_TXN_START_PREPARED` / `MSG_BKUP_TXN_START_PREPARED` message

`GTM_StartPreparedTransactionGXID()`

Starts to prepare a transaction by GXID.

This function converts GXID to the handle and delegates internal logic to `GTM_StartPreparedTransaction`.

This function is not used at present.

`GTM_GetGIDData()`

Gets information of a prepared transaction. This is called from the following codes:

Caller	File & Description
`ProcessGetGIDDataTransactionCommand`	`gtm_txn.c` Used in processing `MSG_TXN_GET_GID_DATA` message

`GTM_GetAllPrepared()`

Not implemented yet

`GTM_GetStatus()`

Gets transaction status. This is called from the following codes:

Caller	File & Description
`GTM_GetStatusGXID`	`gtm_txn.c` Convert GXID to handle and delegates internal logic

`GTM_GetStatusGXID()`

Gets transaction status by GXID.

This function converts GXID to handle and delegates internal logic to `GTM_GetStatus`.

This function is never used.

`GTM_GetAllTransactions()`

Not implemented yet

`GTM_RemoveAllTransInfos()`

Removes all the transaction information associated with the caller thread and the given backend.

This function removes all the transaction information associated with the caller thread and the given backend from opened transaction list. This also calculates `latestCompletedXid`.

This function is used to clear implicitly aborted transactions when a backend or a GTM proxy disconnect without committing, aborting and preparing.

This is called from the following codes:

Caller	File & Description
`GTM_ThreadMain`	`main.c` Called when the message loop detected a backend disconnection.
`ProcessCommand`	`main.c` Used in processing `MSG_BACKEND_DISCONNECT`

5.5.4.3 gtm_snap.c

This module supplies snapshot handling functions on GTM.

`GTM_GetSnapshotData()`

Not implemented yet

`GTM_GetTransactionSnapshot()`

Gets snapshot for the given transactions.

This function takes transaction snapshots and saves to transaction information object.

Although this function looks supporting `SERIALIZABLE` isolation level, the code is not complete. It doesn't a matter as long as `READ COMMITTED` isolation level is used.

This is called from the following codes:

Caller	File & Description
`ProcessGetSnapshotCommand`	`gtm_snap.c` Used in processing `MSG_SNAPSHOT_GET`
`ProcessGetSnapshotCommandMulti`	`gtm_snap.c` Used in processing `MSG_SNAPSHOT_GET_MULTI`

`GTM_FreeCachedTransInfo()`

Not implemented yet

`GTM_BkupBeginTransactionMulti()`

Updates global transaction information to given ones. This is called from the following codes:

Caller	File & Description
`GTM_BkupBeginTransaction`	`gtm_txn.c` Delegate internal logic

`ProcessBeginTransactionCommandMulti()`

Not implemented yet

`GTM_SaveTxnInfo()`

Saves next GXID to the control file. This is called from the following codes:

Caller	File & Description
`GTM_MakeBackup`	`gtm_backup.c` Used to save next GXID to the backup file
`ServerLoop`	`main.c` Used in shutdown sequence

`GTM_RestoreTxnInfo()`

Restores the next GXID from the control file.

This function also sets `latestCompletedXid`. If the control file is not available, it uses given value from a `gtm` command line option.

This is called from the following codes:

Caller	File & Description
`gtm_standby_restore_next_gxid`	`gtm_standby.c` `GTM_RestoreTxnInfo(NULL,next_gxid);`
`main`	`main.c` Used in start up sequence

`GTM_BkupBeginTransaction()`

Updates global transaction information to given one.

This function delegates internal logic to `GTM_BeginTransaction`.

This is called from the following codes:

Caller	File & Description
`ProcessBkupBeginTransactionCommand`	`gtm_txn.c` Used in processing `MSG_BKUP_TXN_BEGIN` message

`GTM_FreeSnapshotData()`

Releases the snapshot data

This function releases the snapshot data. Please note that the snapshot itself is not freed by this function.

This function is not used at present.

5.5.4.4 `gtm_seq.c`

This module supplies Sequence handling infrastructure. This is one of the most importarnt part of GTM.

An instance of `struct GTM_SeqInfo` is created corresponding to a sequence in a database. The structure is shown in Table 5.4. Most of the member variables has corresponding PostgreSQL's sequence attribute. `gs_ref_count` is a GTM specific variable which manages reference to automatic destruction of the sequence, but there's no case that the sequence is destroyed by the reference count.

The sequence information is stored into `GTMSequences` global hash table. Sequence information is distributed by the hash value of their name so that character string comparison is not needed afterwords.

Table 5.4: Members of `struct GTM_SeqInfo`

Member variable	Description
`gs_key`	Same as `sequence_name` of the sequence in PostgreSQL.
`gs_value`	Same as `last_value` of the sequence in PostgreSQL.
`gs_backedUpValue`	The last value of the sequence which is backuped to the control file.
`gs_init_value`	Same as `start_value` of the sequence in PostgreSQL.
`gs_last_value`	*Updated but not used at present.*
`gs_increment_by`	Same as `increment_bu` of the sequence in PostgreSQL.
`gs_min_value`	Same as `min_value` of the sequence in PostgreSQL.
`gs_max_value`	Same as `max_value` of the sequence in PostgreSQL.
`gs_cycle`	Same as `is_cycled` of the sequence in PostgreSQL.
`gs_called`	Same as `is_called` of the sequence in PostgreSQL.
`gs_ref_count`	The reference count of the sequence object.
`gs_state`	The state of the sequence. This variable could take `SEQ_STATE_ACTIVE`|`SEQ_STATE_DELETED`.
`gs_lock`	The lock for the sequence object.

`GTM_InitSeqManager()`

Initializes global variable for the sequence manager. This is called from the following codes:

Caller	File & Description
`BaseInit`	`main.c` Used in start up sequence

`GTM_SeqOpen()`

Initializes a new sequence.

This function initializes a new sequence. Optionally set the initial value of the sequence.

This is called from the following codes:

Caller	File & Description
`ProcessSequenceInitCommand`	`gtm_seq.c` Used in processing `MSG_SEQUENCE_INIT` message

`GTM_SeqAlter()`

Alter a sequence.

This function alternates current sequence parameters and given one. This function can't rename the sequence.

This is called from the following codes:

Caller	File & Description
`ProcessSequenceAlterCommand`	`gtm_seq.c` Used in processing Process `MSG_SEQUENCE_ALTER` / `MSG_BKUP_SEQUENCE_ALTER` / `MSG_BKUP_SEQUENCE_ALTER` message

`GTM_SeqClose()`

Destroys the given sequence depending on the type of given key

This function can take a sequence name or a database name. If a sequence name is given, it destroys the sequence. If database name is given, it destroys all of the sequences belogs to the database using prefix of full quolified sequence name. Latter functionarity is used when dropping a database.

This is called from the following codes:

Caller	File & Description
`GTM_SeqRename`	`gtm_seq.c` Used in renaming sequence to destroy old sequence info.
`ProcessSequenceCloseCommand`	`gtm_seq.c` Used in processing `MSG_SEQUENCE_CLOSE` / `MSG_BKUP_SEQUENCE_CLOSE` / `MSG_BKUP_SEQUENCE_CLOSE` message

`GTM_SeqRename()`

Rename an existing sequence with a new name

This function creates a new sequence object and copies parameters from old one, which is destroyed then.

This is called from the following codes:

Caller	File & Description
`ProcessSequenceRenameCommand`	`gtm_seq.c` Used in processing `MSG_SEQUENCE_RENAME` / `MSG_BKUP_SEQUENCE_RENAME` / `MSG_BKUP_SEQUENCE_RENAME` message

`GTM_SeqGetNext()`

Gets the next value for the sequence. This is called from the following codes:

Caller	File & Description
`ProcessSequenceGetNextCommand`	`gtm_seq.c` Used in processing `MSG_SEQUENCE_GET_NEXT` / `MSG_BKUP_SEQUENCE_GET_NEXT` / `MSG_BKUP_SEQUENCE_GET_NEXT` message.

`GTM_SeqSetVal()`

Sets values for the sequence. This is called from the following codes:

Caller	File & Description
`ProcessSequenceSetValCommand`	`gtm_seq.c` Used in processing `MSG_SEQUENCE_SET_VAL` / `MSG_BKUP_SEQUENCE_SET_VAL` / `MSG_BKUP_SEQUENCE_SET_VAL` message.

`GTM_SeqReset()`

Resets the sequence. This is called from the following codes:

Caller	File & Description
`ProcessSequenceResetCommand`	`gtm_seq.c` Used in processing `MSG_SEQUENCE_RESET` / `MSG_BKUP_SEQUENCE_RESET` / `MSG_BKUP_SEQUENCE_RESULT` message.

`GTM_SaveSeqInfo()`

Saves all the sequence information.

This is called from the following codes:

Caller	File & Description
`GTM_MakeBackup`	`gtm_backup.c` Used in making a backup file. This function is never used.
`ServerLoop`	`main.c` Used in shutting down sequence to save sequence information for next run.

`GTM_RestoreSeqInfo()`

Restores sequence information from the control file.

- This function is used only by the functions in the same file.
- This is also called from the following codes:

Caller	File & Description
`main`	`main.c` `GTM_RestoreSeqInfo(ctlf);`‘

`GTM_SeqRestore()`

Restores a sequence. This is called from the following codes:

Caller	File & Description
`GTM_RestoreSeqInfo`	`gtm_seq.c` Used in restoring sequences information from a file, which is typically the control file
`gtm_standby_restore_sequence`	`gtm_standby.c` Used in restoring sequences information from a GTM master

`GTM_NeedSeqRestoreUpdate()`

Tests if backup data needs update after the given sequence is touched.

This function is not used at present.

`GTM_WriteRestorePointSeq()`

Writes the restoration point of all the sequences.

This function saves restore point of all sequences to the file, which is typically the control file. The restore point is saved to deal with abnormal termination of GTM, they are advanced 2000 count to current value at maximum to avoid frequent disk write.

This is called from the following codes:

Caller	File & Description
`GTM_WriteRestorePoint`	`gtm_backup.c` Used in making the control file to write sequences information
`GTM_WriteBarrierBackup`	`gtm_backup.c` Used in making a barrier file to write sequences information

5.5.4.5 `gtm_thread.c`

This module supplies thread handling functions.

The things to note here is thread-specific data is stored in `GTM_ThreadInfo` structure. It is allocated in thread-local storage and we can access it using `GetMyThreadInfo` macro. `GetMyThreadInfo` uses pthread function to obtain the pointer and the key `threadinfo_key` is created in `main.c` and `main_proxy.c`.

`MyPort` macro is implicitly used in libpq functions to determine what connection to use. This macro is redefined in GTM to refer to variables in thread-specific data. This change reduces much effort to port libpq into thread model.

Memory contexts are similar to the case of `MyPort`. Many memory contexts such as `TopMemoryContext`, `ErrorContext`, `MessageContext` and `CurrentMemoryContext`, are stored in `GTM_ThreadInfo` for each thread and corresponding macros are also redefined. All of the memory allocated with these context are freed by thread cleanup function when the thread exits.

`GTM_ThreadAdd()`

Adds the given thrinfo structure to the global array.

This function adds the given thrinfo structure to the global array. If the array is full, it expands it automatically.

This is called from the following codes:

Caller	File & Description
GTM_ThreadCreate	gtm_thread.c Used in creating a new thread for a GTM client.
BaseInit	main.c Used in the initialization of the main thread.

`GTM_ThreadRemove()`

Removes given `GTM_ThreadInfo` structure from the global array.

This is called from the following codes:

Caller	File & Description
GTM_ThreadCleanup	gtm_thread.c Used in cleaning up of a thread

`GTM_ThreadJoin()`

Waits for exitting of given thread.

This function is not used at present.

`GTM_ThreadExit()`

Exits this thread immediately.

This function is not used at present.

`GTM_LockAllOtherThreads()`

Locks all the thread information objects of all other threads.

This function is not used at present.

`GTM_UnlockAllOtherThreads()`

Unlocks all the information objects of all the other threads.

This function is not used at present.

`GTM_DoForAllOtherThreads()`

Invokes callback function function for each of other thread information objects. This is called from the following codes:

Caller	File & Description
`ProcessPGXCNodeRegister`	`register_gtm.c` Used in processing `MSG_NODE_REGISTER` message to disconnect connections to GTM slave in all threads when new GTM standby is registered.

`GTM_ThreadCreate()`

Creates a new thread and assigns the given connection to it.

This function creates a new thread, thread local memory contexts and a thread information object, and assign the given connection to the thread.

This is called from the following codes:

Caller	File & Description
`GTMAddConnection`	`main.c` Used in adding new connection from a GTM client.

`GTM_GetThreadInfo()`

Not implemented yet.

5.5.4.6 `gtm_backup.c`

This module supplies backup functions on GTM.

`GTM_WriteRestorePoint()`

Writes restoration point.

This function saves restoration point of GXID and sequences to the control file. The restore point is saved to deal with abnormal termination of GTM. They are advanced 2000 count to current value at maximum.

This is called from the following codes:

Caller	File & Description
`main`	`main.c` Used in start up sequence after all the data are restored.
`ProcessCommand`	`main.c` Used in main loop when backup is needed.
`PromoteToActive`	`main.c` Used in promoting to activate.

`GTM_MakeBackup()`

Saves the next GXID and sequence information to given backup file.

This function is not used at present.

`GTM_SetNeedBackup()`

Set **need backup** flag to true

This function sets **need backup** flag to true. This flag means transaction or sequence information are need to be saved.

This is called from the following codes:

- When the sequence value is created or jumped or removed.
- When the sequence value is incremented and it catches up with backed-up value.
- When GXID is incremented and it catches up with backed-up value.

`GTM_NeedBackup()`

Tests **need backup** flag. This is called from the following codes:

Caller	File & Description
`ProcessCommand`	`main.c` Used in message loop to decide whether do backup or not.

`GTM_WriteBarrierBackup()`

Creates GTM restration point file corresponding to a barrier. This is called from the following codes:

Caller	File & Description
`ProcessBarrierCommand`	`main.c` Used in processing `MSG_BARRIER` / `MSG_BKUP_BARRIER` message

5.5.4.7 `gtm_standby.c`

This module supplies functionalities of GTM Standby.

`gtm_is_standby()`

Not implemented yet.

`gtm_set_standby()`

Not implemented yet.

`gtm_set_active_conninfo()`

Not implemented yet.

`gtm_standby_start_startup()`

Initializes GTM standby startup.

This function connects to GTM active and initialize locks for standby mode.

This is called from the following codes:

Caller	File & Description
`main`	`main.c` Used in start up sequence of GTM standby

`gtm_standby_finish_startup()`

Finishes GTM standby startup

This function closes connection to GTM active for connect-back from master.

This is called from the following codes:

Caller	File & Description
`main`	`main.c` Used in start up sequence of GTM standby

`gtm_standby_restore_next_gxid()`

Gets the next GXID value from GTM active. This is called from the following codes:

Caller	File & Description
`main`	`main.c` Used in start up sequence of GTM standby

`gtm_standby_restore_gxid()`

Restores global transaction information from GTM active. This is called from the following codes:

Caller	File & Description
`main`	`main.c` Used in start up sequence of GTM standby

`gtm_standby_restore_sequence()`

Restores sequence information from GTM active. This is called from the following codes:

Caller	File & Description
`main`	`main.c` Used in start up sequence of GTM standby

`gtm_standby_restore_node()`

Restores node information from GTM active. This is called from the following codes:

Caller	File & Description
`main`	`main.c` Used in start up sequence of GTM standby

`gtm_standby_register_self()`

Registers itself to the GTM active as a “disconnected” node.

This function registers myself to the GTM active as a “disconnected” node before restore starts. This status would be updated later after restoring completion.

This function’s comment saids above, but `PorcessPGXCNodeRegister()` which processes `MSG_NODE_REGISTER` message ignores the status “disconnected”.

This is called from the following codes:

Caller	File & Description
`main`	`main.c` Used in start up sequence of GTM standby

`gtm_standby_activate_self()`

Update node status of itself from "disconnected" to "connected" in GTM active.

This function unregisters myself once, after that it registers myself again as “connected” node.

This is called from the following codes:

Caller	File & Description
`main`	`main.c` Used in start up sequence of GTM standby

`gtm_standby_connect_to_standby()`

Makes a connection to the GTM standby.

This is called from the following codes:

Caller	File & Description
`ServerLoop`	`main.c` Used in GTM active accept loop when a GTM client connected.
`GTM_ThreadMain`	`main.c` Used in GTM active message loop when a GTM slave newly registered.

`gtm_standby_disconnect_from_standby()`

Disconnects connection from GTM standby.

This function disconnects given connection if it is not in standby mode.

This is called from the following codes:

Caller	File & Description
`gtm_standby_reconnect_to_standby`	`gtm_standby.c` Used in reconnection to GTM standby to close old connection.
`ServerLoop`	`main.c` Used in accept loop to cancel connection to GTM standby when GTM can't accept more connection.
`GTM_ThreadMain`	`main.c` Used in message loop to close existing connection to GTM standby when GTM standby is unregistered.

`gtm_standby_reconnect_to_standby()`

Reconnects to GTM standby.

This function closes old connection and reconnects to GTM standby if it is not in standby mode.

This is called from the following codes:

Caller	File & Description
`gtm_standby_check_communication_error`	`gtm_standby.c` Used in checking communication error with standby when it detects an error.

`gtm_standby_check_communication_error()`

Checks if communication with standby made an error.

This function checks whether the communication with standby made an error. If an error is detected, it reconnects to GTM standby.

This is called from everywhere which makes interaction with GTM standby.

`find_standby_node_info()`

Finds “one” GTM standby node info.

This function returns node information of GTM standby. Please note that GTM cannot have multiple GTM standby nodes.

This is called from the following codes:

Caller	File & Description
`gtm_standby_connect_to_standby_int`	`gtm_standby.c` Used in connecting to GTM standby
`ProcessPGXCNodeRegister`	`register_gtm.c` Used in processing `MSG_NODE_REGISTER` message to check the standby node has not been registered yet.

`gtm_standby_begin_backup()`

Notifies GTM standby is beginning backup of GTM active. This is called from the following codes:

Caller	File & Description
`main`	`main.c` Used in start up sequence of GTM standby

`gtm_standby_end_backup()`

Notifies GTM standby is ending backup of GTM active. This is called from the following codes:

Caller	File & Description
`main`	`main.c` Used in start up sequence of GTM standby

`gtm_standby_closeActiveConn()`

Not implemented yet.

`gtm_standby_finishActiveConn()`

Unregisters itself from GTM active.

This function is not used at present.

`ProcessGTMBeginBackup()`

Handler of `MSG_BEGIN_BACKUP`.

This function sets thread status to `GTM_THREAD_BACKUP` and locks all of other thread information objects. This function also sends a result immediately to GTM standby.

This is called from the following codes:

Caller	File & Description
`ProcessCommand`	`main.c` Used in processing `MSG_BEGIN_BACKUP`

`ProcessGTMEndBackup()`

Processes `MSG_END_BACKUP`.

This function resets thread status to `GTM_THREAD_RUNNING` from `GTM_THREAD_BACKUP` and unlocks all of other thread information objects. This function also sends a result immediately to GTM standby.

This is called from the following codes:

Caller	File & Description
`ProcessCommand`	`main.c` Used in processing `MSG_END_BACKUP`

5.5.4.8 `gtm_time.c`

This module supplies timestamp handling functions on GTM.

`GTM_TimestampGetCurrent()`

Gets the current timestamp. This is called from the following codes:

Caller	File & Description
`ProcessBeginTransactionCommand`	`gtm_txn.c` Used to get transaction start timew
`ProcessBeginTransactionGetGXIDCommand`	`gtm_txn.c` Used to get transaction start timew
`ProcessBeginTransactionGetGXIDCommandMulti`	`gtm_txn.c` Used to get transaction start timew

5.5.4.9 `replication.c`

This module supplies controlling the initialization and end of replication process of GTM data. These function is implemented but never used in Postgres-XC, so explanations are omitted.

5.5.4.10 `gtm_stat.c`

This module is not implemented yet.

5.5.4.11 `gtm_stats.c`

This module is not implemented yet.

5.5.5 Configuration Modules

Since many part of configuration related functions seem to be copied from PostgreSQL and these are not point of the GTM. The explanations of these files listed in Table 5.5 are omitted.

Table 5.5: Source files related to configuration

Path
`src/gtm/main/gtm_opt.c`
`src/gtm/config/gtm_opt_handler.c`
`src/gtm/config/gtm_opt_scanner.l`

5.6 GTM Proxy

This module supplies proxy function of GTM to reduce the network traffic to GTM. Please refer to section 1.5.2 for the functional details.

GTM Proxy is implemented as a independent process to the postmaster and the GTM. It means that GTM Proxy has its own binary, configuration file, log file and pid file, and we need to start the process separately.

Many codes are shared with GTM, and codes specific only to GTM Proxy are described here.

5.6.1 Utility functions

5.6.1.1 `proxy_utils.c`

This module provides utility functions in GTM Proxy.

`gtm_standby_check_communication_error()`

No operation.

This function is a dummy function of GTM Proxy to avoid object link problem.

Most of command processing functions are existing only in GTM, but a few are both in GTM and GTM Proxy, which consist of same binary objects. All the command processing functions require calling `gtm_standby_check_communication_error()` for GTM.

5.6.2 Main Program

5.6.2.1 `proxy_main.c`

This file contains main module to run GTM Proxy process.

The GTM Proxy is initialized as following sequence in `main()`. It's very simple: Setup configuration, initialize the main thread, restore information from files, create worker threads and accept connections.

`InitializeGTMOptions()`
↓
(Parse command line options and load configuration file)
↓
`BaseInit()`
↓
`GTM_Recovery_RestoreRegisterInfo()`
↓
(Establish input sockets)
↓
(Setup signal handlers)
↓
(Create worker threads)
↓
`ServerLoop()`

`ServerLoop()` is brought from same name function of `postmaster.c`. It registers itself to GTM, waits for new connections and calls `GTMProxyAddConnection()` to assign them to worker threads. If it detects a signal to abort, the process just exits.

`GTMProxy_ThreadMain()` is a main function and includes the message loop. This function seems to be copied from `postmaster.c:BackendRun()`. First it initializes many things such as a memory context, the connection to GTM, buffer strings and an exception stack. As in the case of PostgreSQL, a signal and an error in child functions is notified with long jump, so an exception handler is registered using `sigsetjmp()`. But the long jump is allowed in very narrow block in `GTMProxy_ThreadMain` because a worker thread handles multiple backends and handles multiple message simultaneously. The block allowed to jump is surrounded `Enable_Longjmp()` and `Disable_Longjmp()`

`GTM_ThreadMain()` reads messages from backends with `ReadCommand()` and calls `ProcessCommand()` for each message to process the "command" message from GTM clients. `ProcessCommand()` dispatches the message as shown in Table 5.6. The processing function dispatched a command message processes the message. Most of the messages are passed to GTM as is, with a proxy header using `GTMProxy_ProxyCommand()`. Some messages are pended to pack into single message using `GTMProxy_CommandPending()`, and `GTMProxy_ProcessPendingCommands()` called after all of the backends are read and handled by `ProcessCommand()` which handles these pended messages. Exceptions are messages `MSG_NODE_REGISTER` and `MSG_NODE_UNREGISTER`. These messages are passed to GTM with the host name of the GTM Proxy using `GTMProxy_ProxyPGXCNodeCommand()`. So the proxy functions just put the message into libpq buffer. After all messages are ready to send, GTM Proxy append `MSG_DATA_FLUSH` message that has type "F" and flush the buffer.

The messages sent by the proxy functions are stored into a linked list, GTM Proxy handles the response for each message in the list. The response is read from GTM with `GTMPQgetResult()`, and it is handled by `ProcessResponse()`. `ProcessResponse()` finds appropriate connection to the sent message and sends back the response message. If the sent message is packed message, unpacks it and sends corresponding response to each backend.

If the message loop detects disconnection of a backend, it sends `MSG_BACKEND_DISCONNECT` message to GTM with `GTMProxy_CommandPending()`. The connection information is removed from the thread after the end of the message loop.

In an opposite case that a new connection is assigned to the thread, the connection handshaked and reading socket array is reconstructed at the beginning of the message loop. It means that there's no traffic in existing connection, the new connection spoils one second passes at maximum.

Table 5.6: GTM Proxy message processing functions

Message	Processing function	Proxy function
MSG_NODE_REGISTER	ProcessPGXCNodeCommand()	GTMProxy_ProxyPGXCNodeCommand()
MSG_NODE_UNREGISTER		GTMProxy_ProxyPGXCNodeCommand()
MSG_TXN_BEGIN_GETGXID	ProcessTransactionCommand()	GTMProxy_CommandPending()
MSG_TXN_COMMIT		GTMProxy_CommandPending()
MSG_TXN_ROLLBACK		GTMProxy_CommandPending()
MSG_TXN_BEGIN		*Not supported*
MSG_TXN_GET_GXID		*Not supported*
MSG_TXN_BEGIN_GETGXID_ AUTOVACUUM		GTMProxy_ProxyCommand()
MSG_TXN_PREPARE		GTMProxy_ProxyCommand()
MSG_TXN_START_PREPARED		GTMProxy_ProxyCommand()
MSG_TXN_GET_GID_DATA		GTMProxy_ProxyCommand()
MSG_TXN_COMMIT_PREPARED		GTMProxy_ProxyCommand()
MSG_SNAPSHOT_GET	ProcessSnapshotCommand()	GTMProxy_CommandPending()[4]
MSG_SNAPSHOT_GXID_GET		*Not supported*
MSG_SEQUENCE_INIT	ProcessSequenceCommand()	GTMProxy_ProxyCommand()
MSG_SEQUENCE_ALTER		GTMProxy_ProxyCommand()
MSG_SEQUENCE_GET_NEXT		GTMProxy_ProxyCommand()
MSG_SEQUENCE_SET_VAL		GTMProxy_ProxyCommand()
MSG_SEQUENCE_RESET		GTMProxy_ProxyCommand()
MSG_SEQUENCE_CLOSE		GTMProxy_ProxyCommand()
MSG_SEQUENCE_RENAME		GTMProxy_ProxyCommand()
MSG_BARRIER	ProcessBarrierCommand()	GTMProxy_ProxyCommand()

5.6.2.2 `proxy_thread.c`

This module supplies thread handling function in GTM Proxy. This module is similar to `gtm_thread.c`, but please note that GTM Proxy doesn't create new thread per GTM client. GTM Proxy adopts worker thread model, a worker thread handles multiple backends.

`GTMProxy_ThreadAdd()`

Adds the given thrinfo structure to the global array

This function adds the given thrinfo structure to the global array. If the array is full, it is expanded automatically.

[4] If the message does not allow grouping, `GTMProxy_ProxyCommand()` is called.

This is called from the following codes:

Caller	File & Description
`GTMProxy_ThreadCreate`	`proxy_thread.c` Used in creating new thread
`BaseInit`	`proxy_main.c` Used in start up sequence to register the main thread.

`GTMProxy_ThreadRemove()`

Removes the given thrinfo structure from the global array.

This function is not used at present.

`GTMProxy_ThreadJoin()`

Waits given thread to exit.

This function is not used at present.

`GTMProxy_ThreadExit()`

Exits this thread immediately.

This function is not used at present.

`GTMProxy_ThreadCreate()`

Creates a new thread and assigns it to the given thread information slot.

This function creates a new thread, thread local memory contexts and a thread information object, and then assigns the thread to the given thread information slot.

Please note that the comment says that the function assigns connection to thread, it's bogus.

This is called from the following codes:

Caller	File & Description
`main`	`proxy_main.c` Used in start up sequence to create worker threads.

`GTMProxy_GetThreadInfo()`

Not implemented yet.

`GTMProxy_ThreadAddConnection()`

Adds a connection to a worker thread.

This function adds the given connection to the thread selected by a round-robin manner. The caller is responsible only for accepting the connection. Other things including the authentication is done by the worker thread when it finds a new entry in the connection list.

This function also assigns the connection to the connection ID. It is thread local ID and it is distinct from index of the connection information array.

This is called from the following codes:

Caller	File & Description
`GTMProxyAddConnection`	`proxy_main.c` Used in adding new connection from a GTM client.

`GTMProxy_ThreadRemoveConnection()`

This function removes the given connection from the assignment of a worker thread. It chinks a gap in connection information array made by removing the connection. The index of connections may change.

This function also releases the connection ID assigned to given connection.

This is called from the following codes:

Caller	File & Description
`GTMProxy_ThreadMain`	`proxy_main.c` Used in the message loop when it detects disconnection of a GTM client.

5.6.3 Configuration Modules

Since many part of configuration related functions seem to be copied from PostgreSQL and these are not point of the GTM Proxy. The explanations of these files listed in Table 5.7 are omitted.

Table 5.7: Source files related to configuration

Path
`src/gtm/proxy/gtm_proxy_opt.c`

5.7 `Pgxc_ctl` module

This section describes internal structure of `pgxc_ctl` module.

For the usage and tutorial of this module, see Part II of this report.

5.7.1 Outline of the module

Source material of this module will be found in the directory `contrib/pgxc_ctl`

This module provide the following feature:

- Configure Postgres-XC cluster including gtm master/slave, gtm_proxies, coordinator master/slave and datanode master/slave.
- Initialize Postgres-XC cluster based upon the configuration definition.
- Start and stop Postgres-XC cluster.
- Failover each component if the slave is configured and running.
- Monitor if each component is running.
- Add and remove components.
- Other command interface needed for Postgres-XC cluster operation.

`pgxc_ctl` is essentially a `ssh` wrapper to run shell script at remote nodes to perform each of the above operations.

The following section describes outline of `pgxc_ctl` source code structure, its general flow and each component's structure.

5.7.2 `pgxc_ctl` source code structure

Table 5.8 (page 127) is the list of `pgxc_ctl` source file.

5.7.3 Outline of `pgxc_ctl` behavior

The outline of `pgxc_ctl` is as follows:

1. Handles command line options (`main():pgxc_ctl.c`) and environment file options (`setup_my_env():pgxc_ctl.c`).
2. Begins logging (`startLog():pgxc_ctl.c`).
3. Reads and check configuration.
   (a) Loads default configuration file (bash script) (`prepare_pgxc_ctl_bash():pgxc_ctl.c`).
   (b) Loads configuration file (`build_configuration_path():pgxc_ctl.c`).
   (c) Reads configuration variables (`read_configuration():pgxc_ctl.c`).
   (d) Checks configuration variables (`check_configuration():config.c`).
4. Reads one line of command and handles it. (`do_command():do_command.c`).

Table 5.8: pgxc_ctl source file list

Source File	Description
bash_handler.c	bash script handler module of Postgres-XC configuration and operation tool.
bash_handler.h	Header file to define bash_handler.c interface.
config.c	Handles pgxc_ctl configuration.
config.h	Header file to define config.c interface.
coord_cmd.c	Coordinator operation module.
coord_cmd.h	coord_cmd.c interface definition.
datanode_cmd.c	Datanode operation module
datanode_cmd.h	datanode_cmd.c interface definition.
do_command.c	High level command handler.
do_command.h	do_command.c interface definition.
do_shell.c	Infrastructure for ssh command preparation and execution.
do_shell.h	do_shell.c interface definition.
gtm_cmd.c	GTM operation module.
gtm_cmd.h	gtm_cmd.c interface definition.
gtm_util.c	Command handler with GTM.
gtm_util.h	gtm_util.c interface definition.
make_signature	Shell script to build signature file and configuration file template embedded in pgxc_ctl_bash.c.
mcxt.c	Memory handler.
monitor.c	Monitoring Postgres-XC components.
monitor.h	monitor.c interface definition.
pgxc_ctl.bash	Original pgxc_ctl module in bash script. This is useful to understand pgxc_ctl behavior.
pgxc_ctl_bash.c	This file contains default configuration values and bash script to read the configuration file. Generated by make_signature.
pgxc_ctl.c	Main module.
pgxc_ctl.h	Main module interface definition.
pgxc_ctl_bash.c	Module holds configuration template. Generated by make_signature.
pgxc_ctl_bash_2	Original template to be embedded into pgxc_ctl_bash.c
pgxc_ctl_conf_part	Original template to be embedded into pgxc_ctl_bash.c
pgxc_ctl_log.c	Logging module.
pgxc_ctl_log.h	pgxc_ctl_log.c interface definition.
signature.h	Holds signature information to test if working environment matches make_signature generation.
utils.c	Miscellaneous utility functions.
utils.h	utils.c interface definition.
variables.c	Variable module.
variables.h	variable.c interface definition.
varnames.h	Definition of variable symbol and variable name string.

5.7.4 Inside each program files

This section describes entries in each program files. Header file description may not be given if it contains only function entry declarations.

In each function description, you will find that sometimes execution is divided into two steps, **preparation** and **execution**. This allows to run similar `ssh` script in parallel at more than one servers.

This saves much time for configuration, start and stop whole Postgres-XC cluster.

5.7.4.1 `bash_handler.c`

This module consists of the following functions.

`install_pgxc_ctl_bash()`

Builds shell script which contains default configuration parameters (variable `pgxc_ctl_conf_prototype`) and bash functions to extract configuration variables (variable `pgxc_ctl_bash_script`).

This function is called from the following codes:

Caller	File & Description
`read_configuration()`	`pgxc_ctl.c` Used to read configuration variables.
`prepare_pgxc_ctl_bash()`	`pgxc_ctl.c` Used to extract `bash` script for default configuration and bash functions.

`uninstall_pgxc_ctl_bash()`

Removes `bash` script installed by `instal_pgxc_ctl_bash()`.

This is called from the following code:

Caller	File & Description
`read_configuration()`	`pgxc_ctl.c` Used to read configuration variables.

`read_config_file()`

Runs configuration file as `bash` script and reads configuration variables.

This is for work and is not used by other codes now.

5.7.4.2 config.c

This is configuration parser module.

As defined in pgxc_ctl_bash_script[] variable defined in pgxc_ctl_bash.c. pgxc_ctl will read one variable value in one line as

varname value value ...

More than one value may be defined if the variable is an array. If the variable is defined as a scalar, only the first value will be taken.

get_word()

This function takes line buffer, scans it, sets a token found and returns the next scanning point. This function destroys input line string and returns the token address within the original line buffer. The caller must must copy the found token for later use.

This function is used in various place to parse configuration variable output. Macro GetToken() may be defined in several module for the shortcut to this function.

parse_line()

Parses a line of configuration script output and sets the variable and its value to internal variable infrastructure.

This is used only within config.c module.

parse_line_selet()

This function checks if the configuration script output line is one of the specified set of variable value and set it to internal variable infrastructure. Used within config.c module.

read_vars()

Reads configuration script output and sets all the variables and their name to internal variable infrastructure.

read_selected_vars()

Reads configuration script output and sets variables and their values to the internal variable infrastructure only those matches specified set of variables.

This is called from the following code:

Caller	File & Description
setup_my_env()	pgxc_ctl.c Used to setup environmental variables.

`install_conf_prototype()`

This function builds the configuration file prototype.

This is for the future usage and is not used at present.

`addServer()`

This function checks if the given server has already been in the server list and add it if it is new to the list.

This is called from the following code:

Caller	File & Description
`makeServerList`	`config.c` Used to build list of servers of the cluster.

`makeSererList()`

This function builds Postgres-XC server list in internal variable infrastructure.

This is called from the following codes:

Caller	File & Description
`check_configuration()`	`config.c` Used in initial configuration check and build server list.
`add_gtmSlave()`	`gtm_cmd.c` Used in adding gtm slave.
`add_gtmProxy()`	`gtm_cmd.c` Used in adding gtm proxy.
`remove_gtmProxy()`	`gtm_cmd.c` Used in removing gtm proxy.

`is_none()`

This function is used to check if a given name (node name, server name, etc.) is `NULL`. So far, "`none`" and "`N/A`" are interrupted as `NULL`.

This is very common function and called from various functions. The usage is quite obvious and no caller information is given here.

`emptyGtmSlave()`

Initializes gtm slave information to `NULL`.

This is called from the following codes:

Caller	File & Description
`handle_no_slaves()`	`config.c` Used in initial configuration check to handle `NULL` values.

`checkConfiguredAndSize()`

Checks if all the specified variable has same number of array member.

Used inside `config.c` module.

`checkSPecificResourceConflict()`

Checks resource conflict in the configuration. Component name, port and work directory are checked.

This is called from the following codes:

Caller	File & Description
`add_gtmSlave()`	`gtm_cmd.c` Used in adding gtm slave.
`add_gtmProxy()`	`gtm_cmd.c` Used in adding a gtm proxy.

`checkNameConflict()`

Tests if a given name does not conflict with others.

This is called from the following codes:

Caller	File & Description
`add_coordinatorMaster()`	`coord_cmd.c` Used in adding a coordinator master.
`checkSPecificResourceConflict()`	`config.c` Used in checking all the resource conflict.
`add_datanodeMaster()`	`datanode_cmd.c` Used in adding a datanode master.

`checkPortConflict()`

Tests if a given port at the given host conflicts with other ports in the host.

This is called from the following codes:

Caller	File & Description
add_coordinatorMaster()	coord_cmd.c Used in adding a coordinator master.
add_coordinatorSlave()	coord_cmd.c Used in adding a coordinator slave.
checkSPecificResourceConflict()	config.c Used in checking all the resource conflict.
add_datanodeMaster()	datanode_cmd.c Used in adding a datanode master.
add_datanodeSlave()	datanode_cmd.c Used in adding a datanode slave.

`checkDirConflict()`

Tests if a given directory at the given host conflicts with other directories in the host.

This is called from the following codes:

Caller	File & Description
add_coordinatorMaster()	coord_cmd.c Used in adding a coordinator master.
add_coordinatorSlave()	coord_cmd.c Used in adding a coordinator slave.
checkSPecificResourceConflict()	config.c Used in checking all the resource conflict.
add_datanodeMaster()	datanode_cmd.c Used in adding a datanode master.
add_datanodeSlave()	datanode_cmd.c Used in adding a datanode slave.

`checkResourceConflict()`

Tests if there's any conflict among source and destination checks duplicate in names, ports and rectories.

This is called from the following codes:

Caller	File & Description
verifyResource()	config.c Used in verifying resource conflict at running.

`verifyResource()`

This checks whole Postgres-XC resource is configured to run as a cluster.

Caller	File & Description
check_configuration()	config.c Used in checking if minimum components are configured as Postgres-XC cluster.

check_configuration()

Checks if minimum components are configured as Postgres-XC cluster.

Called from the following codes:

Caller	File & Description
main()	pgxc_ctl.c Used in initial configuration read.

backup_configuration()

This function backs up configuration file to a remote site as specified. pgxc_ctl adds updated configuration line at the last of the configuration file when the cluster changes by failover, adding and removing nodes. This feature helps to maintain pgxc_ctl configuration file safely.

It is called by the following codes:

Caller	File & Description
`add_coordinatorMaster()`	`coord_cmd.c` Used in adding a coordinator master.
`add_coordinatorSlave()`	`coord_cmd.c` Used in adding a coordinator slave.
`remove_coordinatorMaster()`	`coord_cmd.c` Used in removing a coordinator master.
`remove_coordinatorSlave()`	`coord_cmd.c` Used in removing a coordinator slave.
`add_datanodeMaster()`	`datanode_cmd.c` Used in adding a datanode master.
`add_datanodeSlave()`	`datanode_cmd.c` Used in adding a datanode slave.
`remove_datanodeMaster()`	`datanode_cmd.c` Used in adding a datanode master.
`remove_datanodeSlave()`	`datanode_cmd.c` Used in adding a datanode slave.
`add_gtmSlave()`	`gtm_cmd.c` Used in adding gtm slave.
`remove_gtmSlave()`	`gtm_cmd.c` Used in adding gtm slave.
`failover_gtm()`	`gtm_cmd.c` Used in gtm failover.
`remove_gtmProxy()`	`gtm_cmd.c` Used in removing a gtm proxy.

`getNodeType()`

Returns node type (gtm, gtm_proxy, coordinator, datanode, or server name).

It is called by the following codes:

Caller	File & Description
`monitor_something()`	`monitor.c` Used in getting the type of the given name to determine how to monitor the target.
`kill_something()`	`do_command.c` Used in determining what is going to be killed.
`show_config_something()`	`do_command.c()` Used in determining what configuration to show about.
`do_clean_command()`	`do_command.c` Used in determining what kind of resource going to cleanup.

`getDefaultWalSender()`

Determine maximum number of WAL sender process.

It is called by the following codes:

Caller	File & Description
`add_coordinatorSlave()`	`coord_cmd.c` Used in adding coordinator slave.
`add_datanodeSlave()`	`datanode_cmd.c` Used in adding datanode slave.

5.7.4.3 `coord_cmd.c`

This module performs operation on coordinators and consists of the following functions.

`init_coordinator_master_all()`

This is a wrapper function for `init_coordinator_master()` to initialize all the coordinator master defined in the configuration file.

It is called by the following codes:

Caller	File & Description
`init_all()`	`do_command.c` Used in initializing everything defined in the configuration file.
`do_init_command()`	`do_command.c` Used in handling `init coordinator all` command and `init coordinator master all` command.

`prepare_initCoordinatorMaster()`

This function prepares internal `cmd_t` structure to describe the step for the initialization of one coordinator master.

The step includes the following:

1. Checks if the target coordinator master is not running.
2. Cleans up the work directory.
3. Run `initdb`.
4. Determines which gtm_proxy to use, or to use gtm directly.
5. Constructs `postgresql.conf`.
6. Constructs WAL shipping replication if the slave is configured.
7. Constructs `pg_hba.conf` file.

.

It is called from `init_coordinator_master` in `coord_cmd.c` to initialize one or more than one coordinator masters.

`init_coordinator_master()`

This function initializes specified coordinator masters, which can be one or more than one.

The step is as follows:

1. Prepares initialization steps for all the coordinator masters specified using `prepare_initCoordinatorMaster()`.
2. Performs the steps using `doCmdList()` function defined in `do_shell.c`

`init_coordinator_slave_all()`

This function initializes all the coordinator slaves defined in the configuration file.

It is a wrapper of `init_coordinator_slave()` function described below.

It is called from the following codes;

Caller	File & Description
`init_all()`	`do_command.c` Used in initializing everything defined in the configuration file.
`do_init_command()`	`do_command.c` Used in handling `init coordinator all` command and `init coordinator slave all` command.

`prepare_initCoordinatorSlave()`

This function prepares internal `cmd_t` structure to describe the step for the initialization of one coordinator slave.

The step includes the following:

1. Checks if corresponding coordinator master is configured.
2. Cleans up and reinitialize the work directory.
3. Checks if the coordinator master is running. It is necessary to build the base backup of the master using `pg_basebackup` utility.
4. Builds the base backup. The source code has additional codes to build the base backup with primitive way.

5. Builds `recovery.conf` file at the slave.

6. Configures `postgresql.conf` file at the slave.

.

It is called from `init_coordinator_slave()` in `coord_cmd.c` to initialize one or more than one coordinator slaves.

`init_coordinator_slave()`

This function initializes specified coordinator slaves, which can be one or more than one.

The step is as follows:

1. Checks if coordinator slave is configured.

2. Prepares initialization steps for all the coordinator slaves specified using `prepare_initCoordinatorSlave()`.

3. Performs the steps using `doCmdList()` function defined in `do_shell.c`

`configure_nodes_all()`

This is a wrapper function for `configure_nodes()` and is called form `init_all()` function to perform `init all` command.

`configure_nodes_all()`

This function issues `CREATE NODE` and `ALTER NODE` statement at all the coordinators to configure Postgres-XC cluster at each coordinator.

This function is a wrapper of `configure_nodes()` and is called from `init_all()` function at `do_command.c` module to perform `init all` command.

`configure_nodes()`

This function runs `CREATE NODE` and `ALTER NODE` statement at give coordinators to configure Postgres-XC cluster at each coordinator.

This function uses `prepare_configureNode()` to set up needed steps for each coordinator. It is called from `do_configure_command()` at `do_command.c` module.

`prepare_configureNode()`

This function prepares necessary step to configure one coordinator with `CREATE NODE` and `ALTER NODE` statement. It is called from `configure_nodes()` at `coord_cmd.c`.

`ALTER NODE` statement is used to update own coordinator information.

`kill_coordinator_master_all()`

This is a wrapper for `kill_coordinator_master()`. It is for the future usage and is not used now. `kill_coordinator_master()` is used instead.

`prepare_killCoordinatorMaster()`

Build necessary step to kill one coordinator master. It is called by `kill_coordinator_master()` described below.

`kill_coordinator_master()`

This function kills specified coordinator masters, which can be one or more than one, using `prepare_killCoordinatorMaster()` in `coord_cmd.c`.

`kill_coordinator_slave_all()`

This is a wrapper for `kill_coordinator_slave()`. It is for the future usage and is not used now. `kill_coordinator_slave()` is used instead.

`prepare_killCoordinatorSlave()`

Build necessary step to kill one coordinator slave. It is called by `kill_coordinator_slave()` described below.

`kill_coordinator_slave()`

This function kills specified coordinator slaves, which can be one or more than one, using `prepare_killCoordinatorSlave()` in `coord_cmd.c`.

`prepare_cleanCoordinatorMaster()`

This function prepares necessary step to cleanup the working directory of a given coordinator master.

It is called from the following codes:

Caller	File & Description
`clean_coordinator_master()`	`coord_cmd.c` Used to clean the work directory of one or more than one coordinator masters.
`do_clean_command()`	`docommand.c` Used in performing `clean` command.

`clean_coordinator_master()`

This function cleans up working directory of one or more than one coordinator master specified. Necessary steps are build using `prepare_cleanCoordinatorMaster()` in `coord_cmd.c`.

It is called from the following codes:

Caller	File & Description
`clean_coordinator_master_all()`	`coord_cmd.c` Used to clean the work directory of all the coordinator masters.
`do_clean_command()`	`docommand.c` Used in performing `clean` command.

`clean_coordinator_master_all()`

This is a wrapper for `clean_coordinator_master()` and is called from `do_clean()` in `do_command.c` to cleanup all the coordinator master's work directories.

`prepare_cleanCoordinatorSlave()`

This function prepares necessary step to cleanup the working directory of a given coordinator slave.

It is called from the following codes:

Caller	File & Description
`clean_coordinator_slave()`	`coord_cmd.c` Used to clean the work directory of one or more than one coordinator masters.
`do_clean_command()`	`docommand.c` Used in performing `clean` command.

`clean_coordinator_slave()`

This function cleans up working directory of one or more than one coordinator master specified. Necessary steps are build using `prepare_cleanCoordinatorSlave()` in `coord_cmd.c`.

It is called from the following codes:

Caller	File & Description
`clean_coordinator_slave_all()`	`coord_cmd.c` Used to clean the work directory of all the coordinator masters.
`do_clean_command()`	`docommand.c` Used in performing `clean` command.

`clean_coordinator_slave_all()`

This is a wrapper for `clean_coordinator_slave()` and is called from `do_clean()` in `do_command.c` to cleanup all the coordinator slave's work directories.

`add_coordinatorMaster()`

This function adds coordinator master to Postgres-XC cluster.

Because coordinator master would be added one after another, handling to add more than one coordinator masters may not make a good sense. It is done in series with separate `pgxc_ctl` command.

For steps done, see section 25.4 in part II on page 284.

`add_coordinatorSlave()`

This function adds a coordinator slave to specified coordinator master.

Because a coordinator slave would be added one after another, adding more than one coordinator slaves may not make a good sense. It is done in series with separate `pgxc_ctl` command.

For steps done, see section 25.5 in part II on page 285.

`remove_coordinatorMaster()`

This function removes one coordinator master from Postgres-XC cluster.

For steps done, see section 26.3 in part II on page 290.

`remove_coordinatorSlave()`

This function removes one coordinator slave from Postgres-XC cluster.

For steps done, see section 26.4 in part II on page 290.

`start_coordinator_master_all()`

This function is a wrapper to the function `start_coordinator_master()` in `coord_cmd.c` to start all the coordinator master.

This is called from the following codes:

Caller	File & Description
`init_all()`	`do_command.c` Used in starting everything after the whole cluster initialization.
`start_all()`	`do_command.c` Used in starting everything.
`do_start_command()`	`do_command.c` Used in handling `start coordinator all` command and `start coordinator master all` command.

`prepare_startCoordinatorMaster()`

Prepares necessary steps to start one coordinator master.

This is called from `start_coordinator_master()` in `coord_cmd.c` to start one or more than one coordinator master.

`start_coordinator_master()`

Starts one ore more than one coordinator master in parallel.

This is called from the following codes;

Caller	File & Description
`add_coordinatorMaster()`	`coord_cmd.c` Used in starting added coordinator master.
`start_coordinator_master_all()`	`coord_cmd.c` Used in starting every coordinator master.
`do_start_command()`	`do_command.c` Used in handling `start coordinator master all` command and `start coordinator` command.

`start_coordinator_slave_all()`

This function is a wrapper to the function `start_coordinator_slave()` in `coord_cmd.c` to start all the coordinator master.

This is called from the following codes:

Caller	File & Description
`init_all()`	`do_command.c` Used in starting everything after the whole cluster initialization.
`start_all()`	`do_command.c` Used in starting everything.
`do_start_command()`	`do_command.c` Used in handling `start coordinator all` command and `start coordinator slave all` command.

`prepare_startCoordinatorSlave()`

Prepares necessary steps to start one coordinator master. This is called from `start_coordinator_slave()` in `coord_cmd.c` to start one or more than one coordinator master.

`start_coordinator_slave()`

Starts one or more than one coordinator slaves in parallel. This is called from the following codes;

Caller	File & Description
add_coordinatorSlave()	coord_cmd.c Used in starting added coordinator slave.
start_coordinator_slave_all()	coord_cmd.c Used in starting every coordinator slave.
do_start_command()	do_command.c Used in handling start coordinator slave all command and start coordinator command.

stop_coordinator_master_all()

This function is a wrapper to the function stop_coordinator_master() in coord_cmd.c to stop all the coordinator master. This is called from the following codes:

Caller	File & Description
stop_all()	do_command.c Used in stopping everything.
do_stop_command()	do_command.c Used in handling stop coordinator all command and stop coordinator master all command.

prepare_stopCoordinatorMaster()

Prepares necessary steps to stop one coordinator master. This is called from stop_coordinator_master() in coord_cmd.c to stop one or more than one coordinator master.

stop_coordinator_master()

Stops one ore more than one coordinator masters in parallel. This is called from the following codes;

Caller	File & Description
stop_coordinator_master_all()	coord_cmd.c Used to stop every coordinator master.
do_stop_command()	do_command.c Used in handling stop coordinator master all command and stop coordinator command.

stop_coordinator_slave_all()

This function is a wrapper to the function stop_coordinator_slave() in coord_cmd.c to stop all the coordinator slaves. This is called from the following codes:

Caller	File & Description
`stop_all()`	`do_command.c` Used in stopping everything.
`do_stop_command()`	`do_command.c` Used in handling `stop coordinator all` command and `stop coordinator slave all` command.

`prepare_stopCoordinatorSlave()`

Prepares necessary steps to stop one coordinator slave. This is called from `stop_coordinator_slave()` in `coord_cmd.c` to stop one or more than one coordinator slave.

`stop_coordinator_slave()`

Stops one ore more than one coordinator slaves in parallel. This is called from the following codes;

Caller	File & Description
`stop_coordinator_slave_all()`	`coord_cmd.c` Used to stop all the coordinator slaves.
`do_stop_command()`	`do_command.c` Used in handling "`stop coordinator slave all`" command and "`stop coordinator`" command.

`failover_coordinator()`

This function promotes specified coordinator slave to master. This can handle more than one coordinator failover but is done in series, not in parallel.

`failover_oneCoordinator()` takes care of each coordinator failover as described next.

This is called from `do_failover_command()` in `do_command.c` to handle `failover coordinator` command.

`failover_oneCoordinator()`

Performs one coordinator slave promotion to master. This is called from `failover_coordinator()` in `do_command.c`.

The steps done in this function is described in section 24.2 of part II on page 280.

`show_config_coordMasterSlaveMulti()`

Shows coordinator master and slave configuration for given names.

This is called from `show_configuration` in `do_command.c` to handle `show configuration` command.

`show_config_coordMasterMulti()`

Shows configuration of one or more than one coordinator master. This is called by the following codes:

Caller	File & Description
`show_config_coordMasterSlaveMulti()`	`coord_cmd.c` Used to stop every coordinator slave.
`show_configuration()`	`do_command.c` Used in handling `show configuration` command.

`show_config_coordMaster()`

Shows configuration of given coordinator master. This is called from the following codes:

Caller	File & Description
`show_config_coordMasterSlaveMulti()`	`coord_cmd.c`
`show_config_coordMasterMulti()`	`coord_cmd.c`
`show_config_something()`	`do_command.c`
`show_config_host()`	`do_command.c`

`show_config_coordSlave()`

Shows configuration of given coordinator slave. This is called from the following codes:

Caller	File & Description
`show_config_coordMasterSlaveMulti()`	`coord_cmd.c`
`show_config_coordSlaveMulti()`	`coord_cmd.c`
`show_config_something()`	`do_command.c`
`show_config_host()`	`do_command.c`

`check_AllCoordRunning()`

Checks if all the coordinator masters are running. This is called from `add_coordinatorMaster()` in `coord_cmd.c` to check if coordinator master can be added to Postgres-XC cluster.

5.7.4.4 `datanode_cmd.c`

`datanode_cmd.c` structure is very similar to `coord_cmd.c`. There will be no difficulty to understand the implementation with the last section's description.

5.7.4.5 do_command.c

do_command.c performs most of pgxc_ctl command.

do_echo_command()

Performs echo command. This is called from do_singleLine() in do_command.c.

do_prepareConfFile()

Builds template of pgxc_ctl configuration file. This is called from do_singleLine() in do_command.c to perform prepare command.

do_deploy()

Performs deploy command. This is called from do_singleLine() in do_comamnd.c to perform deploy command.

deploy_xc()

Performs deploy command. This is called from do_deploy() in do_command.c.

do_set()

Sets (set of) value to specified variable. This functions sets up any variable even if it is not pre-defined in the configuration file. This is called from do_singleline() in do_command.c to perform set command.

do_failover_command()

Performs failover command and called from do_singleline() in do_command.c

do_reconnect_command()

Performs reconnect command and called from do_singleline() in do_command.c.

do_kill_command()

Performs kill command and called from do_singleline() in do_command.c.

init_all()

Performs init all command and called from do_init() command in do_command.c.

do_init_command()

Performs init command and called from do_singleline() in do_command.c.

5.7.4.6 do_shell.c

This module is a basic infrastructure to run various shell script.

Many functions in this module use two structures called `cmd_t` and `cmdList_t` as interfaces to the caller.

`cmd_t` defines series of shell script which can run either locally or at remote servers. Shell commands defined in this structure are executed in series.

`cmdList_t` consists of one or more than one `cmd_t` structures, which are executed in parallel.

Definition of these structures are given in `do_shell.h`.

Functions in this module are as follows:

`do_shell_SigHandler()`

Signal handler during command execution.

`createLocalFileName()`

Creates path to stdin, stdout, or stderr file for each piece of local shell command. This is called from the following codes;

Caller	File & Description
`doImmediate()`	`do_shell.c`
`prepareStdout()`	`do_shell.c`
`show_Resource()`	`do_command.c`
`do_stop_command()`	`do_command.c` Used in performing "`show resource`" command.

`createRemoteFileName()`

Creates path to stdin, stdout, or stderr file for each piece of remote shell command. This is called from the following codes;

Caller	File & Description
`doImmediate()`	`do_shell.c`
`prepareStdout()`	`do_shell.c`

`doImmediateRaw()`

This function runs any command foreground locally. Argument to this function is same as `printf()` so that caller do not have to prepare command string using `spirntf()`.

This is called from the following codes:

Caller	File & Description
add_coordinatorMaster()	coord_cmd.c
doImmediate()	do_shell.c
touchStdout()	do_shell.c
doCmdEl()	do_shell.c
doCmdList()	do_shell.c
do_cleanCmdEl()	do_shell.c
doConfigBackup()	do_shell.c
show_Resource()	do_command.c
do_singleLine()	do_command.c
add_datanodeMaster()	datanode_cmd.c

`pgxc_popen_wRaw()`

Begins local shell command using `popen()` so that the caller can write data to the shell as `stdin`.

Argument to this function is same as `printf()` so that caller do not have to prepare command string using `spirntf()`.

This is a utility function called from various codes.

`pgxc_popen_w()`

Begins remote shell command using `popen()` so that the caller can write data to the shell as `stdin`.

Argument to this function is same as `printf()` with remote host name so that caller do not have to prepare command string using `spirntf()`.

This is a utility function called from various codes.

`doImmediate()`

This function executes one remote shell command at foreground.

Argument to this function is same as `printf()` with additional host name and `stdin` file name. Caller must prepare the contents of `stdin` file before calling.

This is a utility function called from various codes.

`initCmdList()`

Allocates and initializes `cmdList_t` structure.

This is a utility function called from various codes.

`initCmd()`

Allocates and initializes `cmd_t` structure.

This is a utility function called from various codes.

`clearStdin()`

Removes `stdin` file defined in `cmd_t` structure and cleans its entry in `cmd_t` structure.

This is a utility function called from various codes.

`doCmd()`

Executes shell commands defined in `cmd_t` structure one after another.

This is a utility function called from various codes.

`allocActualCmd()`

Allocates a buffer to store actual shell command to run into `cmd_t` structure. This is used to launch actual shell command with redirection of `stdin`, `stdout` and `stderr` as needed.

This is a utility function called from various codes.

`doCmdEl()`

Executes one remote or local shell command defined in `cmd_t` structure. This function resolves `stdin`, `stdout` and `stderr` redirection as needed.

This is a utility function called from various codes.

`doCmdList()`

Executes shell commands defined in `cmdList_t` structure in parallel.

This is a utility function called from various codes.

`appendCmdEl()`

Appends given `cmd_t` structure at the end of specified `cmd_t` structure.

This is a utility function called from various codes.

`do_cleanCmdEl()`

Releases given `cmd_t` structure. Please note that this function does not take care of internal `cmd_t` structure chain. This is handled by `do_cleanCmd()` function described below. Before freeing all the memory allocated, this removes `stdin` and `stdout` file defined in this structure.

This is a utility function called from various codes.

`do_cleanCmd()`

Releases all the shell command chain in the given `cmd_t` structure. Each command chain element is released by do_cleanCmdEl().

This is a utility function called from various codes.

do_cleanCmdList()

Releases all the shell command defined in the given cmdList_t structure. Each cmd_t structure defined in it is released by do_cleanCmd() function described above.

This is a utility function called from various codes.

addCmd()

Adds cmd_t structure to cmdList_t structure.

cleanLastCmd()

Releases the mast cmd_t structure in the given cmd_t shell command chain.

This is a utility function and is called from various codes.

nextSize()

Calculates new buffer size when it is enlarged.

This is a utility function and is called from various codes.

getCleanHostname()

Gets host name without domain qualification.

This is a utility function and is called from various codes.

prepareStdout()

Scans each shell command defined in cmdList_t structure. If stdout is not set, set it.

This is a utility function and is called from various codes.

makeConfigBackupCmd()

Builds cmd_t structure to perform configuration file backup.

This is for future use and is not used yet.

doConfigBackup()

Backs up configuration file.

This is called from failover_oneDatanode() in datanode_cmd.c to promote one datanode.

`dump_cmdList()`

Prints content of given `cmdList_t` structure. This is for debug and is called from `doCmdList()` in `do_shell.c`.

5.7.4.7 `gtm_cmd.c`

This module performs configuration, initialization and operation of gtm and gtm_proxy.

Functions defined in this module are as follows:

`prepare_initGtmMaster()`

Builds `cmd_t` structure to initialize gtm master. This is called from `init_gtm_master()` in `gtm_cmd.c` to initialize gtm master.

`init_gtm_master()`

Initializes gtm master. This is called from `init_all()` and `do_init_command()` in `do_command.c`.

`add_gtmSlave()`

Adds gtm slave. This is called from `do_add_command()` in `do_command.c`.

`remove_gtmSlave()`

Removes gtm slave. This is called from `do_remove_command()` in `do_command.c`.

`prepare_initGtmSlave()`

Builds `cmd_t` structure to initialize gtm slave. This is called from `init_gtm_slave()` in `gtm_cmd.c`.

`init_gtm_slave()`

Initializes gtm slave. This is called from the following codes:

Caller	File & Description
`init_all()`	`do_command.c`
`do_init()`	`do_command.c`
`add_gtmSlave()`	`gtm_cmd.c`

`prepare_startGtmMaster()`

Builds `cmd_t` structure to start gtm master. This is called from `start_gtm_master()` in `gtm_cmd.c`.

`start_gtm_master()`

Starts gtm master. This is called from the following codes:

Caller	File & Description
`init_all()`	`do_command.c`
`start_all()`	`do_command.c`
`add_gtmSlave()`	`gtm_cmd.c`
`init_all()`	`do_command.c`
`start_all()`	`do_command.c`
`do_start()`	`do_command.c`

`prepare_startGtmSlave()`

Builds `cmd_t` structure to start gtm slave. This is called from `start_gtm_slave()` in `gtm_cmd.c`.

`start_gtm_slave()`

Starts gtm slave. This is called from the following codes:

Caller	File & Description
`init_all()`	`do_command.c`
`start_all()`	`do_command.c`
`do_start()`	`do_command.c`
`add_gtmSlave()`	`gtm_cmd.c`

`prepare_stopGtmMaster()`

Builds `cmd_t` structure to stop gtm master. This is called from `stop_gtm_master()` in `gtm_cmd.c`.

`stop_gtm_master()`

Stops gtm master. This is called from the following codes:

Caller	File & Description
`stop_all()`	`do_command.c`
`do_stop_command()`	`do_command.c`

`prepare_stopGtmSlave()`

Builds `cmd_t` structure to stop gtm slave. This is called from `stop_gtm_slave` in `gtm_cmd.c`.

`stop_gtm_slave()`

Stops gtm slave. This is called from `stop_all()` and `do_stop_command()` in `do_command.c`.

`prepare_killGtmMaster()`

Builds `cmd_t` structure to kill gtm master. This is called from `kill_gtm_master` in `gtm_cmd.c`.

`kill_gtm_master()`

Kills gtm master process. This is called from `do_kill_command()` in `do_command.c`.

`prepare_killGtmSlave()`

Builds `cmd_t` structure to kill gtm master. This is called from `kill_gtm_master()` in `gtm_cmd.c`.

`kill_gtm_master()`

Kills gtm master. This is called from `do_kill_command()` in `do_command.c`.

`prepare_killGtmSlave()`

Builds `cmd_t` structure to kill gtm slave. This is called from `kill_gtm_slave()` in `gtm_cmd.c`.

`kill_gtm_slave()`

Kills gtm slave. This is called from `do_kill_command()` in `do_command.c`.

`failover_gtm()`

Promotes a gtm slave and run it as the new master. This is called from `do_failover_command()` in `do_command.c`.

`prepare_cleanGtmMaster()`

Builds `cmd_t` structure to cleanup gtm master resources. This is called from `clean_gtm_master()` in `gtm_cmd.c`.

`clean_gtm_master()`

Cleans up gtm master resources. This is called from `do_clean_command()` in `do_command.c`

`prepare_cleanGtmSlave()`

Build `cmd_t` structure to cleanup gtm slave resources. This is called from the following codes:

Caller	File & Description
`do_clean_command()`	`do_command.c`
`clean_gtm_slave()`	`gtm_cmd.c`

`clean_gtm_slave()`

Cleans up gtm slave resources.

Caller	File & Description
do_clean_command()	do_command.c
remove_gtmSlave	gtm_cmd.c

add_gtmProxy()

Adds gtm proxy. Please note that postgresql.conf of affected coordinator and datanode will be updated and they will be restarted.

This is called from do_add_command() in do_command.c.

remove_gtmProxy()

Removes gtm proxy. Please note that postgresql.conf of affected coordinator and datanode will be updated and they will be restarted.

This is called from do_remove_command() in do_command.c.

prepare_initGtmProxy()

Builds cmd_t structure to initialize gtm proxy. This is called from init_gtm_proxy() in gtm_cmd.c.

init_gtm_proxy()

Initializes gtm proxy. This is called from add_gtmProxy() and init_gtm_proxy_all() in gtm_cmd.c.

init_gtm_proxy_all()

Initializes all the gtm proxies defined in the configuration file. This is called from init_all() and do_init_command() in do_command.c.

prepare_startGtmProxy()

Builds cmd_t structure to start gtm proxy. This is called form start_gtm_proxy() in gtm_cmd.c.

start_gtm_proxy()

Starts gtm proxy. This is called from the following codes:

Caller	File & Description
do_start_command()	do_command.c
rdd_gtmProxy()	gtm_cmd.c
start_gtm_proxy_all()	gtm_cmd.c

start_gtm_proxy_all()

Starts all the gtm proxies. This is called from init_all(), start_all() and do_start_command() in do_command.c.

prepare_stopGtmProxy()

Builds cmd_t structure to stop gtm proxy. This is called from stop_gtm_proxy() in gtm_cmd.c.

stop_gtm_proxy()

Stops gtm proxies. This is called from the following codes:

Caller	File & Description
do_stop_command()	do_command.c
stop_gtm_proxy_all()	gtm_cmd.c

stop_gtm_proxy_all()

Stops all the gtm proxies. This is called from stop_all() and do_stop_command() in do_command.c.

prepare_killGtmProxy()

Builds cmd_t structure to kill a gtm proxy. This is called from kill_gtm_proxy() in gtm_cmd.c.

kill_gtm_proxy()

Kills gtm proxies. This is called from the following codes:

Caller	File & Description
do_kill_command()	do_command.c
kill_something()	do_command.c
kill_gtm_proxy_all()	gtm_cmd.c

kill_gtm_proxy_all()

Kills all the gtm proxies. It is for the future use and is not used now.

prepare_reconnectGtmProxy()

Builds cmd_t structure to perform reconnect command. This is called from reconnect_gtm_proxy() in gtm_cmd.c.

reconnect_gtm_proxy()

Reconnects gtm proxies to new gtm master. This is called from the following codes:

Caller	File & Description
do_reconnect_command()	do_command.c
reconnect_gtm_proxy_all()	gtm_cmd.c

prepare_cleanGtmProxy()

Builds cmd_t structure to clean up gtm proxy resources. This is called from clean_gtm_proxy() in gtm_cmd.c.

clean_gtm_proxy()

Cleans up gtm proxy resources. This is called from the following codes:

Caller	File & Description
do_clean_command()	do_command.c
remove_gtmProxy()	gtm_cmd.c
clean_gtm_proxy_all()	gtm_cmd.c

show_config_gtmMaster()

Shows gtm master configuration. This is called from the following codes:

Caller	File & Description
show_config_something()	do_command.c
show_configuration()	do_command.c
show_config_host()	do_command.c

show_config_gtmSlave()

Shows gtm slave configuration. This is called from the following codes:

Caller	File & Description
show_config_something()	do_command.c
show_configuration()	do_command.c
show_config_host()	do_command.c

show_config_gtmProxies()

Shows configuration of gtm proxies.. This is called from show_configuration() from do_command.c.

show_config_gtmProxy()

Shows configuration of a gtm proxy. This is called from the following codes:

Caller	File & Description
`show_config_something()`	`do_command.c`
`show_config_host()`	`do_command.c`
`show_config_gtmProxies()`	`gtm_cmd.c`

5.7.4.8 `gtm_util.c`

This module handles direct communication with gtm and gtm proxies internally.

Functions defined in this module are as follows:

`inputError()`

Internal input error handler. This is a utility function and called from various codes inside `gtm_util.c`.

`unregisterFromGtm()`

Handles `unregister` command to unregister specified node from GTM. This is called from `do_singleLine()` in `do_command.c`.

`connectGTM()`

Establish connection to gtm. This is a utility function and called from various codes inside `gtm_util.c`.

`process_unregister_command()`

Handles unregistration from GTM. This code uses gtm module from `src/gtm`.

This function is called from `unregisterFromGtm()` in `gtm_util.c` and also used inside macros `unregister_gtm_proxy()`, `unregister_coordinator()` and `unregister_detanode()` defined in `gtm_util.h`.

5.7.4.9 `make_signature`

This is a bash script which creates signature file to build following files:

- Signature file used to check if runtime bash script matches `pgxc_ctl` binary.
- Source code which contains runtime bash script to provide default configuration values and to read predefined configuration variable values.

5.7.4.10 mcxt.c

This is abstract memory handling module. Some Postgres-XC modules depend upon `palloc()`, `pfree()` and other postmaster-specific memory allocation, which should be replaced with bare malloc() and free() in `pgxc_ctl`. This modules handles such difference.

Implementation is straightforward and the detailed description of functions are not given here, except for major function interfaces.

It is for the future use and is not used at present.

5.7.4.11 monitor.c

This module monitors if specified component is running. Structure of the module is very straightforward and detailed description of all the functions will not be given, except for important ones.

Monitoring itself for gtm/gtm proxy and coordinator/datanode are done in different manners.

Monitoring gtm and gtm proxy is implemented in the function `do_gtm_ping()`, which establishes connection to specified gtm or gtm proxy by using `PQconnectGTM()` function provided at `src/gtm`.

Monitoring coordinator and datanode is done by the function `pingNode()` function in `utils.c`. In this function, `PQPing()` is used to check if the target coordinator or datanode is running.

5.7.4.12 pgxc_ctl.bash

This is the original bash script used to write current `pgxc_ctl`. The script is helpful to learn what steps are taken in `pgxc_ctl` and is here because of this reason.

5.7.4.13 pgxc_ctl.c

This is the main `pgxc_ctl` utility.

This handles bash command option when `pgxc_ctl` is invoked, reads configuration file, sets up working environment and handles each command line by line.

This section will give general steps how `pgxc_ctl` runs.

1. `main()` handles its own command line options. If help or version printing is specified, do them and quit.
2. Reads environment file from `/etc/pgxc_ctl` and `$HOME/`.`pgxc_ctl` using `setup_my_env()` in `pgxc_ctl.c`. This function reads environment parameters from these files such as `pgxc_ctl` home directory, prompt, verbosity, log directory, log file name, and configuration file name among others.
3. Starts logging with `sgtartLog()` in `pgxc_ctl.c`.

4. Installs bash script file which contains bash scripts to read final bash variables defining Postgres-XC configuration, as well as default value of the configuration. prepare_pgxc_ctl_bash() in pgxc_ctl.c handles this. All the scripts will be found in pgxc_ctl_bash.c.

5. Builds a path to the configuration file using build_configuration_path() in pgxc_ctl.c.

6. Reads configuration variable values using read_configuration() in pgxc_ctl.c. This function invokes the bash script installed in the step 4 with options to read the configuration file. The script will read all the predefined bash script variables to construct Postgres-XC cluster configuration. This way, DBAs can write any bash shortcuts for their needs to make configuration more comprehensible.

7. Checks the configuration. If there's no conflict, then pgxc_ctl begins to read a command line by line by do_command() in do_command.c.

pgxc_ctl accepts its command as pgxc_ctl command arguments. In this case, only these commands are handled and pgxc_ctl exists then.

5.7.4.14 pgxc_ctl_bash.c

This module holds configuration template. It is generated by make_signature.

5.7.4.15 pgxc_ctl_bash.c

This module contains default configuration values and bash script to read the configuration file. It is generated by make_signature.

5.7.4.16 pgxc_ctl_bash_2

This module is bash script which holds bash functions to read pgxc_ctl configuration file. This is read by make_signature and included into pgxc_ctl_bash.c.

5.7.4.17 pgxc_ctl_conf_part

This module is bash script which holds default configuration variables. This is read by make_signature and included into pgxc_ctl_bash.c.

5.7.4.18 pgxc_ctl_log.c

This is pgxc_ctl's logging module. Functions in this modules are simplified version of elog module in the postmaster and gtm.

Detailed description of this module will not be given here.

5.7.4.19 signature.h

This holds signature information generated by `make_signature` script used to check if internal bash script matches `pgxc_ctl` binary. It is for the future use and is not used at present.

5.7.4.20 utils.c

This module contains miscellaneous utility functions.

Most of them are wrapper function for general library. Non-wrapper function description will be given. Because they are general utility functions, their caller may no be given unless needed.

`appendFiles()`

Appends contents of the specified file path to specified file descriptor. It is used to collect more than one optional files.

`prepareLocalStdin()`

Collects contents of one or more than one file into a single one as `stdin` and returns its file descriptor.

`makeActualNodeList()`

Checks list of the node, suppress `NULL` value (as the string "`none`" or "`N/A`") and return the list with valid node list.

`gtmProxyIdx()`

Finds gtm proxy local to the given component.

`coordIdx()`

Finds internal index of the given coordinator.

`datanodeIdx()`

Finds internal index of the given datanode.

`getEffectiveGtmProxyIdxFromServerName()`

Finds gtm proxy available at given server.

`get_prog_pid()`

Finds the process ID of given component.

`pingNode()`

Monitors if specified coordinator or datanode is running. It uses `PQPing()` API of libpq.

`getChPidList()`

Gets a list of child processes of given PID at given host.

`getIpAddress()`

Finds IP address of the given host.

`myUsleep()`

Sleeps in microsecond.

5.7.4.21 variables.c

`pgxc_ctl` depends upon `bash` variable values read from the configuration file. To make source code more comprehensible and maintain clear relation to the original configuration file, most of `pgxc_ctl` modules depends upon variable system provided by this module.

`varnames.h` defines macros to translate C language symbols into real parameter values. See below for details.

`variables.h` gives some internal structure of the variable system, as well as convenient macros.

First of all, `pgxc_ctl`'s variable system does not distinguish arrays from scalars. Each variable is internally treated as an array.

If application (other code in `pgxc_ctl`) need to handle some variable as scalar, it just takes first member of the array. Shortcut macros are provided to handle this simply.

All the variables are represented as single list, as defined in `pgxc_ctl_var` structure in `variables.h`, as well as the head and tail defined as `var_head` and `var_tail` respectively.

To improve the performance, variables are organized into hash index. Hash bucket structure is defined as `pgxc_var_hash` structure in `variables.h`.

External functions and macros defined in `variable.c` and `variable.h` are as follows:

`init_var_hash()`

Initializes has table.

`add_var_hash()`

Adds new variable to the hash.

`new_var()`

Adds new variable.

`remove_var()`

Removes the variable from the list and hash table, then free it.

`add_val()`

Adds a value to the variable as a new array element.

`add_val_name()`

Adds a value to the variable with a given name.

`find_var()`

Finds variable structure with a given name.

`svar()`

Returns the value of the variable as a scalar.

`aval()`

Returns the value of the variable as an array.

`reset_value()`

Resets the value of the variable to NULL.

`assign_val()`

Copys the source variable values to the destination.

`reset_var()`

Similar to `reset_value()` but variable is specified by name, not structure address.

`reset_var_val()`

Resets specified variable value and assign specified value to it.

`confirm_var()`

Finds the address of specified variable. If not found, allocate it.

`pirnt_vars()`

Prints all the variables.

`print_var()`

Prints specified variable.

`log_var()`

Prints specified variable to the log.

`arraySizeName()`

Returns array size of the variable specified by name.

`arraySize()`

Returns array size of the variable specified by address.

`add_member()`

Appends the specified array value to specified variable array.

`clean_array()`

Frees specified array value.

`var_assign()`

Frees the destination and assign the source to the destination.

`listValue()`

Returns string representation of the variable value.

`ifExists()`

Returns if the variable exists and valid value is assigned to it.

`extendVar()`

Extends the size of the array of the variable and add specified value to it. Extended array element value is initialized with specified padding string.

`assign_arrayEl()`

Assigns a string scalar value to specified element of the array. If specified element index is larger than existing value, then the size will be extended and additional element is initialized with the padding string before the specified element value is set.

`doesExist()`

Tests if specified array element of the specified variable name exists.

`AddMember(a,b)`

This is a macro defined in `variables.h`. Appends specified array `a` to the variable specified by the name `b` and replace the value `a` with the new array.

`CleanArray(a)`

Cleans the array `a` and assign `NULL` to `a`.

`VAR(a)`

Returns address of the variable with the name `a`.

5.7.4.22 `varnames.h`

Defines variable symbol for the name. The symbol `VAR_`*name* is defined as the variable name "*name*."

5.8 Pgxc_clean module

`pgxc_clean` is Postgres-XC utility to cleanup transaction commit state inconsistency due to the crash of any coordinator or datanode.

`pgxc_clean` visits all two-phase commit (2PC, afterwords) transactions of all the nodes and check if there's any outstanding 2PC transactions, which are not prepared/committed/aborted, and see if the status is consistent. If there's any conflict, `pgxc_clean` cleans up the status. For 2PC transactions which need application intervention or cannot resolve conflict, `pgxc_clean` prints a message so that DBA can determine how to fix these conflicts.

Reference document of `pgxc_clean` will be found at `http://postgres-xc.sourceforge.net/docs/1_2_1pgxcclean.html`.

Source code is located at `contrib/pgxc_clean`.

5.8.1 Two-Phase Commit Transactions in Postgres-XC

Postgres-XC has two kinds of 2PC transactions as follows:

Implicit 2PC: Implicit 2PC transaction is the one where more than one nodes are updated. 2PC commit protocol is needed to maintain transaction integrity. When to commit, "`PREPARE TRANSACTION`" statement is issued to all the involved node and then "`COMMIT PREPARED`" statement is issued immediately. When to abort, no "`PREPARE TRANSACTION`" statement is needed.

ABORT statement is issued to all the target node instead. Implicit 2PC should not survive a session and if its status is "**PREPARED**", this transaction is intended to be committed.

Transaction id for implicit 2PC transaction is "__XC[0-9]+".

Explicit 2PC: Explicit 2PC transaction is the one where PREPARE TRANSACTION statement is issued by the application. When more than one node are involved in the transaction, PREPARE TRANSACTION, COMMIT PREPARED and ABORT command are propagated to all the involved node[5]. Explicit 2PC transaction survives a session and node restart, that is, even if 2PC transaction's status is "**PREPARED**", we cannot determine if it should be committed or aborted. Only application can tell this.

5.8.2 Transaction Commit Steps

In Postgres-XC, if a transaction is involved by more than one node, successful implicit 2PC transaction are handled as follows:

Handles SQL statement from application
↓
Receives COMMIT or end of transaction block
↓
Issues PREPARE TRANSACTION to all the involved nodes
↓
Issues COMMIT PREPARED to all the involved nodes

Successful explicit 2PC transaction are handled as follows:

Handles SQL statement from application
↓
Receives PREPARE TRANSACTION from application
↓
Propagates PREPARE TRANSACTION to all the involved nodes
↓
Receives COMMIT PREPARED and propagate it to involved nodes.

Please note that final COMMIT PREPARED is not included in explicit 2PC transaction handling. This has to be done when Postgres-XC receives COMMIT PREPARED statement.

[5] In implementation, Postgres-XC needs to maintain involved nodes in 2PC information, which has not been done yet. Postgres-XC needs extension so that pg_prepared_xacts includes a set of nodes involved

5.8.3 Possible Transaction Status Conflicts

Transaction status conflict may occur only with system failure, including node crash or hardware crash.

Here, we analyze possible crash/failure point and transaction status conflict among nodes.

In this section, we may use a term **root transaction**, which is the one taking care of connection form an application directly. Other piece of transaction created by the root transaction may be called **child transaction**.

5.8.3.1 Implicit 2PC

Before receiving `COMMIT` If root transaction fails, connections to all the child transactions fail. In this case, when transaction recovery is done, all the transactions are marks as "aborted" in CLOG and we don't need to do any more cleanup work.

While handling `PREPARE TRANSACTION` If any failure occurs in each `PREPARE TRANSACTION` handling, `ABORT` will be issued to all the node. If any node crashes and cannot receive `ABORT`, transaction will remain **prepared** in failed nodes while it is aborted in other nodes. This is one possible conflict visible when this node recovers and joins the cluster again.

While handling `COMMIT PREPARED` Because `PREPARE TRANSACTION` is successful in all the involved nodes, subsequent `COMMIT PREPARED` must be successful to if there's no system failure, such as crash storage failure.

After failed node is back, their transaction status is "prepared" while others are "committed".

5.8.3.2 Explicit 2PC

Before receiving `PREPARE TRANSACTION` Same as implicit 2PC before receiving `COMMIT`. All the transaction at all nodes fail and they are marked "aborted" in each local CLOG.

While propagating `PREPARE TRANSACTION` If some node crashes before `PREPARE TRANSACTION` is propagated, their status will eventually become "aborted", while others are "prepared".

While propagating `COMMIT PREPARED` If some of the involved node fail, their transaction status will remain "prepared", while others are "committed".

5.8.3.3 Possible status conflicts and their cleaning-up

Possible status will be summarized as follows:

- "Committed" at some node, "prepared" at others. In both implicit and explicit 2PC, it is obvious that this transaction is to be committed.

- "Aborted" at some node, "prepared" at others. In both implicit and explicit 2PC, it is obvious the transaction should be aborted.

- All "prepared". In implicit 2PC, it is obvious that it should be committed. In explicit 2PC, application should determine if this is to be aborted or committed.

`pgxc_clean` visits datanodes/coordinators and obtains any piece of transactions which remain "prepared", then test this transaction at other nodes to find the above conflicts and cleans them up.

To obtain all the prepared transaction information, `pgxc_clean` uses `EXECUTE DIRECT` statement to read `pg_prepared_xacts` system view.

To obtain status of prepared transaction at other nodes, it visits `pg_prepared_xacts` to obtain locally prepared transaction. It uses `pgxc_is_committed()` system function to obtain committed or aborted transaction status.

In cleaning-up, `pgxc_clean` need to issue `COMMIT PREPARED` or `ROLLBACK PREPARED` statement to target node using `EXECUTE DIRECT` statement. Usually `EXECUTE DIRECT` statement allow only read operation. To override this, `pgxc_clean` turns on `xc_maintenance_mode` connection parameter to ON. This is Postgres-XC-specific GUC parameter only DBA can turn it on to allow `EXECUTE DIRECT` to perform update operation as well. This is actually a GUC parameter which cannot turn on in `postgresql.conf`.

5.9 Pgxc_monitor module

`pgxc_monitor` is Postgres-XC utility to check if specified node is running.

Reference document will be found at `http://postgres-xc.sourceforge.net/docs/1_2_1/pgxcmonitor.html`.

Its source code is located at `contrib/pgxc_monitor`.

General description how to monitor each component is given below. Please note that `pgxc_monitor` does not include retry if it couldn't determine the target is running. Applications should do it.

5.9.1 Monitoring GTM and GTM Proxy

It is performed by connecting to specified GTM and GTM proxy using GTM client library function `PQconnectGTM()`. It determines that specified GTM or GTM Proxy is running if connection is established.

5.9.2 Monitoring Coordinator and Datanode

It is performed by issuing "`SELECT 1;`" SQL statement through `psql`.

5.10 Pgxc_ddl module

`pgxc_ddl` is Postgres-XC utility to propagate coordinator's catalog update to other nodes.

Reference document will be found at `http://postgres-xc.sourceforge.net/docs/1_2_1/pgxcddl.html` and the code, it is a `bash` script in fact, is found at `src/contrib/pgxc_ddl`.

`pgxc_ddl` was needed when Postgres-XC didn't propagate its DDL handling automatically, version 0.9 or earlier, and it is not needed at present releases later than 1.0.

Chapter 6

Configure Database

6.1 Changes in initdb

Essentially, we have two extension/modification to `initdb` utility:

- Setting up its own node name to `pgxc_node` catalog.
- Vacuum freeze every template and intial database: `template0`, `template1` and `postgres`.

Other extensions in the catalog, built-in functions and other embedded database objects specific to Postgres-XC are in catalog initialization and initdb does not need specific change for this.

The background of the need of complete vacuum freeze comes from dynamic node addition.

In original initdb, it comsumes some XIDs locally. Although initdb cannot run with GTM so far, it was not an issue if all the nodes are initialized at the same time. When GTM works for the first time, it can begin with a GXID larger than the last XID value consumed by initdb. `-x` option worked to deal with this situation.

When a new node should be added dynamically, current GXID could be of any value. It can be just before GXID wraps around! To enable new node (initialized with `initdb`) to be added to running cluster, it was essential to vacuum freeze the database completeley before joining Postgres-XC clusetr so that it can begin with any GXID value.

Following shows inddividual extension to `initdb`. Please note that `initdb` source file is essentially one. `findtimezone.c` and `po` directory are common to Postgres-XC which does not need any modification.

6.1.1 New option

`--nodename` option specifies its node name. `initdb` itself does not require if the new node should be a coordinator or a datanode. It is specified with `-Z` option of `pg_ctl` utility, although it is not a good idea to switch between coordinator and datanode while the node is already in use.

You will find this change in `main()` and `setup_config()` functions.

6.1.2 Vacuum freeze

Vacuum freeze feature is implemented as `vacuumfreeze()` function. This is done by running postgres binary against new initialized database cluster and issue `VACUUM FREEZE` command.

Please note that postgres binary need to run with local XID in this case because GTM might not have configured and running yet. For this purpose, `initdb` runs `postgres` binary against the new database with "`--single`" and "`--localxid`" options. Former option specifies `postgres` to run as a single user mode and the latter specifies to run with local XID, which is Postgres-XC-specific extension to `postgres` binary.

`vacuumfreeze()` function is called from `main()` function.

6.2 Extension of postgresql.conf

6.2.1 List of additional GUC parameters

All the GUC parameters specific to Postgres-XC will be found at `guc.c` module.

`enable_remotejoin` type: `boolean`; default: `true`

Enables join push-down to remote node. This should be turned on all the time unless you are debugging Postgres-XC planner or executor.

`enable_remotejoin` type: `boolean`; default: `true`

Enables fast query shipping. This should be turned on all the time unless you are debugging Postgres-XC planner or executor.

`enable_remotegroup` type: `boolean`; default: `true`

Enables `GROUP BY` pushdown. This shoudl be turned on all the time unless you are debugging Postgres-XC planner or executor.

`enable_remotesort` type: `boolean`; default: `true`

Enables `ORDER BY` pushdonw. This should be turned on all the time unless you are debugging Postgres-XC planner or executor.

`enable_remotelimit` type: `boolean`; default: `true`

Enables `LIMIT` clause pushdown. This should be turned on all the time unless you are debugging Postgres-XC planner or executor.

`gtm_backup_barrier` type: `boolean`; default: `false`

Controls if GTM backs up restart point for each `BARRIER`.

`persistent_datanode_connections` type: `boolean`; default: `false`

If on, the pooler will not release the acquired connec tion.

`enforce_two_phase_commit` type: `boolean`; default: `true`

This is mainly for the compatibility and consistency in the regression test. If set to `false`, implicit two phase commit will not be used.

`xc_maintenance_mode` type: `boolean`; default: `false`

When `true`, it enables `EXECUTE DIRECT` to perform write operation. This cannot be turned on in `postgresql.conf`. Only a superuser can tur this on using `SET` command. It is intended to be used to maintain cluster-wide data consistency. It is used in `pgxc_clean`.

`require_replicated_table_pkey` type: `boolean`; default: `true`

This congtrols replicated table update. If no primary key or other unique constraint is not available in replicated table update and needs ctid for update, this option make such statment to fail, when turned on. To maintain replicated table consisiten, you should not turn this off unless you fully understand its outcome and you intend to do so.

`min_pool_size` type: `integer`; default: `1`

Specifies minimum number of pooled connection to datanodes.

`max_pool_size` type: `integer`; default: `100`

Specifies maximum number of pooled connection to datanodes.

`pooler_port` type: `integer`; default: `6667`

Port number for pooler to listen.

`gtm_port` type: `integer`; default: `6666`

Port number of GTM or GTM proxy to connect to. If the node (coordinator or datanode) is connecting to GTM directly, specify GTM port number. If conneted through GTM proxy, specify GTM proxy port number.

`max_datanodes` type: `integer`; default: `16`

Maximum number of datanodes.

`max_coordinators` type: `integer`; default: `16`

Maximum number of coordinators.

`gtm_host` type: `string`; default: `none`

Host name of GTM or GTM proxy connecting.

`pgxc_node_name` type: `string`; default: `none`

GTM node name.

`remotetype` type: `enum`; default: ``‘‘application’’``

Not used at present.

6.2.2 Additional functions to handle GUC parameters

`check_pgxc_maintenance_mode()` function was added to disable `xc_maintenance_mode` GUC turned on at `postgresql.conf` file. This function checks if the parameter value was set to `true` in `postgresql.conf`.

Chapter 7

Database Maintenance

7.1 Vacuum

Vacuum has two major classes: VACUUM and Auto Vacuum (LAZY VACUUM). There is a significant difference of Vacuum behavior from other statment. Datanodes obtain GXID and the snapshot for Vacuum directly from GTM by themselves.

VACUUM is issed by DBA to collect garbages and analyze a database. Basically, VACUUM maintains the database just like PostgreSQL. But there's a bit difference to support Postgres-XC specific situation.

VACUUM checks that the XID doesn't go backward after last vacuum to protect the database. But the XID could go out-of-sync in Postgres-XC when the datanode switch to global XID from local XID or it has been idle for a long time (and other nodes have been worked hard.) To fix this case, Postgres-XC allows rewinding XID in standalone mode.

Auto Vacuum is invoked periodically specified by autovacuum_naptime GUC parameter.

In PostgreSQL, Auto Vacuum is also performed each 65k transactions, but this feature does not work well in Postgres-XC, because if the node does not involved in milestone transaction, he doesn't start Auto Vacuum.

As VACUUM and Auto Vacuum shares the code, changes to Auto Vacuum are almost same as VACUUM. But please not that Auto Vacuum inquires special GXID that is not listed in global snapshot. PostgreSQL excludes Auto Vacuum transaction too. To do so, Auto Vacuum inquires a GXID to a GTM using dedicated message TXN_BEGIN_GETGXID_AUTOVACUUM.

7.2 Changes in pg_dump

Postgres-XC has three extension/modification to pg_dump and pg_dumpall utility:

- Including node definitions. (pg_dumpall only)
- Including table distribution parameters into table definitions.
- Obtain sequence values not only from sequence relations but GTM.

With --dump-nodes option, pg_dump_all includes nodes' definitions to dumped data by scanning pg_catalog.pgxc_node and pg_catalog.pgxc_group.

Table distribution parameters mean DISTRIBUTE BY and TO NODE clauses. If a user want to include TO NODE clause into the dump file, --include-nodes option makes that clause.

In Postgres-XC, locally cached values in sequence relations might not be latest values because the sequence values could have been modified by other coordinators. To backup the latest sequence values, pg_dump calls pg_catalog.nextval.

7.3 Table Redistribution

When we make a change in a distribution rule or the distribution target of a table, Postgres-XC redistribute existing data in the table. To redistribute data, the coordinator collects all the data from relevant datanodes, and then redistributes the data according to the new rule.

The data redistribution is performed automatically by `ALTER TABLE` command. When `ALTER NODE` command detects the need of redistribution, it calls `BuildRedistribCommands()` to build redistribution state object. The redistribution state object includes SQL commands to redistribute, they are consist of `COPY TO`, `TRUNCATE` and `COPY FROM`. And `ALTER TABLE` calls `PGXCRedistribTable()` to perform redistruibution. This function called twice: before and after update the catalog. Please visit `redistrib.c` for the logic of the redistribution.

Chapter 8

Cluster Management

8.1 Cluster Node

8.1.1 Cluster management statement

To manage the cluster node, there are `CREATE NODE`, `ALTER NODE`, and `DROP NODE` statements.

Postgres-XC added the symtax to implement these cluster management statement. The grammar is defined in `src/backend/parser/gram.y`. Additional changes were made to support these statements. The changes are listed in Table 8.1.

Table 8.1: Changes to support node management statements

File	Function name	Description
`utility.c`	`standard_ProcessUtility()`	Changed to recognize `CreateNodeStmt`, `AlterNodeStmt` and `DropNodeStmt` type statements as utility statements and call appropriate functions.
`copyfunc.c`	`CopyObject()`	Added support for `CreateNodeStmt`, `AlterNodeStmt` and `DropNodeStmt`.
`equalfunc.c`	`equal()`	Added support for `CreateNodeStmt`, `AlterNodeStmt` and `DropNodeStmt`.
`utility.c`	`IsStmtAllowedInLockMode()`	Allowed to execute `CreateNodeStmt`, `AlterNodeStmt` and `DropNodeStmt` in lock mode.
`utility.c`	`CreateCommandTag()`	Added support for `CreateNodeStmt`, `AlterNodeStmt` and `DropNodeStmt`.

Changes are simple: add new case branches to `switch-case` block for node tags and write a code for them.

```
case T_CheckPointStmt:
    retval = (void *) makeNode(CheckPointStmt);
    break;
#ifdef PGXC
case T_BarrierStmt:
    retval = _copyBarrierStmt(from);
    break;
case T_AlterNodeStmt:
    retval = _copyAlterNodeStmt(from);
    break;
case T_CreateNodeStmt:
```

The implementation of cluster management statement functions is described later.

8.1.2 Node information catalog

Node management statements manipulates the system catalog `pgxc_node`. The definition of the system catalog is given below. You can find the column in the catalog corresponding to the options of node management statements.

```
                          Table "pg_catalog.pgxc_node"
      Column      |  Type   | Modifiers | Storage | Stats target | Description
------------------+---------+-----------+---------+--------------+-------------
 node_name        | name    | not null  | plain   |              |
 node_type        | "char"  | not null  | plain   |              |
 node_port        | integer | not null  | plain   |              |
 node_host        | name    | not null  | plain   |              |
 nodeis_primary   | boolean | not null  | plain   |              |
 nodeis_preferred | boolean | not null  | plain   |              |
 node_id          | integer | not null  | plain   |              |
Indexes:
    "pgxc_node_id_index" UNIQUE, btree (node_id), tablespace "pg_global"
    "pgxc_node_name_index" UNIQUE, btree (node_name), tablespace "pg_global"
    "pgxc_node_oid_index" UNIQUE, btree (oid), tablespace "pg_global"
Has OIDs: yes
Tablespace: "pg_global"
```

This system catalog is defined in `src/include/catalog/pgxc_node.h`, which is created by `initdb` through the BKI file. To get the node's attribute easily, utility functions listed in Table 8.2 are implemented in `src/backend/utils/cache/lsyscache.c`. For the more high level use, please see the following **Node Manager** subsection.

Table 8.2: Node attribute query functions

Function name	Description
`get_pgxc_nodeoid()`	Get the node OID by node name.
`get_pgxc_nodename()`	Get the node name by OID.
`get_pgxc_node_id()`	Get the node ID by OID.
`get_pgxc_nodetype()`	Get the node type by OID.
`get_pgxc_nodehost()`	Get the node host by OID.
`get_pgxc_nodeport()`	Get the node port by OID.
`is_pgxc_nodepreferred()`	Get whether the node is preferred by OID.
`is_pgxc_nodeprimary()`	Get whether the node is primary by OID.

8.1.3 Node manager

The node manager is implemented in `nodemgr.c`. This subsection describes the APIs of the node manager.

The node manager consists of two types functions as follows:

- Gets node information from the shared memory where the information in `pgxc_node` catalog are copied.
- Manipulates node information in the `pgxc_node` catalog.

Please note that information in the shared memory and the system catalog are not always synchronized. To update information in the shared memory, a program need to call specific API. The pooler uses the shared memory only and both information are used by the database/tablespace size calculator and the advisory lock.

`NodeTablesShmemInit()`

Creates or attaches the shared memory for the node table.

This function creates or attaches the shared memory for the node table. The shared memory for the node table has fixed size. It means that the maximum size of the memory is allocated when it is initialized.

This function is called in the initialization sequence of the process. It is needed both for `postmaster` and other various processes like backends etc.

This is called from the following codes:

Caller	File & Description
`CreateSharedMemoryAndSemaphores`	`ipci.c` Used in creating and attaching various shared memories.

`NodeTablesShmemSize()`

Calcurates the size of shared memory for the node table.

This function calcurates the maximum memory size required for the node talbe.

This is called from the following codes:

Caller	File & Description
`CreateSharedMemoryAndSemaphores`	`ipci.c` Used in estimating the size of the shared memory required in total.

`PgxcNodeListAndCount()`

Updates node definitions in the shared memory tables with the system catalog data.

This is called from the following codes:

Caller	File & Description
`InitMultinodeExecutor`	`pgxcnode.c` Called prior to calling `PgxcNodeGetOids()`
`PoolManagerCheckConnectionInfo`	`poolmgr.c` Called before request to check the pooled connection.
`PoolManagerReloadConnectionInfo`	`poolmgr.c` Called before request to reload the connection info.

`PgxcNodeGetOids()`

Builds a list of OIDs of coordinators and data nodes.

This function creates a list of Oids of Coordinators and Datanodes currently exist in the shared memory, as well as number of Coordinators and Datanodes.

This is called from the following codes (excluding call from pooler):

Caller	File & Description
`pgxc_tablespace_size`	`dbsize.c` Used in calcurating tablespace size
`pgxc_database_size`	`dbsize.c` Used in calcurating database size
`pgxc_advisory_lock`	`lockfuncs.c` Used in inquiring advisory locks

`PgxcNodeGetDefinition()`

Find node definition in the shared memory node table by oid.

This function is called from the pooler.

`PgxcNodeAlter()`

Alter a PGXC node.

This function is utility function which directly processes `ALTER NODE` statement as follows:

1. Opens the heap using `open_heap()`,
2. Obtains a copy of current tuple of the target node from the system cache using `SearchSysCacheCopy1()`,
3. Updates it using `heap_modify_tuple()` and `simple_heap_update()`,
4. Updates the index of the system catalog using `CatalogUpdateIndexes()`,
5. Closes the heap using `close_heap()`.

This is called from the following codes:

Caller	File & Description
`standard_ProcessUtility`	`utility.c` Used to process the `AlterNodeStmt` type statement

`PgxcNodeCreate()`

Adds a PGXC node.

This function is a utility function which directly handles `CREATE NODE` statement as follows:

1. Generates new node id,

2. Opens the heap using `open_heap()`,
3. Creates new tuple for the target node using `heap_from_tuple()`,
4. Insert it using `simple_heap_insert()`,
5. Updates index of the catalog using `CatalogUpdateIndexes()` and
6. Closes the heap using `close_heap()`.

This is called from the following codes:

Caller	File & Description
`standard_ProcessUtility`	`utility.c` Used to process the `CreateNodeStmt` type statement

`PgxcNodeRemove()`

Remove a PGXC node

This function is a utility function which directly processes `DROP NODE` statement as follows:

1. Opens the heap using `open_heap()`,
2. Obtains current tuple for the target node from the system cache using `SearchSysCache1()`,
3. Delete it using `simple_heap_delete()`,
4. Release the cached tuple using `ReleaseSysCache()()`
5. and Closes heap using `close_heap()`.

This is called from the following codes:

Caller	File & Description
`standard_ProcessUtility`	`utility.c` Used to process the `DropNodeStmt` type statement

8.2 Node Group

8.2.1 Node group management statement

To manage the cluster node group, there are `CREATE NODE GROUP` and `DROP NODE GROUP` statements.

Postgres-XC extends the grammar to implement these statement. The grammar is defined in `src/backend/parser/gram.y`. And some changes made to support these statements. The changes are listed in Table 8.3.

The implementation of the statement function is described later.

Table 8.3: Changes to support node group management statements

File	Function name	Description
utility.c	standard_ProcessUtility()	Changed to recognize the CreateGroupStmt and DropGroupStmt type statement as a utility function and call appropriate function.
copyfunc.c	CopyObject()	Added support for CreateGroupStmt and DropGroupStmt.
equalfunc.c	equal()	Added support for CreateGroupStmt and DropGroupStmt.
utility.c	CreateCommandTag()	Added support for CreateGroupStmt and DropGroupStmt.

8.2.2 Group information catalog

These statements manipulates the system catalog pgxc_group internally. The definition of the table follows. You can find the column in the catalog corresponding to the options of the group management statement.

```
                        Table "pg_catalog.pgxc_group"
    Column     |    Type    | Modifiers | Storage | Stats target | Description
---------------+-----------+-----------+---------+--------------+-------------
 group_name    | name      | not null  | plain   |              |
 group_members | oidvector | not null  | plain   |              |
Indexes:
    "pgxc_group_name_index" UNIQUE, btree (group_name), tablespace "pg_global"
    "pgxc_group_oid" UNIQUE, btree (oid), tablespace "pg_global"
Has OIDs: yes
Tablespace: "pg_global"
```

This catalog is defined in src/include/catalog/pgxc_group.h, which are created by initdb through the BKI file. To get the node's attribute easily, utility functions listed in Table 8.4 are implemented in rc/backend/utils/cache/lsyscache.c. For the more high level use, please see the following **Group Manager** subsection.

Table 8.4: Node group attribute query functions

Function name	Description
get_pgxc_groupoid()	Get the group OID by group name.
get_pgxc_groupmembers()	Get node OIDs of the group members by group OID.

8.2.3 Group manager

The group manager is implemented in groupmgr.c. This subsection describes APIs of the group manager.

The group manager has following function:

- Manipulate node information in the `pgxc_group` catalog.

`PgxcGroupCreate()`

Creates a PGXC node group.

This function is a utility function which handles `CREATE NODE GROUP` statement directly as follows:

1. Builds a OID list of new group member node and converts the list to oid vector with `buildoidvector()`,
2. Opens the heap with `open_heap()`,
3. Creates new tuple for the target group with `heap_from_tuple()`,
4. Insert it with `simple_heap_insert()`,
5. Updates index of the catalog with `CatalogUpdateIndexes()`,
6. and Closes heap with `close_heap()`.

This is called from the following codes:

Caller	File & Description
`standard_ProcessUtility`	`utility.c` Used to process the `CreateGroupStmt` type statement

`PgxcGroupRemove()`

Removes a PGXC node group.

This function is a utility function which handles `DROP NODEnGROUP` statement directly as follows:

1. Opens the heap with `open_heap()`,
2. Obtains current tuple for the target node from the system cache with `SearchSysCache1()`,
3. Delete it with `simple_heap_delete()`,
4. Release the cached tuple with `ReleaseSysCache()`
5. Closes heap with `close_heap()`.

This is called from the following codes:

Caller	File & Description
`standard_ProcessUtility`	`utility.c` Used to process the `DropGrouptmt` type statement

8.3 Table Distribution Attributes

8.3.1 Table distribution statement

Postgres-XC stores extended table information around distribution, it is given in the table definition like "CREATE TABLE .. DISTRIBUTED BY .. NODE ..". Extended statements are listed below.

- CREATE TABLE
- CREATE TABLE AS
- ALTER TABLE

To implement these extension, Postgres-XC extends the grammar. The grammar is defined in src/backend/parser/gram.y.

Table 8.5: Changes of statements manipulates distribution information

Statement	File name	Function name	Description
CREATE TABLE	tablecmds.c	DefineRelation()	Changed to create pgxc_class entry using parsed tree node.
CREATE TABLE AS	-	-	Same as CREATE TABLE
ALTER TABLE	tablecmd.c	AlterTableGetLockLevel()	Set lock level for AT_DistributeBy, AT_SubCluster, AT_AddNodeList and AT_DeleteNodeList to exclusive
	tablecmd.c	ATExecCmd()	Adds support for AT_DistributeBy, AT_SubCluster, AT_AddNodeList and AT_DeleteNodeList
		ATPrepCmd()	Adds support for AT_DistributeBy, AT_SubCluster, AT_AddNodeList and AT_DeleteNodeList
	tablecmd.c	ATController()	Changed to build redistribution commands and execute it.
DROP TABLE	dependency.c	doDeletion()	Added support for deletion of OCLASS_PGXC_CALSS object

8.3.2 Table distribution information catalog

These table distribution/redistribution statements manipulates the system catalog pgxc_class internally. The definition of the table follows. You can find the column in the catalog corresponding to the options of the extended statement.

```
                          Table "pg_catalog.pgxc_class"
     Column       |   Type    | Modifiers | Storage | Stats target | Description
------------------+-----------+-----------+---------+--------------+-------------
 pcrelid          | oid       | not null  | plain   |              |
 pclocatortype    | "char"    | not null  | plain   |              |
 pcattnum         | smallint  | not null  | plain   |              |
 pchashalgorithm  | smallint  | not null  | plain   |              |
 pchashbuckets    | smallint  | not null  | plain   |              |
 nodeoids         | oidvector | not null  | plain   |              |
Indexes:
    "pgxc_class_pcrelid_index" UNIQUE, btree (pcrelid)
Has OIDs: no
```

This catalog is defined in `src/include/catalog/pgxc_class.h`, which are created by `initdb` through the BKI file. To get the node's attribute easily, utility functions listed in Table 8.6 are implemented in `src/backend/utils/cache/lsyscache.c`. Other high level functions are described in next section.

Table 8.6: Extended class attribute query functions

Function name	Description
`get_pgxc_classnodes()`	Gets the target node OIDs by table OID.

8.3.3 High-level functions for distributed table

8.3.3.1 `pgxc_class.c`

This module supplies functions to manipulate `pgxc_class`.

`PgxcClassCreate()`

Creates a `pgxc_class` entry.

This function manipulates `pgxc_class` system catalog information.

This is called from the following codes:

Caller	File & Description
`AddRelationDistribution`	`heap.c` Used in adding to `pgxc_class` table

`PgxcClassAlter()`

Modifies a `pgxc_class` entry with given data.

This function manipulates `pgxc_class` system catalog information.

This is called from the following codes:

Caller	File & Description
`AtExecDistributeBy`	`tablecmds.c`
`AtExecSubCluster`	`tablecmds.c`
`AtExecAddNode`	`tablecmds.c`
`AtExecDeleteNode`	`tablecmds.c`

`RemovePgxcClass()`

Removes extended PGXC information.

This function manipulates `pgxc_class` system catalog information.

This is called from the following codes:

Caller	File & Description
`doDeletion`	`dependency.c` Used in removing `pgxc_class` table

Chapter 9

Database Object and DDL

9.1 DDL Propagation to Other Nodes

As mentioned in the architecture section 1.3.4 at page 11, Postgres-XC propagates DDL execution to other node except for node management statements: `CREATE NODE`, `ALTER NODE`, `DROP NODE`, `CREATE NODE GROUP`, and `DROP NODE GROUP`.

In PostgreSQL, functions handling DDL statements start with those in `backend/tcop/utility.c`. Parsed DDL statements are fist handled by functions `ProcessUtility()`. If no hook is defined, then it is passed to `standard_ProcessUtility()`. If target object supports event triggers, then they are passed to `ProcessUtilitySlow()`.

DDL propagation to other nodes is implemented in each DDL execution code.

The following list shows an example for handling `CREATE TABLESPACE` statement in `standard_ProcessUtility()`.

```
        case T_CreateTableSpaceStmt:
#ifdef PGXC
            if (IS_PGXC_COORDINATOR && !IsConnFromCoord())
#endif
            /* no event triggers for global objects */
            PreventTransactionChain(isTopLevel, "CREATE TABLESPACE");
            CreateTableSpace((CreateTableSpaceStmt *) parsetree);
#ifdef PGXC
            if (IS_PGXC_COORDINATOR && !IsConnFromCoord())
                ExecUtilityWithMessage(queryString, sentToRemote, false);
#endif
            break;
```

The first block of the directive "`#ifdefPGXC`" tests if the current node is coordinator and the DDL is not from another coordinator. If so, then this is the root and has to check transaction chain.

The second block of the directive "`#ifdefPGXC`" again tests if it the current node is a coordinator and the DDL is not from another coordinator. If so, this node has to take care of DDL propagation. Otherwise, it just concentrate on local handling.

DDL propagation is handled in the same manner for other DDSs as well.

9.2 Additional Error Handling

If all the DDL can be handled within a transaction block and can handle abort correctly in vanilla PostgreSQL, Postgres-XC can use implicit 2PC to handle errors in other nodes. Unfortunately, some DDL such as `CRETE TABLESPACE` cannot be issued inside transaction block. Postgres-XC has to propagate such DDL too and has to complete the statement in atomic way. In other words, Postgres-XC has to guarantee that the DDL cannot be successful only partly in some nodes. Postgres-XC needs separate feature to maintain cluster-wide data integrity in such case, without using 2PC.

To make this cleanup work, Postgres-XC defines internal structure `dbcleanup_info` of the type `abort_callback_type` inside `execRemote.c`.

`set_dbcleanup_callback()` function registers cleanup function and cleanup data specific to each DDL which run only outside a transaction block. If this is registered, then `AtEOXact_DBCleanup()` will invoke it at the end of a transaction.

9.3 Additional Functions to handle DDL

`utility.c` is an entry point of most of the DDL handlers. To propagate DDL to other nodes, Postgres-XC implements several utility functions in this module.

`IsStmtAllowedInLockedMode()`

Determines if a given statement can run within a transaction block. It is used to determine if dedicated error handling is needed.

`ExecUtilityWithMessage()`

This function performs a statement in a remote node within a transaction block. This handles error by attaching failed node name and by rethrowing it.

`ExecUtilityStmtOnNodes()`

This function executes a utility statement on nodes, including coordinators.

`ExecUtilityFindNodes()`

Determines the list of nodes to launch query on.

`ExecUtilityFindNodesRelkind()`

Determines which node a statement should be executed on the given relation.

`GetNodesForCommentUtility()`

Returns Object ID of object commented.

`GetNodesForRulesUtility()`

Gets the nodes to execute the given RULE related to a utility statement.

`DropStmtPreTreatment()`

Performs a pre-treatment of `DROP` statement on a remote coordinator.

9.4 Tablespace

This section describes Postgres-XC's tablespace implementation.

9.4.1 Creating Tablespace

To create a tablespace, we need to specify an absolute directory path. In Postgres-XC, the tablespace need to be propagated to all the nodes, hence the directory path too. To maintain this syntax, Postgres-XC uses all this absolute path in all the nodes.

This sounds reasonable if each node is configured in different server. There's no resource conflict. However, standard Postgres-XC configuration advises to install both coordinator and datanode at the same server and we need to resolve this conflict.

Simple rule introduced here is to qualify tablespace directory path with the node name which does not conflict. It is safe enough because node name has to be unique throughout Postgres-XC cluster.

The following shows additional code to do this in the function `CreateTableSpace()` in `tablespace.c`.

```
    /*
     * Check that location isn't too long. Remember that we're going to append
     * 'PG_XXX/<dboid>/<relid>.<nnn>'.  FYI, we never actually reference the
     * whole path, but mkdir() uses the first two parts.
     */
    if (strlen(location) + 1 + strlen(TABLESPACE_VERSION_DIRECTORY) + 1 +
#ifdef PGXC
        /*
         * In Postgres-XC, node name is added in the tablespace folder name to
         * insure unique names for nodes sharing the same server.
         * So real format is PG_XXX_<nodename>/<dboid>/<relid>.<nnn>''
         */
        strlen(PGXCNodeName) + 1 +
#endif
        OIDCHARS + 1 + OIDCHARS + 1 + OIDCHARS > MAXPGPATH)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_OBJECT_DEFINITION),
                 errmsg("tablespace location \"%s\" is too long",
                        location)));
```

Node name is added to each tablespace directory so that it does not conflict if coordinator and datanode are configured in the same server and even if more than one coordinator or datanode is configured in the same saver.

Before WAL records are written for this operation, `CreateTableSpace()` registers its cleanup function like:

```
#ifdef PGXC
    /*
     * Even if we have succeeded, the transaction can be aborted because of
     * failure on other nodes. So register for cleanup.
     */
    set_dbcleanup_callback(createtbspc_abort_callback,
                           &tablespaceoid, sizeof(tablespaceoid));
#endif
```

`createtbspc_abort_callback()` is implemented in `tablespace.c()` as well, where created subdirectory under the tablespace path is removed for cleanup.

9.4.2 Modifying Tablespace

`AlterTableSpaceOptions()` is the handler of `ALTER TABLESPACE` statement.

This can be performed within transaction block and there's no Postgres-XC-specific modification to this implementation.

9.4.3 Dropping Tablespace

`DROP TABLESPACE` cannot run within a transaction block.

`DropTableSpace()` is the handler of this statement and there's no Postgres-XC-specific modification [1].

9.5 Materialized View

Just like usual views, materialized view is created at coordinator level, not datanode level, and is replicated among all the coordinators. When materialized view is created, originating coordinator collects all the rows and replicate them.

When materialized view is refreshed, originating coordinator corrects all the rows, drops all the existing rows and then replicates new ones.

9.5.1 Creating Materialized View

Materialized view is created by `CREATE MATERIALIZED VIEW` statement. Internally, this statement is handled as a variant of `CREATE TABLE AS` statement and handled by `ExecCreateTableAs()` in `createas.c`.

The following is how this is handled in `utility.c`.

```
            case T_CreateTableAsStmt:
                ExecCreateTableAs((CreateTableAsStmt *) parsetree,
                                  queryString, params, completionTag);
#ifdef PGXC
                /* Send CREATE MATERIALIZED VIEW command to all coordinators. */
                Assert(((CreateTableAsStmt *) parsetree)->relkind == OBJECT_MATVIEW);
                if (!((CreateTableAsStmt *) parsetree)->into->skipData && !
                     IsConnFromCoord())
                    pgxc_send_matview_data(((CreateTableAsStmt *) parsetree)->into->rel,
                                           queryString);
                else
                    ExecUtilityStmtOnNodes(queryString, NULL, sentToRemote, false,
                         EXEC_ON_COORDS, false);

#endif /* PGXC */
```

[1] The reporter is not sure if this implementation is reasonable. If `DROP TABLESPACE` fails in any of the nodes while propagating, the operation has to be cleaned up, if vanilla PostgreSQL is doing so. It may need more in-depth analysis of DROP TABLE failure handling in vanilla PostgreSQL.

```
                break;
```

Please note that Postgres-XC does not support `CREATE TABLE AS` statement and the above code is just for `CREATE MATERIALIZED VIEW` statement at present.

Piece of the code at the parser (`gram.y`) is as follows:

```
/*****************************************************************************
 *
 *      QUERY :
 *              CREATE MATERIALIZED VIEW relname AS SelectStmt
 *
 *****************************************************************************/

CreateMatViewStmt:
        CREATE OptNoLog MATERIALIZED VIEW create_mv_target AS SelectStmt opt_with_data
                {
                    CreateTableAsStmt *ctas = makeNode(CreateTableAsStmt);
                    ctas->query = $7;
                    ctas->into = $5;
                    ctas->relkind = OBJECT_MATVIEW;
                    ctas->is_select_into = false;
                    /* cram additional flags into the IntoClause */
                    $5->rel->relpersistence = $2;
                    $5->skipData = !($8);
                    $$ = (Node *) ctas;
                }
        ;
```

You will see that parse tree for `CREATE MATERIALIZED VIEW` statement is the same as `CREATE TABLE AS` statement.

`CREATE TABLE AS` statement is blocked at present at `gram.y`. Therefore, `CreateTableAsStmt` node is used only for `CREATE MATERIALIZED VIEW` at present.

9.5.2 Refreshing Materialized View

Contents of materialized views are refreshed by `REFRESH MATERIALIZED VIEW` statement. In Postgres-XC, materialized view refreshment causes all the old data are replaced with all the present data.

This is handled by PostgreSQL backend function `ExecRefreshMatView()`. Its code is almost the same as vanilla PostgreSQL. Only one difference is if it is from another coordinator, that is, if new row data comes from originating coordinator, the data is handled using `COPY` protocol, not by running queries.

Code snip in `ExecRefreshMatView()` is as follows:

```
#ifdef PGXC
    /*
     * If the REFRESH command was received from other coordinator, it will also send
     * the data to be filled in the materialized view, using COPY protocol.
    */
    if (IsConnFromCoord())
    {
        Assert(IS_PGXC_COORDINATOR);
        pgxc_fill_matview_by_copy(dest, stmt->skipData, 0, NULL);
    }
    else
#endif /* PGXC */
```

At the originating coordinator, `REFRESH MATERIALIZED VIEW` statement is handled locally first, and then the rows are propagated to other coordinators by using `pgxc_send_matview_data()` function.

Implementation of two Postgres-XC-specific functions is as follows:

`pgxc_send_matview_data()`

This function is implemented in `matview.c`. It opens specified materialized view, collect all the rows and send them to other coordinators using `COPY` command protocol.

`pgxc_fill_matview_by_copy()`

This function is implemented in `matview.c`. It receives table rows sent by `pgxc_send_matview_data()` and stores them in the target materialized view.

9.5.3 Dropping Materialized View

It is handled by `ExecDropStmt()` function in `utility.c`. Additions in Postgres-XC is as follows:

- Before removing local materialized view, Postgres-XC checks objects to be dropped. In materialized view, we don't have this yet.
- After materialized view was removed locally, and if it is done in originating coordinator, then the DDL is propagated to other coordinators using `ExecUtilityStmtOnNodes()`, implemented in `utility.c`.

9.6 Automatic Updatable View

An issue to support automatic updatable views is determining if a statement is updating distribution key, which is not allowed in Postgres-XC.

Code changed a bit to handle this in view update.

When a statement is rewritten which updates an updatable view, the result may include all the columns including distribution column, which is not updating anyway.

The change determines this more strictly to allow such case.

9.7 Trigger

9.7.1 Trigger Syntax

Trigger syntax is defined in `gram.y`. There is no Postgres-XC-specific change in trngger syntax.

9.7.2 Creating Trigger

`CREATE TRIGGER` statement is parsed into `CreateTrigStmt` structure and passed to `ProcessUtilitySlow()` function in `utility.c`. Here, after local handling has been done, the statement is propagated to other nodes where the base relation is defined.

`commands/trigger.c` implements most of trigger DDL handler. `CreateTrigger()` is the main handler of `CREATE TRIGGER` statement. They have no Postgres-XC-specific changes.

9.7.3 Changing Trigger definition

It is handled by `ExecRenameStmt()` in `alter.c`. Before this, the statement is propagated to other nodes where the base relation is defined [2].

In the case of `ALTER TRIGGER`, it is then passed to `remanetrig()` in `trigger.c`.

They have no Postgres-XC-specific change.

9.7.4 Dropping Trigger

This statement is handled by `ExecDropStmt()` in `utility.c` and then passed to `RemoveObjects()` in `dropcmds.c` before it is propagated to other nodes.

`RemoveObjects()` does not have Postgres-XC-specific changes.

9.7.5 Firing Trigger

Most of the changes needed to support triggers are in firing triggers, implemented in `trigger.c`.

This section describes Postgres-XC-specific utility functions in this module and then describes changes to existing trigger firing functions.

`pgxc_should_exec_triggers()`

Determines if all of the triggers for the relation should be executed here, on this node. On a coordinator, returns true if there is at least one non-shippable trigger for the relation that matches the given event, level and timing. (or for any local-only table for that matter), returns false if all of the matching triggers are shippable.

PG behaviour is such that the triggers for the same table should be executed in alphabetical order. This make it essential to execute all the triggers on the same node, be it coordinator or datanode. So the idea used here is: if all matching triggers are shippable, they should be executed on local tables (i.e. on datanodes). Even if there is at least one single trigger that is not shippable, all the triggers should be fired on remote tables (i.e. on the coordinator). This ensures that either all the triggers are executed on coordinator, or all are executed on datanodes.

[2] We need to check if `ALTER TABLE ... ADD NODE` handle this correctly.

`pgxc_is_trigger_firable()`

This function is defined only to handle the special case if the trigger is an internal trigger. Once we support global constraints, we should not handle this as a special case: global constraint triggers would be executed just like normal triggers. Internal triggers are internally created triggers for constraints such as foreign key or unique constraints. Currently we always execute an internal trigger on datanodes, assuming that the constraint trigger function is always shippable to datanodes. We can safely assume so because we disallow constraint creation for scenarios where the constraint needs access to records on other nodes.

`pgxc_is_internal_trig_firable()`

This function determines if a given internal trigger is firable at this node.

`pgxc_get_trigger_tuple()`

Obtains tuple of the trigger target.

`pgxc_check_distcol_update()`

Compares the distribution column values given to the function and error out if they are different. This is called to make sure triggers have not updated the distribution column.

9.7.5.1 Other Trigger-firing Functions

Other trigger-firint functions are modofied to determine if the trigger should be fired in this node. Fire it if yes.

9.8 Event Trigger

There are no Postgres-XC-specific changes in event trigger firing.

Changes a made to propagate all the DDLs to handle event trigger.

The changes are similar to other DDLs.

9.9 Temporary Objects

The change has been made to allow temporary object usage in explicit 2PC transactions.

The background that temporary object is not allowed in 2PC is that **PREPARED** transaction survives the session, while temporary objects do not.

In contrary, implicit 2PCs do not survive the session. Even with crashes, `pgxc_clean` cleans up implicit 2PC transactions so that they do not survive. It is safe to allow temporary objects to be used in implicit 2PCs.

Changes are done in `CommitTransaction()` in `xact.c`.

The patch is as follows:

```
--- a/src/backend/access/transam/xact.c
+++ b/src/backend/access/transam/xact.c
@@ -2117,6 +2117,9 @@ CommitTransaction(void)
 {
    TransactionState s = CurrentTransactionState;
    TransactionId latestXid;
+#ifdef PGXC
+       bool            isImplicit = !(s->blockState == TBLOCK_PREPARE);
+#endif

        ShowTransactionState("CommitTransaction");

@@ -2161,7 +2164,7 @@ CommitTransaction(void)
                */
                if (IsOnCommitActions() || ExecIsTempObjectIncluded())
                {
-                       if (!EnforceTwoPhaseCommit)
+                       if (!EnforceTwoPhaseCommit || isImplicit)
                            ExecSetTempObjectIncluded();
                        else
                            ereport(ERROR,
@@ -2655,7 +2658,11 @@ PrepareTransaction(void)
            * cases, such as a temp table created and dropped all within the
            * transaction.  That seems to require much more bookkeeping though.
            */
+#ifdef PGXC
+       if (MyXactAccessedTempRel && !isImplicit)
+#else
        if (MyXactAccessedTempRel)
+#endif
            ereport(ERROR,
                    (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                     errmsg("cannot PREPARE a transaction that has operated on temporary
                         tables")));
```

Chapter 10

Transaction Management

10.1 Cluster-wide MVCC

Postgres-XC's transaction management is compliant with ACID and ensures atomic visibility for multi-node read transactions among the cluster. This feature is achieved by implementing cluster-wide MVCC. Postgres-XC basically uses PostgreSQL's MVCC mechanism implemented in `src/backend/utils/time/tqual.c`: visibility is checked by `xmin`, `xmax`, **CLOG** and transaction snapshot (sometimes cmin and cmax are included.) Postgres-XC just extends PostgreSQL's mechanism to assign the transaction ID and to feed snapshot to global. The external component which feeds global transaction information is called Global Transaction Manager (GTM), which clusetr-wide MVCC depends upon. It was implemented separately from the core of coordinator and datanode. The source code will be found in `src/gtm/main` as described at section 5.5, page 88.

When a user issues a DML statement to a coordinator, the coordinator obtains a global transaction ID (GXID) and a global transaction snapshot from GTM and send it to datanodes. Datanodes manipulate their database using GXID and snapshot from the coordinator. In such manner, datanodes share the same transaction context and a transaction can maintain atomic and uniform visibility when it runs in more than one coordinators and datanodes. At the end of transaction, if more than one nodes are involved in the updates, the coordinator commits the transaction using 2PC protocol implicitly. By keeping track with global transaction status, coordinator reports global transaction status to GTM.

Visibility check sometimes uses transaction-local command ID generated by coordinators. It is also sent to datanodes before each command is shipped. If the command ID is advanced locally in some of involved datanodes, they notify the change to the coordinator.

The detailed modifications to the transaction mechanism is described in section 10.2, page 202.

Postgres-XC supports all transaction isolation levels implemented in PostgreSQL. If the transaction isolation level is `REPEATABLE READ`, one snapshot will be obtained and used throughout the transaction. If the isolation mode is `READ COMMITTED`, the coordinator obtains fresh snapshot for each statement. Please note that this snapshot is used for more than one statements if one statment is devided into multiple ones by the planner. [1].

Some small changes were needed to make this global MCXX work with existing PostgreSQL code. For example, CLOG expanding algorithm needed some modification. In a globalized transaction, some nodes may not be involved and GXID of such transaction is missing in such nodes. CLOG needs an extention to hand this missing GXID to make CLOG expansion works correctly. It is implemented at `src/backend/access/transam/clog.c:ExpandCLOG()`.

10.2 Global Transaction Management

Coordinators communicate with GTM in the following cases:

[1] I thought Postgres-XC doesn't support `REPEATABLE READ`. Doesn't it? – Saito –
– Koichi –
As design, it does. Sometime after 1.0 was released, new bug was introduced to disable `REPEATABLE READ`. It is not a design issue but a bug. In terms of `SERIALIZABLE`, predicate lock will work corretly with distributed tables. In the case of replicated tables, I believe each read in different datanode will leave their predicate lock. Conflicting writes will be detected in some of the datanodes which ends up with transaction failure. If it is true, Postgres-XC can support `SERIALIZABLE` isolation level too.

- When they require new transaction ID,
- When they need new transaction snapshot,
- When they commit or abort transactions.

Datanodes communicate with GTM when it execute vacuum.

The messages used for the global transaction management between GTM and other nodes (coordinator and datanode) are listed in Table 10.1. This list shows the messages actually implemented both in GTM and coordinator/datanode backends and does not include ones implemented only in GTM but not really used.

Table 10.1: Transaction Control Messages

Message name	Description
`TXN_BEGIN_GETGXID`	Start a new transaction and get GXID.
`TXN_START_PREPARED`	Begins to prepare a transaction for commit.
`TXN_COMMIT`	Commit a running or prepared transaction.
`TXN_COMMIT_PREPARED`	Commit a prepared transaction.
`TXN_PREPARE`	Finish preparing a transaction.
`TXN_ROLLBACK`	Rollback a transaction.
`TXN_GET_GID_DATA`	Get info associated with a GID, and get a GXID.
`SNAPSHOT_GET`	Get a global snapshot
`TXN_BEGIN_GETGXID_AUTOVACUUM`	Start a new transaction and get GXID for autovacuum.

Actual protocol used by GTM client is implemented in `src/gtm/client/gtm_client.c`. They are too primitive to be called from coordinator/datanode backend. Utility functions as shortcut to these primitive implementation was implemented in `src/backend/access/transam/gtm.c`. They are shown in Table 10.2 [2].

GTM utility functions are categorized into four groups as shown in Figure 10.2.

First group consists of `IsGTMConnected()`, `InitGTM()` and `CloseGTM()`. They handle connection to the GTM. These functions are called from other utility function in `gtm.c` internally. `InitGTM()` makes connection to a GTM and stores the connection information to a process local memory. The utility functions internally calls `InitGTM()`, and uses the stored connection. `CloseGTM()` resets the stored connection and information.

Second group consists of `BeginTranGTM()` and `BeginTranAutovacuumGTM()`. They inquires new global transaction ID to GTM. These functions are called from functions in `varsup.c`, which are responsible for OID & XID variables support. The functions in `varsup.c` now use these utility functions instead of original local XID feed mechanism.

Third group consists of `CommitTranGTM()`, `RollbackTranGTM()`, `StartPreparedTranGTM()`, `PrepareTranGTM()` and `CommitPreparedTranGTM()`. They are used to control transaction and are called from functions in `xact.c` and `execRemote.c`. Functions in `xact.c` are modified to use these utility functions to report transaction status to GTM. Changes will be described later.

[2] Is there need to mention about maintenance mode? – Saito –
– Koichi – Maintenance mode is not directly related to this level.

Table 10.2: GTM Utility Functions for coordinator/datanode backends

Function name	Description
`IsGTMConnected()`	Returns whether this backend has connection connected to GTM or not.
`InitGTM()`	Initializes and establishes connection to GTM for this backend.
`CloseGTM()`	Closes connection to GTM.
`BeginTranGTM()`	Inquires new transaction ID to GTM.
`BeginTranAutovacuumGTM()`	Inquires new transaction ID for autovacuum to GTM.
`CommitTranGTM()`	Notifies commit of a transaction to GTM.
`RollbackTranGTM()`	Notifies rollback of a transaction to GTM.
`StartPreparedTranGTM()`	Notifies starting preparation of a transaction to GTM.
`PrepareTranGTM()`	Notifies finished preparation of a transaction to GTM.
`CommitPreparedTranGTM()`	Notifies commit of a prepared transaction to GTM.
`GetGIDDataGTM()`	Gets info associated with a GID, and get a GXID from GTM.
`GetSnapshotGTM()`	Obtains a transaction snapshot from GTM.

Fourth group consists of `GetGIDDataGTM()` and `GetSnapshotGTM()`. They are used to obtain transaction information from GTM. `GetGIDDataGTM()` is called from `execRemote.c` module to obtain GXID from GID. `GetSnapshotGTM()` is called from `procarray.c` module to obtain the snapshot for the transaction from GTM. In this case, they do not scan the backend process array. Please note that a part of obtained information is stored to global variable and used by functions in `procarray.c` module. For example, `GetOldestXmin()` uses variable `RecentGlobalXmin` which is saved in a snapshot inquiring process.

Related to `procarray.c` above, Postgres-XC whips `KnownAssignedXids`*XXXX*`()` functions to disable hot standby feature. Because hot standby needs to provide consistent database views for all the datanode, which is not available yet. They must be different delay in playing back WAL record at slaves and PostgreSQL does not provide any infrastructure to synchronize the playback point. This makes it extremely challanging to provide consistent view to all the slave nodes, which is necessary for Postgres-XC's read transactions to slaves. Moreover, in the slave, current `KnownAssignedXids` ignores latter half of `XLOG_XACT_ASSIGNMENT` wal record and registers all the possible XIDs found at the first half of the wal record. Some of them can be missing and such missing Xids remain in the buffer, causing buffer overflow and the slave crash. It will need various change in the code, while the hot standby does not work correctly.

Functions in `xact.c` are modified to use GXID and global snapshot and to handle Postgres-XC-specific issues. They are listed in Table 10.3. These functions are a part of fundamental transaction management functions called from various functions in the executor, the optimizer, the aggregation functions among others.

Coordinators are required to hold status of both their local transactions and remote transactions of remote nodes involved. A global transaction consists of more than one transactions running at differnt nodes. They are single transaction but it is a collection of local transactions from the point of each node-local view. The structure `RemoteXactState` was introduced to keep track of this global transaction status. The structure is used internally in `src/backend/pgxc/Pool/execRemote.c` which implements the internal communication among

Table 10.3: Modified Transaction Management Functions

Function name	Description
`StartTransaction()`	Turns serializable isolation level into repeatable-reads which is same as pre 9.1 serializable isolation level. Postgres-XC doesn't support 9.1 serializable transactions.
`CommitTransaction()`	If data in the coordinator is involved to the transaction, call `PrepareTransaction()` to prepare the local transaction. After the transaction is prepared, the coordinator begins new transaction with the same GXID as the prepared transaction to continue the commit sequence. Then it calls `PreCommit_Remote()` to propagate commit to nodes with 2PC manner. To handle this prepared transaction locally, it calls `FinishPreparedTransaction()`. Next, it calls `CallGTMCallbacks()` to notify the global transaction is being committed to callback functions, for example, global sequence module will be called back. These callback functions are managed by functions listed at Table 10.4. After this, it cleanups various informations including callback information about GTM, command ID information, among others. Finally, `AtEOXact_GlobalTxn()` request GTM to commit the transaction, and `AtEOXact_Remote()` cleans up last information.
`PrepareTransaction()`	The coordinator calls `PrePrepare_Remote()` to propagate prepare to nodes. Next, it calls `CallGTMCallbacks()` to notify the global transaction is being prepared. Then it cleans up various informations. Finally, `AtEOXact_GlobalTxn()` request GTM to prepare the transaction.
`AbortTransaction()`	The coordinator calls `PreAbort_Remote()` to abort prepared transactions at remote nodes. It calls `FinishPreparedTransaction()` to handle this prepared transaction locally. Next, it calls `CallGTMCallbacks()` to notify the global transaction is being aborted. Then it cleans up various informations. Finally, `AtEOXact_GlobalTxn()` requests GTM to abort the transaction and `AtEOXact_Remote()` cleans up the last information.

Table 10.4: GTM Event Callback Management Functions

Function name	Description
`RegisterGTMCallback()`	Registers or unregisters callback functions for GTM at transaction start or stop. These operations are more or less the transaction callbacks but we need to perform them before `HOLD_INTERRUPTS` as it is a part of transaction management and is not included in xact cleaning. The callback is called when the transaction finishes and could be initialized by events related to GTM that need to be taken care of at the end of a transaction block.
`UnregisterGTMCallback()`	`UnregisterGTMCallback` removes specified functions from the set of callback functions.
`CallGTMCallbacks()`	`CallGTMCallbacks` calls all registered callback functions to notify the event.

nodes, including coordinators and datanodes.

As described in section 10.1 in page 202, a coordinator needs to share the snapshot. It is required to maintain cluster-wide MVCC. To share the snapshot among nodes, the coordinator sends it to nodes using same libpq connection before it sends. Coordinator-side protocol is implemented at `pgxcnode.c:pgxc_node_send_snapshot()`. Datanode-processes it at `postgres.c:PostgresMain()`. `pgxc_node_send_snapshot` is basically called from the common function. `execRemote.c:pgxc_start_command_on_connection()` or common function. `execRemote.c:ExecRemoteUtility()`.

Command ID is also shred among the backends running the same transaction. It's bidirectional. Protocol messages from a coordinator to a datanode is implemented at `pgxcnode.c: pgxc_node_send_cmd_id()` and the datanode handling of them is implemented at `postgres.c: PostgresMain`. `pgxcnode.c:pgxc_node_send_cmd_id()` is called from the common function `execRemtoe.c:pgxc_start_command_on_connection()`. If the command ID increments in any of the datanodes involved, it is reported back to the coordinator. It is implemented in `xact.c: ReportCommandIdChange()`. The coordinator handles it at `pgxc_node.c:handle_response()`. In addition, nodes have to clear their command ID at the atart and end of the transaction. The utility functions listed in Table 10.5 are placed in `xact.c`.

The GTM also supplies timestamp with new global transaction ID. It is saved into variable `GTMxactStartTimestamp` and time difference is saved into variable `GTMdeltaTimestamp`. To use these timestamp, the functions listed in Table 10.6 are modified.

Now let's take a look at internal data of the GTM. The transaction information structure is shown below.

```
typedef struct GTM_TransactionInfo
{
    GTM_TransactionHandle   gti_handle;
    GTM_ThreadID            gti_thread_id;

    bool                    gti_in_use;
    GlobalTransactionId     gti_gxid;
    GTM_TransactionStates   gti_state;
    char                    *gti_coordname;
    GlobalTransactionId     gti_xmin;
    GTM_IsolationLevel      gti_isolevel;
```

Table 10.5: Command ID management functions

Function name	Description
`SaveReceivedCommandId()`	Saves a received command ID from another node for future use.
`SetReceivedCommandId()`	Sets the command ID received from other nodes.
`GetReceivedCommandId()`	Gets the command ID received from other nodes.
`ReportCommandIdChange()`	Reports a change in current command ID at remote node to the Coordinator. This is required because a remote node can increment command ID in case of triggers or constraints.
`IsSendCommandId()`	Gets status of command ID sending. If set at true, command ID needs to be propagated to other nodes.
`SetSendCommandId()`	Sets status of command ID sending. If set at true, command ID needs to be propagated to other nodes.

Table 10.6: Modified functions to handle global timestamp

Function name
`AssignTransactionId()`
`GetCurrentCommandId()`
`GetCurrentTransactionStartTimestamp()`
`GetCurrentStatementStartTimestamp()`
`GetCurrentTransactionStopTimestamp()`
`RecordTransactionCommit()`
`RecordTransactionAbort()`
`StartTransaction()`

```
    bool                    gti_readonly;
    GTMProxy_ConnID         gti_backend_id;
    char                    *nodestring; /* List of nodes prepared */
    char                    *gti_gid;

    GTM_SnapshotData        gti_current_snapshot;
    bool                    gti_snapshot_set;

    GTM_RWLock              gti_lock;
    bool                    gti_vacuum;
} GTM_TransactionInfo;

typedef struct GTM_SnapshotData
{
    GlobalTransactionId     sn_xmin;
    GlobalTransactionId     sn_xmax;
    GlobalTransactionId     sn_recent_global_xmin;
    uint32                  sn_xcnt;
    GlobalTransactionId     *sn_xip;
} GTM_SnapshotData;
```

And PostgreSQL's snapshot data structure is here.

```
typedef struct SnapshotData
{
    SnapshotSatisfiesFunc satisfies;    /* tuple test function */

    TransactionId xmin;         /* all XID < xmin are visible to me */
    TransactionId xmax;         /* all XID >= xmax are invisible to me */
    TransactionId *xip;         /* array of xact IDs in progress */
    uint32      xcnt;           /* # of xact ids in xip[] */
#ifdef PGXC  /* PGXC_COORD */
    uint32      max_xcnt;       /* Max # of xact in xip[] */
#endif
    /* note: all ids in xip[] satisfy xmin <= xip[i] < xmax */
    int32       subxcnt;        /* # of xact ids in subxip[] */
    TransactionId *subxip;      /* array of subxact IDs in progress */
    bool        suboverflowed;  /* has the subxip array overflowed? */
    bool        takenDuringRecovery;    /* recovery-shaped snapshot? */
    bool        copied;         /* false if it's a static snapshot */

    CommandId   curcid;         /* in my xact, CID < curcid are visible */
    uint32      active_count;   /* refcount on ActiveSnapshot stack */
    uint32      regd_count;     /* refcount on RegisteredSnapshotList */
} SnapshotData;
```

You will find `GTM_SnapshotData` and `SnapshotData` very similar and GTM manages same data outside the coordinators and datanodes. But GTM doesn't have subtransaction and command ID data. GTM doesn't have subtransaction data because it has not been supported yet. GTM doesn't need to have command ID data because it is local to the originating coordinator which started the transaction. Command ID can be handled locally in the originating coordinator without GTM's help. If it is incremented locally at involved datanodes or other coordinatord, it is notified back to the coordinator for later use.

10.3 Solution to the SPOF Problem

The GTM can be the single point of failure in the cluster. Because beginning of any DML and DDL and DCL operation requires the new transaction ID. To avoid this, Postgres-XC provides GTM slave so that it can promote to the master maitaining all the current global transaction and sequence status.

When the GTM standby starts, it connect to the master and gets all the current data on transactions and sequences. Then the slave sends request to change his attribute to standby, disconnects the original connection to GTM master and waits the connection from GTM master. After the master reestablishes the connection to the slave, it simply propagates the message received from other GTM clients to the slave with the changed message type for backup. The messages used between master and slave are listed in Table 10.7. Please note that the table includes not only transaction management message but also sequence management message. The list includes all the messages listed in Table 10.1. Please also note that this list includes non-transactional messages. The slave processes these messages in the same manner as the master. The difference is the backup message does not require any response. These characteristics contributes to the performance and it enables most of the source code are shared among them. But there's some kind of backup messages that breaks consistency of the cluster when the fail-over occurred before the slave receives it. Such messages are listed in Table 10.8. So the master used `SYNC_STANDBY_RESULT` message. Processing such synchronous messages, master sends `SYNC_STANDBY_RESULT` to the slave succeeding to such message. The slave returns response to `SYNC_STANDBY_RESULT`,

Table 10.7: Transaction Backup Messages

Message name	Description
`BEGIN_BACKUP`	Start backup by Standby
`END_BACKUP`	End backup preparation by Standby
`BKUP_NODE_REGISTER`	Backup of `NODE_REGISTER`
`BKUP_NODE_UNREGISTER`	Backup of `NODE_UNREGISTER`
`BKUP_TXN_BEGIN`	Backup of `TXN_BEGIN`
`BKUP_TXN_BEGIN_GETGXID`	Backup of `TXN_BEGIN_GETGXID`
`BKUP_TXN_START_PREPARED`	Backup of `TXN_START_PREPARED`
`BKUP_TXN_COMMIT`	Backup of `TXN_COMMIT`
`BKUP_TXN_COMMIT_PREPARED`	Backup of `TXN_COMMIT_PREPARED`
`BKUP_TXN_PREPARE`	Backup of `TXN_PREPARE`
`BKUP_TXN_ROLLBACK`	Backup of `TXN_ROLLBACK`
`BKUP_TXN_GET_GXID`	
`BKUP_SEQUENCE_INIT`	Backup of `SEQUENCE_INIT`
`BKUP_SEQUENCE_GET_NEXT`	Backup of `SEQUENCE_GET_NEXT`
`BKUP_SEQUENCE_SET_VAL`	Backup of `SEQUENCE_SET_VAL`
`BKUP_SEQUENCE_RESET`	Backup of `SEQUENCE_RESET`
`BKUP_SEQUENCE_CLOSE`	Backup of `SEQUENCE_CLOSE`
`BKUP_SEQUENCE_RENAME`	Backup of `SEQUENCE_RENAME`
`BKUP_SEQUENCE_ALTER`	Backup of `SEQUENCE_ALTER`
`BKUP_TXN_BEGIN_GETGXID_AUTOVACUUM`	Backup of `TXN_BEGIN_GETGXID_AUTOVACUUM`
`BKUP_BARRIER`	Backup barrier to standby
`TXN_GET_NEXT_GXID`	Get next GXID
`TXN_GXID_LIST`	Obtain global transaction list
`BACKEND_DISCONNECT`	tell GTM that the backend disconnected from the proxy
`BKUP_TXN_BEGIN_GETGXID_MULTI`	Backup of `TXN_BEGIN_GETGXID_MULTI`
`BKUP_TXN_COMMIT_MULTI`	Backup of `TXN_COMMIT_MULTI`
`BKUP_TXN_ROLLBACK_MULTI`	Backup of `TXN_ROLLBACK_MULTI`

Table 10.8: Messages Request to be Synchronized

Message Name
SEQUENCE_INIT
SEQUENCE_ALTER
SEQUENCE_GET_NEXT
SEQUENCE_SET_VAL
SEQUENCE_RESET
SEQUENCE_CLOSE
SEQUENCE_RENAME
BEGIN_TRANSACTION
BEGIN_TRANSACTION_GET_GXID
BEGIN_TRANSACTION_GET_GXID_AUTOVACUUM
BEGIN_TRANSACTION_GET_GXID_MULTI
COMMIT_TRANSACTION
COMMIT_PREPARED_TRANSACTION
ROLLBACK_TRANSACTION
COMMIT_TRANSACTION_MULTI
ROLLBACK_TRANSACTION_MULTI
START_PREPARED_TRANSACTION
PREPARE_TRANSACTION

10.4 Network Bottleneck

Because every transaction control requires interaction to the GTM, the network can be suffered heavy network workload. Seeing as backend processes sends messages discretely, so a huge number of short packets flooded the network. Many short packet reduces the efficiency of the network.

For the purpose of reducing the network load, GTM Proxy was made. GTM Proxy packs same kind of frequently used message into single message, and packs multiple messages into single TCP segment (8kiB at maximum) to reduce the number of packets. GTM Proxy appends `MSG_DATA_FLUSH` to packed TCP segment to sync because GTM has to know when to return the packed result message in single TCP segment. Table 10.9 shows the messages that are packed. If GTM Proxy servers are deployed to each node, it can pack messages without leaking packets to the network.

Table 10.9: Packed Transaction Management Messages

From	To	Description
TXN_BEGIN_GETGXID	TXN_BEGIN_GETGXID_MULTI	Start multiple new transactions and get GXIDs
TXN_COMMIT	TXN_COMMIT_MULTI	Commit multiple running or prepared transactions
TXN_ROLLBACK	TXN_ROLLBACK_MULTI	Rollback multiple transactions
SNAPSHOT_GET	SNAPSHOT_GET_MULTI	Get multiple global snapshots

Figure 10.1 shows an example of interaction with GTM Proxy. A line means single message and its color means kind of the message, and a number means message and response ID. The figure shows following.

- If there's no pending request to GTM, GTM Proxy quickly propagates the message from the backend.
- If there's a pending request to GTM, GTM Proxy doesn't propagates the message from the backend until GTM returns the response.
- The message that has different type is not packed into single message.
- If there's multiple pending messages from the backends, GTM Proxy sends them in the same phase.

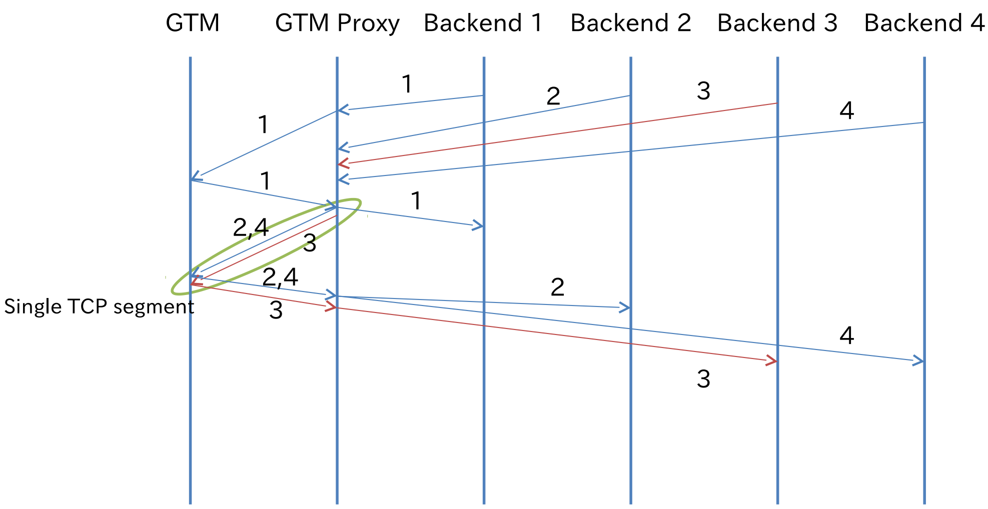

Figure 10.1: Example of GTM Proxy

These characteristics are brought from the internal structure of the GTM proxy.

The GTM proxy is implemented with the worker thread model, and single thread handles multiple connections from the backends and single connection to GTM. After the main server loop `src/gtm/proxy/proxy_main.c:ServerLoop()` accepts a connection from a backend, the connection is dispatched to worker process in `proxy_main.c:GTMProxyAddConnection()`. The example used one worker thread.

`proxy_main.c:GTMProxy_ThreadMain()` is the main loop of worker threads. This loop has two phases.

1st phase reads data from all backend connections, and call `ProcessCommand()` to dispatch received message to `Process`*XXXXXXXX*`Command()`. If the message is packable, `Processi`*XXXXXXXX*`Command()` calls `GTMProxy_CommandPending()` to store the information. If the message is not packaged, calls `GTMProxy_ProxyCommand()` to propagate the message immediately to GTM. Please note that the response is not received here. When all connections has no pending data, received messages are packed into single message, which is send it to the GTM.

2nd phase reads data from the GTM connection and call `ProcessResponse()` to distribute received response to the backends. 2nd phase is repeated until all response correspond to the message received in 1st phase.

Because GTM proxy groups more than one message from coordinator/datanode backend into single packed message, it needs dedicated error handling. GTM protocol was extended to handle this. Without GTM proxy, GTM had direct connection to the backend. So the GTM automatically aborted transaction to cleanup transactions implicitly aborted when the connection is disconnected. With the GTM proxy, the GTM proxy holds connection even if one of the backend's connection handled by a thread disconnected. To avoid the transaction is left, the some packed message includes connection ID that identify the connection received the message. And the GTM proxy sends `MSG_BACKEND_DISCONNECT` to notify a disconnection of a backend. Please note that `MSG_BACKEND_DISCONNECT` message's body doesn't have connection ID. The message notifies the connection ID using `ProxyHdr` which is inserted to every message's body by the GTM proxy.

10.5 Transaction Management at coordinator/datanode backends

Each coordinator/datanode backend needs to connect to GTM to obtain Global Transaction Id (`GXID`) and global snapshot. `gtm.c` module in `src/backend/access/transam` takes care of connection and communication between each backend and GTM.

This section describes functions defined in `gtm.c` and other dedicated transaction handling related to GTM in coordinator/datanode backend processes.

10.5.1 `gtm.c` module

`gtm.c` module handles connection and communication from coordinator/datanode backend and gtm/gtm_proxy.

Functions defined in this module are as follows:

`IsGTMConnected()`

This function checks if connection to GTM is alive. This is called from the following code:

Caller	File & Description
`AtEOXact_GlobalTxn()`	`xact.c` Used to determine what transaction ID should be used at the end of the transaction.

`CheckConnection()`

This function checks if a connection to GTM has been established. If not, it establishes a connection to GTM.

This is a static function and is used only in `gtm.c` locally.

`InitGTM()`

This function establishes a connection to GTM and is used only within `gtm.c` at present.

`CloseGTM()`

This function closes a connection to GTM This is called from the following code:

Caller	File & Description
`PGXCNodeCleanAndRelease()`	`execRemote.c` Called when the backend is ending.

This function is also called within `gtm.c` module internally.

`BeginTranGTM()`

This function informs GTM of the start of a new transaction and obtains global transaction ID (`GXID`).

This function is called from the following codes:

Caller	File & Description
`GetNewTransactionId()`	`varsup.c` Called to obtain global transaction ID.
`GetAuxilliaryTransactionId()`	`xact.c` Used to set auxilliaryTransactionId entry to CurrentTransactionState.

`BeginTranAutovacuumGTM()`

This function is similar to `BeginTranGTM()` but only for autovacuum process. GXID for autovacuum process does not appear in the global snapshot.

This function is called from the following codes:

Caller	File & Description
`GetNewTransactionId()`	`varsup.c` Called when obtaining XID for autovacuum process.

`CommitTranGTM()`

This function tells GTM the specified transaction is committed and sets current transaction ID to invalid value.

This is called from the following codes:

Caller	File & Description
`AtEOXact\_GlobalTxn()`	`xact.c` Called at the end of global transaction. It is called only when it is needed to close the transaction on the GTM

`CommitPreparedTranGTM()`

This function tells GTM to commit a prepared transaction.

This is called from the following codes:

Caller	File & Description
`AtEOXact_GlobalTxn()`	`xact.c` Used to mark the end of global transaction.
`FinishRemotePreparedTransaction()`	`execRemote.c` Used to finish prepared transaction at remote nodes.

`RollbackTranGTM()`

This function tells GTM the specified transaction is aborted and sets current transaction ID to invalid value.

This is called from the following codes:

Caller	File & Description
`AtEOXact_GlobalTxn()`	`xact.c` Called to mark the end of global transaction.
`FinishRemotePreparedTransaction()`	`execRemote.c` Called to finish prepared transaction at remote nodes.

`StartPreparedTranGTM()`

This function tells GTM that prepare transaction commands starts with remote nodes.

This is called from the following codes:

Caller	File & Description
`PreAbort_Remote()`	`execRemote.c` Called to abort remote truncations.
`PostPrepare_Remote()`	`execRemote.c` Called in post-prepare handling in remote nodes.

`PrepareTranGTM()`

This function tells GTM that prepare transaction commands was successful.

This is called from the following codes:

Caller	File & Description
`PreAbort_Remote()`	`execRemote.c` Called at re-abort handling.
`PostPrepare_Remote()`	`execRemte.c` Called at post-prepare handling.

`GetGIDDataGTM()`

This function obtains GTM internal information of fresh GXID, GXID of the prepared transaction, and datanode/coordinator node list involved in the prepare.

This is for the future use.

`GetSnapshotGTM()`

This function obtains global snapshot from GTM.

This is called from the following codes:

Caller	File & Description
`GetSnapshotDataDataNode()`	`procarray.c` Called to get snapshot data for datanode.
`GetSnapshotDataCoordinator()`	`procarray.c` Called to get snapshot data for coordinator.

`CreateSequenceGTM()`

This function creates a sequence on GTM.

This is called from the following codes:

Caller	File & Description
`DefineSequence()`	`sequence.c` Called in creating sequence.

`AlterSequenceGTM()`

This function alters a sequence on GTM.

This is called from the following code:

Caller	File & Description
`AlterSequence()`	`sequence.c` Called to modify the definition of a sequence.

`GetNextValGTM()`

This function gets the next sequence value.

This is called from the following codes:

Caller	File & Description
`nextval\_internal()`	`sequence.c` Called to get the next value of the sequence.

`SetValGTM()`

This functions sets the value of the sequence.

This is called form the following codes:

Caller	File & Description
`do_setval()`	`sequence.c` This is called in handling 2 and 3 argument forms of `SETVAL`.

`DropSequenceGTM()`

This function drops sequence depending on the key type.

This is called from the following codes:

Caller	File & Description
`drop_sequence_cb()`	`sequence.c` Called in a callback of sequence drop.
`dropdb()`	`dbcommands.c` Called in dropping a database.

`RenameSequenceGTM()`

This function renames sequence on GTM.

This is called from the following codes:

Caller	File & Description
`RenameRelationInternal()`	`tablecmds.c` Called in changing the name of a relation.
`AlterTableNamespaceInternal()`	`tablecmds.c` This is called in relocating a table or materialized view to another namespace.
`AlterSeqNamespaces()`	`tablecmds.c` This is called to move all `SERIAL` column sequences of the specified relation to another namespace.
`rename_sequence_cb()`	`sequence.c` Called in sequence rename callback.
`doRename()`	`dependency.c` Called in renaming the given object.

`RegisterGTM()`

This function registers the specified node to GTM. Connection for registering is used just once the closed.

This is called from the following codes:

Caller	File & Description
`sigusr1_handler()`	`postmaster.c` Called in handling signal conditions from child processes.

`UnregisterGTM()`

This function unregisters the given node from GTM. Connection for registering is used just once the closed.

This is called form the following codes:

Caller	File & Description
`pmdie()`	`postmaster.c` Called in a signal handler to handle various postmaster signals.
`sigusr1_handler()`	`postmaster.c` Called in handling signal conditions from child processes.

`ReportBarrierGTM()`

This function reports barrier to GTM. This is used to backup GTM restart point for given barrier id.

Caller	File & Description
`RequestBarrier()`	`barrier.c` Called in handling `CREATE BARRIER` statement.

10.5.2 `xact.c` module

This module has Postgres-XC-specific functions as follows:

`RegisterTransactionLocalNode()`

Marks if the local node has done some write activity.

This is called form the following codes:

Caller	File & Description
`ExecRemoteUtility()`	`execRemote.c`

`ForgetTransactionLocalNode()`

Forgets about the local node's involvement in the transaction.

Called from the following codes:

Caller	File & Description
`CommitTransaction()`	`xact.c`
`PrepareTransaction()`	`xact.c`
`AbortTransaction()`	`xact.c`

`IsTransactionLocalNode()`

Checks if the local node is involved in the transaction.

Called form the following codes:

It is for the future use and is not used at present.

`IsXidImplicit()`

Checks if the given xid is for implicit 2PC.

Called form the following code:

Caller	File & Description
`standard_ProcessUtility()`	`utility.c` Called in handling `PREPARE TRANSACTION` command.

Chapter 11

Planner and Executor

This chapter describes implementaion of `SELECT` statement and other DML handling.

As described in section 5.2.1 at page 60, `planner()` in `planner.c` is the entry point of all the PostgreSQL and Postgres-XC's planner.

In vanilla PostgreSQL, if no planner hook is defined, it is handled by `standard_planner()`. In Postgres-XC, in the originating coordinator, it is handled by `pgxc_planner()` instead. In the case of datanode or non-originating coordinator where the statement is shipped from other coordinator, it is handled by `standard_planner()` just like vanilla PostgreSQL.

`pgxc_planner()` is implemented in `pgxcplan.c`, where the statement is handled by `pgxc_handle_unsupported_stmts()` to check if given statements are not supported by Postgres-XC. If a statement is not supported, then it is treated as an error and the planner returns. If all the statements are suported ones, it is passed to `pgxc_FQS_planner()` to check if the whole statement can be shipped to one or more nodes. If so, then the planner returns the plan to the caller. If not, then the statement is passed to `standard_planner()` for further work.

`pgxcplan.c` implements many utility functions used in the planner, as described in section 5.2.1 in page 60.

There's no Postgres-XC-specific code in `standard_planner()` and all Postgres-XC-specific functionarities are implemented in other functions called from here.

Another module to produce remote plan is `pgxcpath.c`, which creates all the remote execute plan as the node `RemoteQueryPath` defined in `relation.h`. Functions defined in this module is described in the section 5.3.1 at page 63.

11.1 Join Pushdown

Join pushdown is handled in the funciton `create_joinrel_rqpath()` as described in subsection 5.3.1 on page 63.

This function assumes at least one of the join relations has remote query path and checks the path to outer relation and innter relation. Each path has already been set up by `create_plainrel_rqpath()` function defined in `pgxcpath.c`. If either relation does not have remote path, then this function returns without changing the original plan. In this case, typically join with system catalog or materialized view, remote relation is materialized at the coordinator and subsequent join operation is performed locally at the coordinator.

Then it checks if outer join is path is still parameterized and it is not shippable.

If shippable, it checks if join quals are shippable using `pgxc_is_expr_shippable()` defined in `pgxcship.c`. In fact, shippability of each qual has already been set up by `create_remotequery_path()` defined in `pgxcpath.c`.

Next, if join is innter join, all the other quals are combined with join quals to be pushed down. Quals other than join quals are not pushed down in the case of outer join. This can be improved in the future.

Now, all the join fell into simple structure and is ready to check whole join shippability and to build the plan into `ExecNodes` structure by `pgxc_is_join_shippable()` defined in `pgxcship.c`.

Here, shippability condition is as follows:

- Both outer and innter relation has `ExecNodes`.
- Join type is inner join, left outer join or full join. Right outer join is not pushed down.
- In the case of left outer join, inner relation path must be shippable.
- In the case of full outer join, both innter and outer relation path must be shippable.
- Both inner and outer relation are replicated table.
- If both inner and outer relation are distributed, then two of them should be distributed in the same manner, with equi-join on the distribution column and the condition is shippable. In this case, the result is merged at the coordinator.
- If outer relation is distributed and inner one is replicated, both left outer join and inner join are pushed down.
- If outer relation is replicated and inner one is distributed, only inner join is pushed down.
- `ExecNodes` of inner and outer nodes should be able to be merged.

Again, if `pgxc_is_join_shippable()` determines the join is not shippable, original path for inner and outer relations work to materialize them at coordinator and to do the rest of the join operations here in the originating coordinator.

By this step, all the local quals have been pushed down to each path using PostgreSQL planner code to reduce the number of fetched rows.

11.2 Order By Pushdown

`ORDER BY` pushdown is handled by `create_remotesort_plan():pgxcplan.c`. This function checks if `ORDER BY` (and any other sort function) can be pushed down and modifies pased in Sort plan and underlying `Remote Query` plan.

11.3 Limit Pushdown

`LIMIt` pushdown is handled by `create_remotelimit_plan():pgxcplan.c`. Similar to `ORDER BY` push down, it checks if `LIMIT` clause is pushable. If so, then it pushes `limitcount` and `limitoffset` if defined. This is done by modifying `RemoteQuery` node.

11.4 Group By Pushdown

`GROUP BY` pushdown is handled by `create_remotegrouping_plan():pgxcplan.c`. Current restriction is as follows:

1. Only plain aggregates (no expressions involving aggregates) and/or expressions in `GROUP BY` clause are pushed down.
2. `DISTINCT` and `ORDER BY` clause are not pushed down.
3. Window functions are not pushed down.
4. `HAVING` clause is not pushed down.

11.5 Window Function Handling

At present, window functions are not pushed down unless whole statement can be shipped.

11.6 Aggregate Function Handling

In Postgres-XC, aggregate function handling need an extension from PostgreSQL. The background is simple pushdown may not work for some kind of aggregate functions. For example, in the calculation of average, we cannot push down `avg()` function to datanodes to calculate global average. Instead, we need to obtain sum and count from each datanode to calculat the final result.

For this extension, Postgres-XC introduced new function layer called **collector function**. The document is available at `http://postgres-xc.sourceforge.net/docs/1_2_1/sql-createaggregate.html`.

Internally, for example, `avg()` function definition in the system catalog is defined in `pg_aggregate.h` as shown in `http://postgres-xc.sourceforge.net/docs/1_2_1/catalog-pg-aggregate.html`.

In `pg_aggregate` system catalog, column `aggcollectfn` was added to define new collector function. Definition of `avg()` function in `pg_aggregate.h` source code is as follows:

```
/* avg */
#ifdef PGXC
  DATA(insert ( 2100  int8_avg_accum  numeric_avg_collect numeric_avg     0   1231    "
      {0,0}" "{0,0}" ));
  DATA(insert ( 2101  int4_avg_accum  int8_avg_collect    int8_avg        0   1016    "
      {0,0}" "{0,0}" ));
  DATA(insert ( 2102  int2_avg_accum  int8_avg_collect    int8_avg        0   1016    "
      {0,0}" "{0,0}" ));
  DATA(insert ( 2103  numeric_avg_accum   numeric_avg_collect numeric_avg     0   1231
         "{0,0}" "{0,0}" ));
  DATA(insert ( 2104  float4_accum    float8_collect  float8_avg      0   1022    "
      {0,0,0}" "{0,0,0}" ));
  DATA(insert ( 2105  float8_accum    float8_collect  float8_avg      0   1022    "
      {0,0,0}" "{0,0,0}" ));
  DATA(insert ( 2106  interval_accum  interval_collect    interval_avg    0   1187    "
      {0 second,0 second}" "{0 second,0 second}" ));
#endif
```

In each line with `DATA` keyword, the second function corresponds to the collector function for various data types.

Example of collector function is as follows:

```
Datum
int8_avg_collect(PG_FUNCTION_ARGS)
{
    ArrayType  *collectarray;
    ArrayType  *transarray = PG_GETARG_ARRAYTYPE_P(1);
    Int8TransTypeData *collectdata;
    Int8TransTypeData *transdata;

    /*
     * If we're invoked by nodeAgg, we can cheat and modify our first
     * parameter in-place to reduce palloc overhead. Otherwise we need to make
     * a copy of it before scribbling on it.
     */
    if (fcinfo->context &&
        (IsA(fcinfo->context, AggState) ||
         IsA(fcinfo->context, WindowAggState)))
        collectarray = PG_GETARG_ARRAYTYPE_P(0);
    else
        collectarray = PG_GETARG_ARRAYTYPE_P_COPY(0);

    if (ARR_HASNULL(collectarray) ||
        ARR_SIZE(collectarray) != ARR_OVERHEAD_NONULLS(1) + sizeof(Int8TransTypeData))
        elog(ERROR, "expected 2-element int8 array");
    collectdata = (Int8TransTypeData *) ARR_DATA_PTR(collectarray);

    if (ARR_HASNULL(transarray) ||
        ARR_SIZE(transarray) != ARR_OVERHEAD_NONULLS(1) + sizeof(Int8TransTypeData))
        elog(ERROR, "expected 2-element int8 array");
    transdata = (Int8TransTypeData *) ARR_DATA_PTR(transarray);

    collectdata->count += transdata->count;
    collectdata->sum += transdata->sum;

    PG_RETURN_ARRAYTYPE_P(collectarray);
}
```

11.7 Global Sequence Implementation

Postgres-XC has to ensure the uniqueness of the sequence value in a cluster. Similar to global transaction management, coordinators needs sequence management of global tables on GTM. GTM is responsible for it. GTM-side implementaion is described in section 5.5 at page 88.

The global tables mentioned above doesn't include temporary tables. Postgres-XC manages the sequences on temporary tables locally just like PostgreSQL does.

Postgres-XC didn't modify sequence handling in core planner and the executor.

Major changes in sequence handling are listed below:

- Define a new sequence:
  1. Define a new sequence
  2. Call `CreateSequenceGTM()` to register sequence information to GTM.
  3. Define a local relation for the new sequence.
  4. Register a callback function to drop the defined sequence when the transaction is aborted.
- Alter sequence information

   1. Call `AlterSequenceGTM()` to alter sequence information in GTM.
   2. Update local sequence information.

- Rename a sequence
   1. Call `RenameSequenceGTM()` to rename the sequence in GTM.
   2. Update local sequence information.
   3. Register a callback function to rename the renamed sequence to original name when the transaction is aborted.

- Get next sequence values
   1. Call `GetNextValGTM()` to rename the sequence in GTM.
   2. Update local sequence information with the values returned by GTM.

- Set sequence value
   1. Call `SetValGTM()` to set the sequence value in GTM.
   2. Update local sequence information.

Did you find that the coordinators have sequence relations like PostgreSQL? Yes, you can refer cached value at there. But please note that the sequence values in GTM could be modified by other coordinators. So `pg_dump` calls `nextval` SQL function to obtaion latest value instead of looking the sequence tables.

Chapter 12

DML

12.1 Top level statment

Basically, XC does not handle `INSERT`, `UPDATE` and `DELETE` statements specifically except for replicated table handling.

These statements are analyzed, rewritten and planned before execution. Subqueries are also planned using vanilla PostgreSQL code. During the analyze, remote tables are marked and they are planned into `RemoteQuery`.

12.2 Returning Clause

If any statement has `RETURNING` clause, this information is set to `retruningList` member of `Query` structure. If DML has `RETURNING` clause, this information is set to `returningList` member of `InsertStmt`, `DeleteStmt` and `UpdateStmt` structure. It is vanilla PostgreSQL structure and no modification was made in Postgres-XC.

So far, `RETURNING` clause is handled as shippable in Postgres-XC, as found in `pgxc_shippability_walker()`.

12.3 DML Handling for Replicated Tables

In updating repliated tables, each coordinator visits **bf primary** node first gor each replicated table updates. This is needed to serialize conflicting updates and to prevent inconsisitent updates among nodes. By doing this, when updates at primariy node is successful, then coordinator can visit other nodes in any order. Conflicting transactions will be blocked at primary node until the first successful transaction finishes.

Through analyzer and other functions, primary node list of a given replicate table will be set to the member `primarynodelist` of `ExecNodes` structure by `GetRelationNodes():locator.c`. This value is tested in the executor, `executeRemote.c`. In the case of copy in (copying tuples to a table), it is handled by `DataNodeCopyIn():execRemote.c`. If primary node is defined for the rlation, the executor extracts one connection to a primary node and handle this first before other replica are handled.

Similar handling was done in `get_exec_connections():execRemote.c` and `redustrub.c: distrib_copy_from()` too.

12.4 Copy statement handling

`COPY TO` collects data from datanodes and `COPY FROM` distributes data to datanodes. These are quite similar to `INSERT` and `SELECT` and look like a DML. But actually they are utility statements. In addition to distinct implementation, `COPY` statements have to calcurate which node and what query to send by themselevs because the planner doesn't analyze utility statements and doesn't locate where to go the query for them. There's a point here. When the coordinator received

COPY FROM, the query doesn't specify column with non-shippable default value, the coordinator have to complete both a query and data.

In addition, COPY processes data in consecutive segments. It means the following actions will be taken in parallel.

- Coordinator receives data from client and propages it to datanodes
- Datanodes receive data from a coordinator and processes it

To calculate involved nodes and save distribution information into structRemoteCopyState, RemoteCopy_GetRelationLoc() is implemented. And RemoteCopy_BuildStatement() builds the query shipped to datanodes. Ofcourse the built query includes the column which has non-shippable default value. These functions are called from BeginCopy(). This function is common to COPY TO and COPY FROM.

CopyFrom() processes COPY FROM statement. To begin the copy, pgxcNodeCopyBegin() prepares a global transaction and connections to datanodes and send COPY FROM query to datanodes. Each copied data obtained from PostgreSQL client by NextCopyFrom() is checked which node to go by GetRelationNodes() and it is sent to datanodes by DataNodeCopyIn(). If COPYFROM need to support binary format data, the signature that indicates binary mode need to be sent to datanodes just after the query is sent to datanodes.

CopyTo() processes COPY TO statement. To begin the copy, pgxcNodeCopyBegin() prepares a global transaction and connections to datanodes and send COPY TO query to datanodes. DataNodeCopyOut() reads copied data from each connection to datanode until it receives message with type 'Z' (Ready For Query) in handle_response(). Copied data rows will come as messages with type 'd' (CopyOutDataRow). They are handled in HandleCopyDataRow() using the storage specified at the combiner. If COPY TO need to support binary format data, the signature that indicates binary mode need to be sent to PostgreSQL client just after the query is sent to datanodes.

COPY statement is implemented in src/backend/command/copy.c and main part of Postgres-XC specific logic is in src/backend/pgxc/copy/remotecopy.c and src/backend/pgxc/pool/execRemote.c. remotecopy.c has utility functions used in copy.c. For example, RemoteCopy_GetRelationLoc() creates RelationLocInfo which has location information of relations involved to the query. RemoteCopy_BuildStatement() builds a copy query executed at datanodes. execRemote.c has functions to execute actual process of the copy command using information prepared in copy.c. Examples include pgxcNodeCopyBegin() and DataNodeCopyIn().

Chapter 13

Session and system functions

This chapter describes additional session and system functions for Postgres-XC.

`xc_pool_reload()`

This function refreshes cached information of `pgxc_node` catalog and cleans up pooled connections managed by the pooler.

`is_committed(transaction_id)`

This function reoprts if a transaction with given GXID is committed or not. Returned information is just local to the issued node. This is intended to be used in `pgxc_clean` or other utilities to cleanup and recover commit status of any two-phase commit transactions.

`pgxc_version()`

This function returns Postgres-XC version.

`pgxc_pool_check()`

This function checks if connection data cached in the pooler is consistent with `pgxc_node`.

`pgxc_lock_for_backup()`

This function locks the cluster for taking backup of the node to be exported to the new node being added.

Chapter 14

Miscellaneous Feature

14.1 Additional `postgresql.conf` configuration parameters

This section describes additional `postgresql.conf` configuration parameters specific to Postgres-XC.

`enable_fast_query_shipping`

This is boolean parameter to specify if fast query shipping is enabled. Usually this should be `ON`.

`enable_remotegroup`

This is boolean parameter to specify if group-by push down to remote nodes is enabled. Usually this should be `ON`.

`enable_remotejoin`

This is boolean parameter to specify if join push down to remote nodes is enabled. Usually this should be `ON`.

`enable_remotelimit`

This is boolean parameter to specify if limit pushdown to remote nodes is enabled. Usually this should be `ON`.

`enable_remotesort`

This is boolean parameter to specify if ORDER BY pushdown to remote node is enabled. Usually this should be `ON`.

`enforce_two_phase_commit`

This is boolean parameter to specify if two phase commit protocol is used for write transactions more than two nodes are involved.

`gtm_backup_barrier`

This is boolean parameter to specify if GTM restart point is backed up for barrier.

gtm_host

This is character string parameter to specify host name of GTM. If you configure GTM proxy, you should specify host name of GTM proxy.

gtm_port

This is integer parameter to specify port number of GTM. If you configure GTM proxy, you should specify port number of GTM proxy.

max_coordinators

This is integer parameter to specify the maximum number of coordinators in the cluster.

max_datanodes

This is numeric parameter to specify the maximum number of datanodes in the cluster.

max_pool_size

This is numeric parameter to specify the maximum number of pooled connection in the pooler.

min_pool_size

This is numeric parameter to specify the minimum number of pooled connection in the pooler.

persistent_datanode_connections

This is boolean parameter to specify if the connections from coordinator to datanodes should keep assigned.

pgxc_node_name

This is character string parameter to specify the node name of itself.

pooler_port

This is numeric parameter to specify the port number of the pooler.

remotetype

Used to identify what is connecting to the backend. Usually, do not modify this parameter.

require_replicated_table_pkey

Boolean parameter to specify if it is not allowed replicated tables without primary key or another unique key combination involved. If this is turned on and no such unique key is not involved, the statement fails.

xc_maintenance_mode

Boolean parameter to control if write operation is allowed in `EXECUTE DIRECT`. This parameter cannot turn on in `postgresql.conf`. Only a superuser can turn it on in a session.

14.2 Additional SQL syntax for Postgres-XC

The following lists Postgres-XC-specific SQL statement syntax. Refer to Postgres-XC documentation for details.

- `ALTER NODE` statement.
- `DISTRIBUTE BY` clause in `ALTER TABLE` statement.
- `CLEAN CONNECTION` statement.
- Collection function in `CREATE AGGREGATE` statement.
- `CREATE BARRIER` statement.
- `CREATE NODE` statement.
- `CREATE NODE GROUP` statement.
- Additional options for `EXPLAIN` statement.
- `DISTRIBUTE BY` clause in `CREATE TABLE` statement.
- `DROP NODE` statement.
- `DROP NODE GROUP` statement.

Chapter 15

Regression Tests

This chapter describes changes to regression test.

15.1 General Changes

Most of the statements in the regression tests are used as is, or with minimum modification. General modification for Postgres-XC is as follows:

- No `DISTRIBUTE BY` clause was given to `CREATE TABLE` or `ALTER TABLE` statement, if the first column is allowd as the distribution column.
- If the first column is not allowd as the distribution column and another column is found suitable, `DISTRIBUTE BY HASH(`*colname*`)` clause is added.
- If no column is suitable for the distribution column, such table was defined as replicated table using `DISTRIBUTE BY REPLICATION`.
- Because order of rows of `SELECT` statement or `RETURNING` clause depends upon the order of execution among datanodes, `ORDER BY` clause was added to make this order reproductive.
- To make `EXPLAIN` result portable, `NODES OFF` and `NUM_NODES OFF` options were added to make test result independent from the number of nodes and their names.

For specific test purpose, some tables in existing PostgreSQL regression test are created as replicated or round-robin distribution.

Example of additional use of `ORDER BY` clause in `join.sql` is shown below.

```
-- Outer joins
-- Note that OUTER is a noise word
--

SELECT '' AS "xxx", *
  FROM J1_TBL LEFT OUTER JOIN J2_TBL USING (i)
  ORDER BY i, k, t;

SELECT '' AS "xxx", *
  FROM J1_TBL LEFT JOIN J2_TBL USING (i)
  ORDER BY i, k, t;

SELECT '' AS "xxx", *
  FROM J1_TBL RIGHT OUTER JOIN J2_TBL USING (i)
  ORDER BY i, j, k, t;
```

Example of additional option use in `EXPLAIN` in `join.sql` is shown below.

```
--
-- test case where a PlaceHolderVar is propagated into a subquery
--

explain (num_nodes off, nodes off, costs off)
```

```
select * from
  int8_tbl t1 left join
  (select q1 as x, 42 as y from int8_tbl t2) ss
  on t1.q2 = ss.x
where
  1 = (select 1 from int8_tbl t3 where ss.y is not null limit 1)
order by 1,2;

select * from
  int8_tbl t1 left join
  (select q1 as x, 42 as y from int8_tbl t2) ss
  on t1.q2 = ss.x
where
  1 = (select 1 from int8_tbl t3 where ss.y is not null limit 1)
order by 1,2;
```

Please note that `ORDER BY` clause is given only to the subsequent `SELECT` statement because regression test needs to test the plan without `ORDER BY` close while subsequent `SELECT` statement needs to produce portable result.

Please also not that `COSTS OFF` option in `EXPLAIN` statements are from vanilla PostgreSQL regression test to make explain result portable too.

15.2 Additional Test for Postgres-XC

Regression test prefixed with `xc_` is Postgres-XC-specific regression test.

Table 15.1 describes each test.

Table 15.1: Postgres-XC-specific Regression Test

Test Name	Description
`xc_alter_table`	Checks Postgres-XC-specific behavior of `ALTER TABLE` statement, such as dropping distribution column is not allowed.
`xc_constraints`	Checks constraint shippability in Postgres-XC for tables with different distribution strategies.
`xc_create_function`	Defines a couple of function used by other Postgres-XC-specific regression tests.
`xc_distkey`	Tests all the supported data types are working as distribution key. Also verifies that comparison with a constant for equality is optimized.
`xc_for_update`	Test of `FOR UPDATE` support in Postgres-XC.
`xc_FQS`	Test if fast query shipping works correctly in various distribution strategy and statements.
`xc_FQS_join`	Dedicated test for join fast query shipping.
`xc_groupby`	Test of `GROUP BY` pushdown and cross node operation for the combination of distributed and replicated tables.
`xc_having`	Tests `HAVING` clause handling for various combination of table distribution.
`xc_limit`	Tests `LIMIT` and `OFFSET` clause push down.
`xc_misc`	Various feature test including plpgsql functions.
`xc_node`	Tests Postgres-XC node management statements.
`xc_params`	Tests non-simple variables (record types without %rowtype descriptor) in SQL statements inside a plpgsql function.
`xc_prepared_xacts`	Tests prepared transactions are working as expected. Tests it is not prepared if a transaction is read only.
`xc_remote`	Tests Postgres-XC remote queries are working. It is done disabling fast query shipping to see standard planner works correctly.
`xc_returning`	Specific `RETURNING` clause test.
`xc_sequence`	Specific sequence test, including checking callback mechanisms on GTM.
`xc_sort`	Test merge sort optimization in Postgres-XC. This works to fetch ordered data from datanodes and merge them at a coordinator.
`xc_temp`	Test temporary object behavior.
`xc_triggers`	Trigger test for various table distribution and trigger functions.
`xc_trigship`	Dedicated test for shippable and non-shippable trigger functions.

Chapter 16

Benchmark Extension

16.1 DBT-1 Benchmark

Postgres-XC uses DBT-1 as basic benchmark test to measure its performance for each build. As described in the section 1.6.1 at page 24, distribution or replication option was added to each table based on its characteristics of the size, update frequency, and join operation with others.

Outline of the table design is illustrated in Figure 1.14 in page 25.

Modified DBT-1 source code will be found in sourceforge reporitory with the url as `git://git.code.sf.net/p/postgres-xc/dbt1postgres-xc-dbt1`.

Other major modification to the benchmark is as follows:

- `author` table is replicated (`create_table.sql`).
- `country` table is replicated (`create_table.sql`).
- `address` table has additional column `addr_c_id` for its customer ID and is hash-distributed using `addr_c_id` (`create_table.sql`).
- `customer` table is hash-distributed using `c_id` (customer ID) (`create_tables.sql`).
- `item` table is replicated and does not have `i_stock` column (`create_tables.sql`).
- `i_stock` column of `item` table is now in the new table `stock`, which is hash-distributed using `st_i_id` (stock item id) (`create_tables.sql`).
- `st_i_id` column is the primary key of `stock` table (`create_indexes.sql`).
- `orders` table is hash-distributed using `o_c_id` column (order cumstomer id) (`create_tables.sql`).
- (`o_id`, `o_c_id`) is the primary key for `orders` table (`create_indexes.sql`).
- `orders` table has additional index over `i_o_c_id` column.
- `order_line` table has additional column `ol_c_id` (order line cumstomer id) and is hash-distributed using `ol_c_id` column (`create_tables.sql`).
- (`ol_o_id`, `ol_id`, `ol_c_id`) is the primary key for `order_line` table (`create_indexes.sql`).
- `cc_xacts` table has additional column `cx_c_id` (cc customer id), which is hash-distributed using `cx_c_id` (`create_tables.sql`).
- (`cx_o_id`, `cx_c_id`) is the primary key of `cc_xact` table (`create_indexes.sql`).
- `shopping_cart` table is hash-distributed using `sc_id` (shopping cart id).
- `shopping_cart_line` table has additional column `scl_c_id` (shopping cart customer id), which is hash-distributed using `scl_sc_id` (shopping cart id) (`create_tables.sql`).
- Foreign key from `adress` table to `customer` table was modified to just an index (`create_fk.sql`).
- Foreign key from `order_line` table to `orders` table was modified (`create_fk.sql`).

- Foreign key from `stock` table to `item` table was added.
- Message modified to reflect changes in table deesign. Modifications are indicated by comments with "`pgxc`."
- SQL statement modified to reflect changes in table design. Modifications are indicated by comments with "`pgxc`."
- Original ODBC interface to the database was changed to libpq (`libpq_com` and `inc_psql_libpq` directories).
- `dbdriver` files are modified according to the interface change from ODBC to libpq. Changes are indicated with `\#ifdef` directive using `LIBPQ` (`eu.c` and `main.c`).

16.2 Pgbench

Modification to pgbench benchmark is as follows:

- For write benchmark, each table was distributed by modulo using bid. The backend to choose modulo, not hash, is that the number of bid value is relatively small and it is not easy to have a good balance of each datanode workload.
- For read benchmark, each table was replicated.

The above changes are reasonable because it is a commo practice to have different table design and arrangement for different workloads.

Part II

Postgres-XC Cluster Design, Configuration and Operation

This section outlines how to configure and operate Postgres-XC database cluster through `pgxc_ctl`. `Pgxc_ctl` source material will be found in the `contrib` module of Postgres-XC distribution. Please note that `pgxc_ctl` does not support all the possible configurations of Postgres-XC, and this is mainly the reason why it is placed at `contrib` module, not in the main source tree.

Although there are some configuration which `pgxc_ctl` does not support, it's a good idea to see what should be done in Postgres-XC cluster configuration and operation and what is being done in each operation internally.

Chapter 17

Writing configuration

17.1 Overview of pgxc_ctl configuration file and environment

Pgxc_ctl configuration file is in fact a bash script. That is, you can write any bash script which helps you to define your Postgres-XC configuration. In later sections, you will find many of such examples.

Default name of the configuration file is pgxc_ctl.conf. You can specify other configuration file with -c option to pgxc_ctl command. The path is absolute or relative to pgxc_ctl directory as described in the next paragraph.

Pgxc_ctl assumes dedicated directory to store its log and other materials. The default directory is $HOME/pgxc_ctl. You can change this by specifying --home option when you start pgxc_ctl. Pgxc_ctl has some more options to control its behavior such as log level and verbosity. You can specify this in ".pgxc_ctl" file placed in your home directory. Each line specifies option and its value such as:

```
[koichi@buildfarm:~]$ cat .pgxc_ctl
xc_prompt 'PGXC$ '
#verbose y
#logMessage 'DEBUG3'
#printMessage 'DEBUG1'
#printLocation y
#logLocation y
#debug y
[koichi@buildfarm:~]$
```

xc_prompt is pgxc_ctl prompt in a string (it does not support serial number or other fancy staff as in bashl). Value of verbose should be y or n. logMessage is the level of the message goes to the log. You can specify MANDATORY, PANIC, ERROR, WARNING, NOTICE, NOTICE2, INFO, DEBUG1, DEBUG2 or DEBUG3. printMessage is the level of the message goes to the terminal you're running pgxc_ctl. printLocation is for debug to print location of pgxc_ctl source code with messages. Usually specify n. Debug also prints some more message for debugging. Usually, specify n.

".pgxc_ctl" environment file is optional. All the default values will be taken if no environment file is found.

Pgxc_ctl log will be printed to the directory pgxc_log under pgxc_ctl directory unless you specify this explicitly with -L option when you start pgxc_ctl.

17.2 Get configuration file template

First of all, you may need configuration file template to begin with. You may not have pgxc_ctl directory. In this case, run pgxc_ctl from your home directory like this.

```
[koichi@node01:~]$ pgxc_ctl prepare
Installing pgxc_ctl_bash script as /home/koichi/pgxc_ctl/pgxc_ctl_bash.
ERROR: File "/home/koichi/pgxc_ctl/pgxc_ctl.conf" not found or not a regular file. No
     such file or directory
Installing pgxc_ctl_bash script as /home/koichi/pgxc_ctl/pgxc_ctl_bash.
Reading configuration using /home/koichi/pgxc_ctl/pgxc_ctl_bash --home /home/koichi/
     pgxc_ctl --configuration /home/koichi/pgxc_ctl/pgxc_ctl.conf
Finished to read configuration.
   ******** PGXC_CTL START ***************

Current directory: /home/koichi/pgxc_ctl
[koichi@node01:~]$ ls pgxc_ctl
coordExtraConfig  pgxc_ctl.conf  pgxc_log/
[koichi@node01:~]$
```

You can specify pgxc_ctl command as pgxc_ctl command line option. With several messages, your pgxc_ctl directory and configuration file are built.

You can specify configuration file name to build as:

```
[koichi@node01:~]$ pgxc_ctl prepare my_pgxc_ctl.conf
Installing pgxc_ctl_bash script as /home/koichi/pgxc_ctl/pgxc_ctl_bash.
ERROR: File "/home/koichi/pgxc_ctl/pgxc_ctl.conf" not found or not a regular file. No
     such file or directory
Installing pgxc_ctl_bash script as /home/koichi/pgxc_ctl/pgxc_ctl_bash.
Reading configuration using /home/koichi/pgxc_ctl/pgxc_ctl_bash --home /home/koichi/
     pgxc_ctl --configuration /home/koichi/pgxc_ctl/pgxc_ctl.conf
Finished to read configuration.
   ******** PGXC_CTL START ***************

Current directory: /home/koichi/pgxc_ctl
[koichi@node01:~]$ ls pgxc_ctl
coordExtraConfig  my_pgxc_ctl.conf  pgxc_log/
[koichi@node01:~]$
```

Please note that you don't have to make pgxc_ctl directory. If not found, pgxc_ctl will make this directory when it runs.

Later on, we use $HOME/pgxc_ctl as pgxc_ctl directory and pgxc_ctl.conf as configuration file respectively. Both are the default.

17.3 How configuration file looks like

The next figure shows the outline of pgxc_ctl configuration file. Details of each portion will be described later, section by section. Again, because the configuration file is bash script, you can use `bash` capability to specify specific configuration. You will see how template configuration uses this.

```
#!/bin/bash
#
# Postgres-XC Configuration file for pgxc_ctl utility.
#
# Configuration file can be specified as -c option from pgxc_ctl command.   Default is
# $PGXC_CTL_HOME/pgxc_ctl.org.
#
# This is bash script so you can make any addition for your convenience to configure
# your Postgres-XC cluster.
#
# Please understand that pgxc_ctl provides only a subset of configuration which pgxc_ctl
# provide.  Here's several several assumptions/restrictions pgxc_ctl depends on.
#
...(omitted)...
# 8) Killing nodes may end up with IPC resource leak, such as semaphore and shared
    memory.
#    Only listening port (socket) will be cleaned with clean command.
#
# 9) Backup and restore are not supported in pgxc_ctl at present.   This is a big task
    and
#    may need considerable resource.
#
#
    ==========================================================================================

#
#
# pgxcInstallDir variable is needed if you invoke "deploy" command from pgxc_ctl utility
    .
# If don't you don't need this variable.
pgxcInstallDir=$HOME/pgxc
#---- OVERALL -----------------------------------------------------------------------
#
pgxcOwner=koichi                # owner of the Postgres-XC database cluster.  Here, we
    use this
                                # both as lines user and database user.  This must be
                                # the super user of each coordinator and datanode.
```

First lines are comments for the general description how the configuration file is composed. You may want to read this a bit carefully to avoid problems and pitfalls.

The configuration file's goal is to specify values of pre-defined variables.

17.4 Common configuration section

You will see common configuration section at the top. In this section, you define the directory where your Postgres-XC binaries are installed, and the set of servers where you're configuring Postgres-XC cluster.

The section looks like:

```
# pgxcInstallDir variable is needed if you invoke "deploy" command from pgxc_ctl utility
    .
# If don't you don't need this variable.
pgxcInstallDir=$HOME/pgxc
#---- OVERALL -------------------------------------------------------------------
#
pgxcOwner=koichi        # owner of the Postgres-XC database cluster.  Here, we use this
                        # both as lines user and database user.  This must be
                        # the super user of each coordinator and datanode.
pgxcUser=$pgxcOwner     # OS user of Postgres-XC owner
```

```
tmpDir=/tmp                   # temporary dir used in XC servers
localTmpDir=$tmpDir           # temporary dir used here locally

configBackup=n                    # If you want config file backup, specify y to this
    value.
configBackupHost=pgxc-linker    # host to backup config file
configBackupDir=$HOME/pgxc      # Backup directory
configBackupFile=pgxc_ctl.bak   # Backup file name --> Need to synchronize when original
     changed.
```

`pgxcInstallDir` variable:

First, you will find the variable `pgxcInstallDir`| This is the directory where Postgres-XC binaries are installed locally. This value is the `--prefix` option value of configure utility used to build Postgres-XC binary from the source code. If you run make and make install, by specifying `--prefix` option as `$pgxcInstallDir` value, you will have `$pgxcInstallDir` like this:

```
[koichi@buildfarm:pgxc]$ pwd
/home/koichi/pgxc
[koichi@buildfarm:pgxc]$ ls -F
bin/  include/  lib/  share/
[koichi@buildfarm:pgxc]$
```

This is used to deploy these binaries to servers with `deploy` command of `pgxc_ctl`. If you're installing binaries with other means, you don't have to worry about this variable.

`pgxcOwner` variable:

Second, you will find `pgxcOwner` variable. This variable specifies owner user of Postgres-XC database.

`pgxcUser` variable:

Next, you will find `pgxcUser` variable. This variable specifies operating system user of each server you're running Postgres-XC. `Pgxc_ctl` uses `ssh` for the operation of Postgres-XC component and assumes that key-based authentication is configured between the server `pgxc_ctl` is running and other servers where you run Postgres-XC components. Key-based authentication configuration is out of the scope of `pgxc_ctl`|

`tmpDir` variable:

`tmpDir` variable specifies the work directory used in `pgxc_ctl` locally. Typical value can be `/tmp`. Depending upon your operating system, another value can be preferred. You may want to use `$HOME/tmp` or other user-specific directory.

`localTmpDir` variable:

`localTmpDir` variable specifies a work directory used in the servers where you're running Postgres-XC components. `Pgxc_ctl` uses the same work directory among all the servers.

`configBackup` variable:

`configBackup` variable specifies if you're backing up configuration file. When you change Postgres-XC cluster configuration by adding/removing nodes or promoting slave to master, `pgxc_ctl` updates your configuration file by adding new lines to specify such changes. If you specify the value "`y`" to this variable, `pgxc_ctl` will backup this change to the file specified by the following variables.

Although the template specifies "`n`," it specifies its backup configuration for your help.

`configBackupHost` variable:

`configBackupHost` variable specifies what server you'd like to backup your `pgxc_ctl` configuration file. It will be a good idea to backup to different server so that you can take this backup and run `pgxc_ctl` at this server when the current `pgxc_ctl` server fails.

`configBackupDir` variable:

`configBackupDir` variable specifies the directory where `pgxc_ctl` configuration file backup is stored. If you don't specify "`y`" to `configBackup` variable, you don't have to worry about this variable.

`configBackupFile` variable:

`configBackupFile` variable specifies the file name of `pgxc_ctl` configuration backup. Unless you specify "`y`" to `configBackup` variable, you don't have to worry about this variable.

17.5 GTM master configuration

Following is GTM master section of `pgxc_ctl` configuration template. It looks very simple.

```
#---- GTM ------------------------------------------------------------

# GTM is mandatory.  You must have at least (and only) one GTM master in your Postgres-
    XC cluster.
# If GTM crashes and you need to reconfigure it, you can do it by pgxc_update_gtm
    command to update
# GTM master with others.   Of course, we provide pgxc_remove_gtm command to remove it.
     This command
# will not stop the current GTM.  It is up to the operator.

#---- Overall -------
gtmName=gtm

#---- GTM Master -----------------------------------------------

#---- Overall ----
gtmMasterServer=node13
gtmMasterPort=20001
gtmMasterDir=$HOME/pgxc/nodes/gtm

#---- Configuration ---
gtmExtraConfig=none         # Will be added gtm.conf for both Master and Slave (done at
    initialization only)
gtmMasterSpecificExtraConfig=none   # Will be added to Master's gtm.conf (done at
    initialization only)
```

`gtmName` variable:

`gtmName` variable defines the node name for GTM. GTM master and slave shares this. Because we have only one GTM master in the cluster, you may not have a chance to use this name in the cluster operation.

`gtmMasterServer` variable:

`gtmMasterServer` variable is the server you are running GTM master.

`gtmMasterPort` variable:

`gtmMasterPort` variable is TCP port number GTM uses to accept connections from GTM-Proxy or coordinator/datanode backend. You should assign unique port number in the host `$gtmMasterServer`.

`gtmMasterDir` variable:

`gtmMasterDir` variable is the work directory for GTM master. Similar to PostgreSQL server, GTM needs dedicated work directory to store its configuration file, status, log and other information.

`gtmExtraConfig` and `gtmMasterSpecificExtraConfig` variable:

In most cases, your GTM configuration is complete with above three configuration parameters. `Pgxc_ctl` takes other configuration variables and composes GTM master configuration file. If you want to specify extra configuration parameter to GTM master, you can use `gtmExtraConfig` and `gtmMasterSpecificExtraConfig` variable.

`gtmExtraConfig` variable specifies the file name where additional `gtm.conf` configuration lines are stored. Contents of these files will go to `gtm.conf` file of both GTM master and slave. `gtmMasterSpecificExtraConfig` variable specifies the file name where `gtm.conf` configuration lines only for GTM master is stored.

Details of `gtm.conf` file will be found at `http://postgres-xc.sourceforge.net/docs/1_2_1/app-gtm.html`. Default value of these variables are set to “`none`,” which means “nothing.” You can specify the value “`none`” for file or server names if you don’t specify any.

`pgxc_ctl` specifies `listen_addresses`|, `port` and `nodename` startup configuration parameters in `gtm.conf`. You should not specify these configuration values in `gtmExtraConfig` or `gtmMasterSpecificExtraConfig` files. If you’d like to specify contents of, for example, `gtmExtraConfig` file, you can do it by adding lines as shown below:

```
#---- Configuration ---
gtmExtraConfig=gtmExtraConfig       # Will be added gtm.conf for both Master and Slave (done at initialization only)
cat > $gtmExtraConfig << EOF
log_min_messages = DEBUG1
EOF
gtmMasterSpecificExtraConfig=none   # Will be added to Master's gtm.conf (done at initialization only)
```

Because the configuration file is a `bash` script, these additional lines will setup the file without supplying additional files.

17.6 GTM slave configuration

GTM slave section of `pgxc_ctl` configuration template is as follows:

```
#---- GTM Slave ----------------------------------------------

# Because GTM is a key component to maintain database consistency, you may want to
    configure GTM slave
# for backup.

#---- Overall ------
gtmSlave=y                  # Specify y if you configure GTM Slave.   Otherwise, GTM
    slave will not be configured and
                            # all the following variables will be reset.
gtmSlaveServer=node12       # value none means GTM slave is not available.  Give none if
     you don't configure GTM Slave.
gtmSlavePort=20001          # Not used if you don't configure GTM slave.
gtmSlaveDir=$HOME/pgxc/nodes/gtm    # Not used if you don't configure GTM slave.
# Please note that when you have GTM failover, then there will be no slave available
    until you configure the slave
# again. (pgxc_add_gtm_slave function will handle it)

#---- Configuration ----
gtmSlaveSpecificExtraConfig=none # Will be added to Slave's gtm.conf (done at
    initialization only)
```

`gtmSlave` variable:

This variable specifies if you use GTM slave. Specify “`y`” if you are configuring GTM slave. Skip this section otherwise.

`gtmSlaveServer` variable:

Specify the server name you're running GTM slave.

`gtmSlavePort` variable:

Specify the port number GTM slave accepts connections. This has to be unique in the server you specified in `gtmSlaveServer` variable.

`ggtmSlaveDir` variable:

Specify the work directory for GTM slave. This has to be unique in the server you specified in `gtmSlaveServer` variable.

`gtmSlaveSpecificExtraConfig` variable:

Specify the file name with `gtm.conf` configuration file entries specific to this GTM slave. For details of `gtm.conf`| please refer to `http://postgres-xc.sourceforge.net/docs/1_2_1/app-gtm.html`. You will find how to setup this file in the configuration file in the last section.

`pgxc_ctl` specifies `listen_addresses`, `port`, and `nodename` startup configuration parameters and you should not specify these configuration values in `gtmSlaveSpecificExtraConfig` file.

17.7 GTM proxy configuration

GTM Proxy is not mandatory in Postgres-XC configuration. Because it provides GTM slave promotion to the master without interpreting Postgres-XC cluster operation, you may want to configure this as well unless you're configuring Postgres-XC locally for a test.

It's a good idea to configure a GTM proxy, a coordinator and a datanode in a server balance the workloads among these components and to leverage local socket.

GTM proxy configuration section looks like this:

```
#---- GTM Proxy --------------------------------------------------------
# GTM proxy will be selected based upon which server each component runs on.
# When fails over to the slave, the slave inherits its master's gtm proxy.  It should be
# reconfigured based upon the new location.
#
# To do so, slave should be restarted.   So pg_ctl promote -> (edit postgresql.conf and recovery.conf) -> pg_ctl restart
#
# You don't have to configure GTM Proxy if you don't' configure GTM slave or you are happy if every component connects
# to GTM Master directly.  If you configure GTL slave, you must configure GTM proxy too.

#---- Shortcuts ------
gtmProxyDir=$HOME/pgxc/nodes/gtm_pxy

#---- Overall -------
gtmProxy=y      # Specify y if you configure at least one GTM proxy.   You may not configure gtm proxies
                # only when you don't' configure GTM slaves.
                # If you specify this value not to y, the following parameters will be set to default empty values.
                # If we find there're no valid Proxy server names (means, every servers are specified
                # as none), then gtmProxy value will be set to "n" and all the entries will be set to
                # empty values.
gtmProxyNames=(gtm_pxy1 gtm_pxy2 gtm_pxy3 gtm_pxy4) # No used if it is not configured
gtmProxyServers=(node06 node07 node08 node09)       # Specify none if you don't' configure it.
gtmProxyPorts=(20001 20001 20001 20001)             # Not used if it is not configured.
gtmProxyDirs=($gtmProxyDir $gtmProxyDir $gtmProxyDir $gtmProxyDir)  # Not used if it is not configured.

#---- Configuration ----
gtmPxyExtraConfig=none # Extra configuration parameter for gtm_proxy.  Coordinator section has an example.
gtmPxySpecificExtraConfig=(none none none none)
```

`gtmProxyDir` variable:

This is a shortcut to specify same value for `gtmProxyDirs` array elements as described later.

`gtmProxy` variable:

`gtmProxy` specifies if you are configuring GTM proxy. Specify "`y`" if you are configuring GTM proxy. Specify "`n`" otherwise.

`gtmProxyNames` variable:

`gtmProxyNames` specifies names of GTM proxies. Because GTM proxies are configured in more than one server, each GTM proxy need to have unique name which is specified in an array. In this template, GTM proxy, coordinator and datanode are configured in four servers.

`gtmProxyServers` variable:

`gtmProxyServers` specifies server for each GTM proxy. This is also an array. Specify servers for corresponding GTM proxy specified in `gtmProxyNames`.

`gtmProxyPorts` variable:

gtmProxyPorts specifies port number of each GTM proxy. This is also an array like gtmProxyNames. Port number must be unique in each servers specified in gtmProxyServers parameter.

gtmProxyDirs variable:

GTM proxy needs dedicated work directory. gtmProxyDirs parameter specifies this. In the template, work variable gtmProxyDir is used to assign the same value to each array element. You can use similar way for you convenience.

gtmPxyExtraConfig variable:

Specify the file name which contain extra gtm_proxy.conf configuration lines. Content of this file will go to all the gtm_proxy.conf files you are configuring. Specify "none" if you are not using this feature.

Details if gtm_proxy.conf file will be found at http://postgres-xc.sourceforge.net/docs/1_2_1/app-gtm-proxy.html.

listen_addresses, worker_threads and gtm_connect_retry_interval configuration options of gtm_proxy.conf will be set by pgxc_ctl and you should not change them with gtmPxyExtraConfig and gtmPxySpecificExtraConfig.

pgxc_ctl will also setup nodename| port, gtm_host and gtm_port. They comes at the last of gtm_proxy.conf file. Specifying them in gtmPxyExtraConfig or gtmPxySpecificExtraConfig will not work.

gtmPxySpecificExtraConfig variable:

You can specify extra configuration for each GTM proxy with this parameter. Specify file name which contains extra gtm_proxy.conf lines for each GTM proxy as an element of this array. Specify "none" element value if you don't use this.

17.8 Coordinator master configuration

If you became familiar with GTM proxy configuration, you will find coordinator and datanode configuration is quite similar. Yes, it has just a few more addition.

Coordinator master configuration section looks as follows. Please be careful that coordinator slave configuration is at the middle of this configuration, which will be explained in the next section.

```
#---- Coordinators ----------------------------------------------------------------------------------------------------

#---- shortcuts ----------
coordMasterDir=$HOME/pgxc/nodes/coord
coordSlaveDir=$HOME/pgxc/nodes/coord_slave
coordArchLogDir=$HOME/pgxc/nodes/coord_archlog

#---- Overall ------------
coordNames=(coord1 coord2 coord3 coord4)        # Master and slave use the same name
coordPorts=(20004 20005 20004 20005)            # Master and slave use the same port
poolerPorts=(20010 20011 20010 20011)           # Master and slave use the same pooler port
coordPgHbaEntries=(192.168.1.0/24)              # Assumes that all the coordinator (master/slave) accepts
... (Omitted) ...
```

```
#---- Master -------------
coordMasterServers=(node06 node07 node08 node09)        # none means this master is not available
coordMasterDirs=($coordMasterDir $coordMasterDir $coordMasterDir $coordMasterDir)
coordMaxWALsernder=5    # max_wal_senders: needed to configure slave. If zero value is specified,
                        # it is expected to supply this parameter explicitly by external files
                            # specified in the following.   If you don't configure slaves, leave this value to zero.
coordMaxWALSenders=($coordMaxWALsernder $coordMaxWALsernder $coordMaxWALsernder $coordMaxWALsernder)
                        # max_wal_senders configuration for each coordinator.

#---- Slave -------------
        ... (omitted) ...

#---- Configuration files---
# Need these when you'd like setup specific non-default configuration
# These files will go to corresponding files for the master.
# You may supply your bash script to setup extra config lines and extra pg_hba.conf entries
# Or you may supply these files manually.
coordExtraConfig=coordExtraConfig   # Extra configuration file for coordinators.
                                # This file will be added to all the coordinators'
                                # postgresql.conf
# Please note that the following sets up minimum parameters which you may want to change.
# You can put your postgresql.conf lines here.
cat > $coordExtraConfig <<EOF
#================================================
# Added to all the coordinator postgresql.conf
# Original: $coordExtraConfig
log_destination = 'stderr'
logging_collector = on
log_directory = 'pg_log'
listen_addresses = '*'
max_connections = 100
EOF

# Additional Configuration file for specific coordinator master.
# You can define each setting by similar means as above.
coordSpecificExtraConfig=(none none none none)
coordExtraPgHba=none    # Extra entry for pg_hba.conf.  This file will be added to all the coordinators' pg_hba.conf
coordSpecificExtraPgHba=(none none none none)
```

First three variable settings for `coordMasterDir`, `coordSlaveDir` and `coordArchDir` are shortcuts to specify the same value to each array element. You can write any script for your convenience.

`coordNames` variable:

Specify each coordinator name in this array element.

`coordMasterDirs` variable:

Specify the working directory for each coordinator in this array element. In this template, `coordMasterDir` variable is used to assign the same value to all the elements.

`coordPorts` variable:

Specify the port number which each coordinator uses to accept connection from application or other coordinators. This value must be unique in the server specified in `coordMasterServers` variable and `coordSlaveServers` variable if you are configuring coordinator slaves.

This template is based upon circular HA configuration where each coordinator slave runs at the next server and master and its slave uses the same port. Please note that each coordinator is assigned different port to meet this configuration.

`poolerPorts` variable:

Coordinator implements connection pooler internally to pool connection to other coordinators and datanodes. This variable specifies port number which the pooler uses internally. The value must be unique in the server specified in `coordMasterServers` variable and `coordSlaveServers` variable if you are configuring coordinator slaves,

`coordPgHbaEntries` variable:

This is a shortcut of configuring `pg_hba.conf` file of each coordinator. Each element specified in this array will be converted into "`host all` *`xxx`* `trust`" format to go to `pg_hba.conf` where *xxx* is the value of the element. If you don't like to have such setups, you should use `coordExtraPgHba` or `coordSpecificExtraPgHba` variable.

`coordMasterServers` variable:

This array specifies what server each coordinator runs.

`coordMasterDirs` variable:

This array specifies work the directory of each coordinator. Please note that this template uses variable `coordMasterDir` to assign the same value to each array element.

`coordMaxWalSenders` variable:

This array specifies `max_wal_sender` configuration parameter value for each coordinator. If you are configuring coordinator slave, this value must be positive.

`coordExtraConfig` and `coordSpecificExtraConfig` `coordextraconfig` specifies the file name which contains postgresql.conf configuration entries for all the coordinators. The following lines are the script to set up the file.

Just like GTM proxy, you can specify `postgresql.conf` file entry for each coordinator with `coordSpecificExtraConfig` array. Specify "`none`" for the element value if you don't use it.

`pgxc_ctl` will set up `port`, `pooler_port`, `gtm_host` and `gtm_port` configuration at the last part of coordinator's `postgresql.conf` file. Reconfiguring these parameters using `coordExtraConfig` and `coordSpecifcExtraConig` will not work.

If you are configuring coordinator slave, `pgxc_ctl` will configure `wal_level`, `archive_mode`, `archive_command`, and `max_wal_senders` as well at the last part. Reconfiguring these parameters using `coordExtraConfig` and `coordSpecificExtraConfig` will not work either in this case.

`coordExtraPgHba` and `coordSpecificExtraPgHba` variable:

`coordExtraPgHba` specifies the file name which contains lines to go to `pg_hba.conf` file of all the coordinators.

Each element of `coordSpecificExtraPgHba` array specifies the file name which contains lines of `pg_hba.conf` file for each coordinator.

17.9 Coordinator slave configuration

Please note that `pgxc_ctl` configures coordinator slaves to use the same port as their masters.

Configuration section for coordinator slave looks like this:

```
#---- Slave -------------
coordSlave=y            # Specify y if you configure at least one coordinator slave.
    Otherwise, the following
                        # configuration parameters will be set to empty values.
                        # If no effective server names are found (that is, every servers
                             are specified as none),
                        # then coordSlave value will be set to n and all the following
                            values will be set to
                        # empty values.
coordSlaveSync=y        # Specify to connect with synchronized mode.
coordSlaveServers=(node07 node08 node09 node06)         # none means this slave is not
    available
coordSlaveDirs=($coordSlaveDir $coordSlaveDir $coordSlaveDir $coordSlaveDir)
coordArchLogDirs=($coordArchLogDir $coordArchLogDir $coordArchLogDir $coordArchLogDir)
```

`acoordSlave` variable:

Specify "`y`" if you are configuring coordinator slaves. Otherwise, specify "`n`."

`coordSlaveSync` variable:

Specify "`y`" if you use synchronous wal shipping for the slave. At present, you should specify "`y`" because asynchronous wal shipping could lose some transactions at promote which could make the cluster inconsistent.

`coordSlaveServers` variable:

Specify which servers each coordinator slave runs.

`coordSlaveDirs` variable:

Specify the work directory for each coordinator slave.

`coordArchLogDirs` variable:

Specify a directory to receive WAL archive for each coordinator slave.

17.10 Datanode master configuration

Datanode master and slave configuration is very similar to coordinator master and slave configuration. One major difference is that datanodes does not have the pooler.

Datanode master configuration section is as follows:

```
#---- Datanodes ------------------------------------------------------------------------------------------------------

#---- Shortcuts --------------
```

```
datanodeMasterDir=$HOME/pgxc/nodes/dn_master
datanodeSlaveDir=$HOME/pgxc/nodes/dn_slave
datanodeArchLogDir=$HOME/pgxc/nodes/datanode_archlog

#---- Overall ---------------
#primaryDatanode=datanode1              # Primary Node.
# At present, xc has a problem to issue ALTER NODE against the primary node.  Until it is fixed, the test will be done
# without this feature.
primaryDatanode=datanode1               # Primary Node.
datanodeNames=(datanode1 datanode2 datanode3 datanode4)
datanodePorts=(20008 20009 20008 20009) # Master and slave use the same port!
datanodePgHbaEntries=(192.168.1.0/24)   # Assumes that all the coordinator (master/slave) accepts
                                        # the same connection
                                        # This list sets up pg_hba.conf for $pgxcOwner user.
                                        # If you'd like to setup other entries, supply them
                                        # through extra configuration files specified below.
# Note: The above parameter is extracted as "host all all 0.0.0.0/0 trust".   If you don't want
# such setups, specify the value () to this variable and supply what you want using datanodeExtraPgHba
# and/or datanodeSpecificExtraPgHba variables.

#---- Master ----------------
datanodeMasterServers=(node06 node07 node08 node09) # none means this master is not available.
                                                    # This means that there should be the master but is down.
                                                    # The cluster is not operational until the master is
                                                    # recovered and ready to run.
datanodeMasterDirs=($datanodeMasterDir $datanodeMasterDir $datanodeMasterDir $datanodeMasterDir)
datanodeMaxWalSender=5                              # max_wal_senders: needed to configure slave. If zero value is
                                                    # specified, it is expected this parameter is explicitly supplied
                                                    # by external configuration files.
                                                    # If you don't configure slaves, leave this value zero.
datanodeMaxWALSenders=($datanodeMaxWalSender $datanodeMaxWalSender $datanodeMaxWalSender $datanodeMaxWalSender)
                        # max_wal_senders configuration for each datanode

#---- Slave -----------------

        ... (Omitted) ...

# ---- Configuration files ---
# You may supply your bash script to setup extra config lines and extra pg_hba.conf entries here.
# These files will go to corresponding files for the master.
# Or you may supply these files manually.
datanodeExtraConfig=none    # Extra configuration file for datanodes.  This file will be added to all the
                            # datanodes' postgresql.conf
datanodeSpecificExtraConfig=(none none none none)
datanodeExtraPgHba=none     # Extra entry for pg_hba.conf.  This file will be added to all the datanodes' postgresql.conf
datanodeSpecificExtraPgHba=(none none none none)
```

Similar to the coordinator, slave configuration is placed in the middle, which will be described in the next section.

`datanodeMasterDir`, `datanodeSlaveDir` and `datanodeArchiLogDir` are shortcuts used in the following configuration.

`primaryDatanode` variable:

This configuration is unique to the datanode, specifying primary datanode name. Primary datanode is the datanode where replicated table update takes place first. This is how to maintain replicated table consistent. In the future release of Postgres-XC, primary datanode may be determined automatically and this parameter may become obsolete.

`datanodeNames` variable:

This array specifies the name of each datanode. Node name of primaryDatanode has to be specified in one of the element.

`datanodePorts` variable:

Specifies the port number which datanode postmaster uses to accept connections. Master and slave of each datanode uses the same port number and this number has to be unique in the servers running datanode master or slave, if configured.

`datanodePgHbaEntries` variable:

Shortcut to specify `pg_hba.conf` file of each datanode. Please see `CoordPgHbaEntires` description for details.

`datanodeMasterServers` variable:

This array specifies server names where each datanode master runs.

`datanodeMasterDirs` variable:

This array specifies the work directory for each datanode master. This hast to be unique in the server where the coordinator master is running.

`datanodeMaxWalSenders` variable:

This array specifies `max_wal_senders` configuration parameter for each datanode's `postgresql.conf`. If you are configuring datanode slave, this value has to be positive.

`datanodeExtraConfig` variable:

Specify the file name which contains extra lines for postgresql.conf file of all the datanodes. Specify "`none`" if you are not using this.

`datanodeSpecificExtraConfig` variable:

This array specifies the file name which contains extra lines for `postgresql.conf` file of each corresponding datanode.

`datanodeExtraPgHba` variable:

Specify the file name which contains additional lines for `pg_hba.conf` file of all the datanodes. Specify "`none`" if you are not using this.

`datanodeSpecificExtraPgHba` variable:

This array specifies the file name which contains extra lines for `pg_hba.conf` of each corresponding datanode.

17.11 Datanode slave configuration

Similar to coordinators, datanode slave uses the same port number as its master.

Datanode slave configuration section looks like:

```
#---- Slave -----------------
datanodeSlave=y  # Specify y if you configure at least one coordinator slave.  Otherwise, the following
                 # configuration parameters will be set to empty values.
                 # If no effective server names are found (that is, every servers are specified as none),
                 # then datanodeSlave value will be set to n and all the following values will be set to
```

```
                  # empty values.
datanodeSlaveServers=(node07 node08 node09 node06)  # value none means this slave is not available
datanodeSlaveSync=y      # If datanode slave is connected in synchronized mode
datanodeSlaveDirs=($datanodeSlaveDir $datanodeSlaveDir $datanodeSlaveDir $datanodeSlaveDir)
datanodeArchLogDirs=( $datanodeArchLogDir $datanodeArchLogDir $datanodeArchLogDir $datanodeArchLogDir )
```

`datanodeSlave` variable:

Specify "`y`" if you are configuring datanode slaves. Otherwise, specify "`n`."

`datanodeSlaveServers` variable:

This array specifies the server where each datanode slave is running.

`datanodeSlaveSync` variable:

Specify if you are using synchronous wal shipping. To maintain database consistency, please specify just "`y`" here to avoid a chance to lose transactions at promotion.

`datanodeSlaveDirs` variable:

This array specifies the directory for each datanode.

`datanodeArchLogDirs` variable:

This array specifies the directory to receive each datanode's archive WAL.

Chapter 18

Initializing Postgres-XC cluster

This section describes how to initialize your Postgres-XC cluster.

When you obtain `pgxc_ctl` configuration file template with `pgxc_ctl` prepare command, you have built `$HOME/pgxc_ctl` directory and your `pgxc_ctl.conf` file at this directory.

You have designed your Postgres-XC configuration and edited `pgxc_ctl.conf` file.

You have configured key-based ssh connection authentication from the computer you are running `pgxc_ctl` to each server you are running one or more Postgres-XC components.

Now you are ready to initialize your Postgres-XC cluster with `pgxc_ctl`.

18.1 Invoke `pgxc_ctl`

Now invoke `pgxc_ctl` as your shell command. If `pgxc_ctl` does not find any error in your configuration, it will print a prompt asking for a command.

If `pgxc_ctl` reports any configuration error, correct errors and try again.

18.2 Deploy Postgres-XC binaries to servers

You should deploy all Postgres-XC binaries to all the servers you are running Postgres-XC components. If you have installed this by binary package or manually, you can skip this section.

If you are deploying binaries with `pgxc_ctl`, then type `deploy all` and return. `pgxc_ctl` will visit servers where at least one Postgres-XC component runs and copy binaries to their installation directory specified in `pgxcInstallDir` configuration variable.

Please note that `deploy all` does not take care of `PATH` environment in your shell. You should do this manually.

18.3 Initialize the cluster

Type `init all` and return. `pgxc_ctl` will do everything needed to configure and start up your Postgres-XC database cluster.

Although `pgxc_ctl` provides more step-by-step initialization, this is for the test and does not provide cluster configuration using `CREAT NODE` statement. It is more convenient to use init all command.

If there's something wrong, errors will be reported. Don't worry. If you need any correction to your configuration file and do it over from the scratch, you should do the following.

1. Issue `kill all` command against `pgxc_ctl` command prompt to kill all the processes at servers. It it doesn't work, then you should kill all the process of `gtm`, `gtm_proxy` and `postgres` manually by visiting each server.
2. Issue `clean all` command against `pgxc_ctl` prompt to clean up all the working directories.
3. Fix the issue in the configuration file or other settings for `pgxc_ctl`.
4. If you need to have additional servers to be involved and if you have deployed Postgres-XC binaries using `deploy all` command, issue `deploy` *`newserver`* command agains `pgxc_ctl` prompt, which deploys Postgres-XC binaries to *newserver*. Otherwise, install Postgres-XC binary in your way.
5. Restart this step from the beginning.

18.4 What `init all` does

`Init all` command does plenty of work inside to initialize each component and configure them to work together. The outline is described below.

Initializes GTM master

1. Kills gtm process if exists, removes the work directory if exists and then creates it.
2. Runs initgtm utility to initialize gtm environment.
3. Configures `gtm.conf` file for the master.
4. Sets up GTM to start with appropriate GXID value.
5. Starts GTM master

Initializes GTM slave if configured

1. Kills gtm process if exists, remove work directory if exists and then create it.
2. Runs initgtm to initialize gtm environment.

3. Configures `gtm.conf` file for the slave.
4. Starts GTM slave.

Initializes GTM proxies if configured

The following steps are done for each gtm_proxy in parallel.

1. Kills gtm_process if exists, remove work directory if exists and them create it.
2. Runs `initgtm` to initialize gtm_proxy environment.
3. Configures `gtm_proxy.conf` file.
4. Starts GTM proxy.

Initializes coordinator masters

The following steps are done for each coordinator master in parallel.

1. Initializes the work directory.
2. Runs `initdb` to initialize a coordinator.
3. Configures `postgresql.conf` file.
4. If coordinator slave is configured, adds wal shipping configuration to `postgresql.conf` file.
5. Starts coordinator master.

Initializes coordinator slaves if configured

The following steps are done for each coordinator slave in parallel.

1. Initializes the work directory.
2. Runs `pg_basebackup` utility to build the base backup.
3. Configures `recovery.conf`.
4. Adds `postgresql.conf` configuration entries to run as the slave.
5. Starts coordinator slave.

Initializes datanode masters

The following steps are done for each datanode master in parallel.

1. Initializes the work directory.

2. Runs `initdb` to initialize a datanode.
3. Configures `postgresql.conf` file.
4. If datanode slave is configured, adds wal shipping configuration to `postgresql.conf` file.
5. Starts datanode master.

Initializes datanode slaves if configured

The following steps are done for each datanode slave in parallel.

1. Initializes the work directory
2. Runs `pg_basebackup` utility to build the base backup.
3. Configures `recovery.conf`.
4. Adds `postgresql.conf` configuration entries to run as the slave.
5. Starts datanode slave.

Configures nodes

1. Runs `CREATE NODE` and `ALTER NODE` statement at each coordinator to finalize the node configuration and make each coordinator ready to accept connections.

Chapter 19

Build your database

When you are successful in init all `pgxc_ctl` command, you are ready to run psql or other utilities. Most of PostgreSQL utilities are ported to Postgres-XC.

They accept `-h` and `-p` command line option to specify what coordinator to connect to. As an alternative, `pgxc_ctl` provides two built-in commands, `Createdb` and `Psql`|

They choose one of the available coordinator and connect to it. You can specify what coordinator to connect with `-` followed by a coordinator name to connect to, not host name or port number.

So you can create your own database by issuing `Createdb` *newdb* against `pgxc_ctl` prompt, or `pgxc_ctl` command argument.

Chapter 20

Run your SQL statements

`pgxc_ctl` provides `Psql` built-in command which invokes psql against specified coordinator. You can specify the coordinator name after '`-`' argument like

```
$ Psql - coord1
```

Where, `coord1` is the coordinator name. If you don't specify coordinator name, `pgxc_ctl` will choose one. You can specify any other psql command options too.

Then you can issue any coordinator Postgres-XC SQL statements.

Chapter 21

Writing applications

Postgres-XC's libpq interface is binary compatible with PG so you can write your application with the same manner as PostgreSQL. Because of the clustering nature, there are several SQL statements which Postgres-XC does not support. Also, there are several SQL statements specific to Postgres-XC. For details, please refer to Postgres-XC document at `http://postgres-xc.sourceforge.net/docs/1_1/` or `http://postgres-xc.sourceforge.net/docs/1_2_1/`.

Chapter 22

Backing up Postgres-XC cluster

22.1 pg_dump and pg_dumpall

As in the case of PostgreSQL, pg_dump and pg_dumpall are the basic backup tool of Postgres-XC. You can connect to one of the coordinators using -h and -p option (sorry, pgxc_ctl does not provide built-in command such as Pg_dump or Pg_dumpall so far). This is almost the same as PostgreSQL.

Backup is consistent and can be restored using psql or pg_restore.

22.2 WAL-shipping backup

You can configure Postgres-XC coordinator and datanode to enable WAL-shipping backup manually. At present, pgxc_ctl does not support this feature. This section does not provide any further description on it so far.

pgxc_ctl provides master/slave configuration and failover of each node. Please use this feature now.

Chapter 23

Recovery from the backup

23.1 Recovery with pg_dump and pg_dumpall

You can restore the database using the backup made by `pg_dump` or `pg_dumpall`| First, re-initialize your cluster and then apply the dump using `psql` (when the backup was taken in text format) or `pg_restore`.

23.2 Recovery from WAL shipping archive

For the same reason as section 22, this is out of the scope of this section.

Chapter 24

Node failover

If you configure slave of GTM, coordinator or datanode and one of hem fails, you can promote the slave and switch over the master.

`pgxc_ctl` provides only manual promotion, not automatic failover. The background is as follows:

1. Automatic failover should be integrated with other resource failover, such as server hardware, network, storage and other software resource including web server and application server.
2. 1 depends upon individual system integration/configuration and it may not be acceptable in some case to provide automatic failover system just within database system.

The following sections will describe `pgxc_ctl` command interface to promote slaves.

24.1 GTM slave promotion

When GTM master fails and you are running GTM slave, you can promote GTM slave to the master. Here is how to do it with `pgxc_ctl`.

You have configured GTM Proxy

With GTM Proxy, you can promote GTM slave without stopping Postgres-XC cluster. If live transactions needs to communicate with GTM while GTM master is out, they will be aborted but you don't have to restart nodes.

First, issue `failover gtm` command at `pgxc_ctl` command prompt like:

```
PGXC$ failover gtm
```

Then, you issue reconnect `gtm_proxy all` command like:

```
PGXC$ reconnect gtm_proxy all
```

With this command, all gtm_proxies will connect to the new master.

Through this step, the following will be done:

1. Runs `gtm_ctl promote` command at gtm slave.
2. Configures `gtm.conf` of the promoted gtm so that it starts as the master next time.
3. Updates your configuration file to reflect these changes. Backup it if specified.

Please note that these commands does not stop old GTM master.

You have not configured GTM Proxy

`pgxc_ctl` does not provide a convenient way to deal with this situation. You have to do the following manually.

1. Runs `gtm_ctl` promote command at gtm slave.
2. Edits `postgresql.conf` file so that they connect to the new gtm master.
3. Restarts all the coordinators and datanodes.

24.2 Coordinator slave promotion

If any coordinator fails and it has a slave running, you can promote it. To do this, you should invoke `failover coordinator` command like:

```
PGXC$ failover coordinator coordname
```

where *coordname* is the coordinator name to promote.

`pgxc_ctl` does the following:

1. Because coordinator slave is running at a different server for the master, determines which gtm_proxy promoting coordinator should connect.
2. Unregisters the coordinator from GTM.
3. Promotes the slave using `pg_ctl promote` command.
4. Edits `postgresql.conf` file to reflect the change in target gtm_proxy. If gtm_proxy is not configured in the server, gtm will be chosen.
5. Issues `pg_ctl restart` to reflect these changes.

6. Updates `pgxc_ctl` configuration file and backup it if specified.
7. Issues `ALTER NODE` statement and `pgxc_pool_reload()` function at all the coordinators to reflect this change.

Please note that all the other coordinator masters should be running to handle `ALTER NODE` statement.

24.3 Datanode slave promotion

If any datanode fails and it has a slave running, you can promote it. To do this, you should invoke failover datanode command like:

```
PGXC$ failover datanode datanodename
```

where *datanodename* is the datanode name to promote.

`pgxc_ctl` will do the following:

1. Because datanode slave is running at a different server for the master, determines which gtm_proxy promoting datanode should connect.
2. Unregisters the datanode from GTM.
3. Promotes the slave using `pg_ctl promote` command.
4. Edits `postgresql.conf` file to reflect the change in target gtm_proxy. If gtm_proxy is not configured in the server, gtm will be chosen.
5. Issues `pg_ctl restart` to reflect these changes.
6. Updates `pgxc_ctl` configuration file and backup it if specified.
7. Issues `ALTER NODE` statement and `pgxc_pool_reload()` function at all the coordinators to reflect this change.

Please note that all the coordinator masters should be running to handle `ALTER NODE` statement.

Chapter 25

Adding nodes

`pgxc_ctl` provides series of command to add nodes. While adding a node, you don't have to stop the whole Postgres-XC cluster but some node may need restart. This section describes the basics of each node addition.

25.1 Adding GTM slave

If you did not configure GTM slave or you don't have GTM slave because original GTM slave has been promoted to the master, you can add GTM slave to your Postgres-XC cluster.

`pgxc_ctl` provides `add gtm slave` command for this purpose. The syntax of the command is as follows:

```
PGXC$ add gtm slave name host port dir
```

name, *host*, *port*, and *dir* are the node name, host where GTM slave runs, port assigned to GTM slave to accept connections and its working directory, respectively.

Adding GTM slave does not affect active transactions.

When adding GTM slave, `pgxc_ctl` does the following:

1. Updates pgxc_ctl configuration file and backup if specified.
2. Initializes gtm slave and start it.

25.2 What about GTM master?

GTM master is Postgres-XC's vital component and it has to be configured and running. `pgxc_ctl` does not provide a means to "**add**" GTM master. To move GTM master to other server, run gtm slave at the target server and promote it.

25.3 Adding a GTM proxy

If you are adding coordinator or datanode at a server where gtm_proxy is not configured, you may want to add gtm_proxy at this server.

You can do this by issuing `add gtm_proxy` command like:

```
PGXC$ add gtm_proxy name host port dir
```

name, *host*, *port*, and *dir* are the node name, host where the GTM proxy runs, port assigned to GTM proxy to accept connections and its working directory, respectively.

If you have not installed Postgres-XC binary to the server, you should do it as described in the Section 18.2.

When adding GTM proxy, `pgxc_ctl` will do the following:

1. Update pgxc_ctl configuration file and backup it if specified.
2. Configure GTM proxy and start it.

25.4 Adding a coordinator master

If you have not installed Postgres-XC binary to the server, you should do it as described in the section 18.2.

If you are adding a coordinator master at a server where GTM proxy is not configured, you may want to configure it first, as described in the section 25.3.

Adding a coordinator master in `pgxc_ctl` is simple. Just invoke `add coordinator master` command like:

```
PGXC$ add coordinator master name host port pooler dir
```

name, *host*, *port*, *pooler*, *dir* are the node name, host where the new coordinator master runs, port assigned to the coordinator to accept connections, port assigned to coordinator connection pooler, and its working directory, respectively.

When adding a coordinator master, `pgxc_ctl` will do the following:

1. Update pgxc_ctl configuration file and back up it if specified.
2. Initialize the working directory and run `initdb` to for initial configuration of the new coordinator master.
3. Determine GTM proxy or GTM to use and update new coordinator master's `postgresql.conf` file.
4. Edit `pg_hba.conf` file to accept minimum connection specified in `coordPgHbaEntries` variable. See section 17.8 for details.

5. Choose an active coordinator and issue `pgxc_lock_for_backup()` to block DDL issued to all the active coordinators.

6. Choose an active coordinator and issue `pg_dumpall` to dump all the catalog information to be imported to the new coordinator master.

7. Start the new coordinator master with `-Z restoremode` and import the catalog exported at the step 6.

8. Stop the new coordinator and start it with `-Z coordinator` option as a coordinator.

9. Issue `CREATE NODE` or `ALTER NODE` and then `pgxc_pool_reload()` at all the coordinators to reflect the change.

10. Close the session opened in the step 5 to release DDL lock.

25.5 Adding a coordinator slave

Please consider to install Postgres-XC binaries and configure GTM proxy as described in the section 25.4.

You can add a coordinator slave just as follows:

```
PGXC$ add coordinator slave name host dir archDir
```

name, *host*, *dir* and *archDir* are the node name, host where the new coordinator slave runs, its working directory and the directory to receive WAL archive from its master, respectively.

When adding a coordinator slave, `pgxc_ctl` will do the following:

1. Initialize the working directory and archive WAL directory.

2. Reconfigure the master's `postgresql.conf` file to begin WAL shipping.

3. Reconfigure the master's `pg_hba.conf` file to accept WAL shipping connection from the new slave.

4. Update pgxc_ctl configuration file and backup it if specified.

5. Restart the master to reflect changes done in 2 and 3.

6. Run `pg_basebackup` to build the master's base backup at the slave's work directory to start with.

7. Update the slave's `postgresql.conf` to run as a slave.

8. Configure the slave's `recovery.conf` file to connect to the master for log shipping.

9. Start the slave.

25.6 Adding a datanode master

Adding a datanode master is similar to adding a coordinator master as described in the section 25.4. Please consider to install Postgres-XC binaries and GTM proxy if needed, as described in the section 18.2.

To add a datanode master, you can issue add datanode master command as follows:

```
PGXC$ add datanode master name host port dir
```

name, *host*, *port*, and *dir* are the node name, host where the new datanode master runs, port number used to accept connections, and the working directory, respectively.

Please note that adding a datanode master does not redistribute the table data automatically because you can specify a set of nodes to distribute or replicate each table. To redistribute tables, use `ALTER TABLE` statement as described in `http://postgres-xc.sourceforge.net/docs/1\_2\_1/sql-altertable.html` and `http://postgres-xc.sourceforge.net/docs/1\_1/sql-altertable.html`.

When adding a datanode master, `pgxc_ctl` will do the following:

1. Update pgxc_ctl configuration file and back up it if specified.
2. Initialize the working directory and run `initdb` to for initial configuration of the new datanode master.
3. Determine GTM proxy or GTM to use and update new datanode master's `postgresql.conf` file.
4. Edit `pg_hba.conf` file to accept minimum connection specified in `datanodePgHbaEntries` variable. See section 17.10 for details.
5. Choose an active coordinator and issue `pgxc_lock_for_backup()` to block DDL issued to all the active coordinators [1].
6. Choose an active datanode and issue `pg_dumpall` to dump all the catalog information to be imported to the new coordinator master.
7. Start the new datanode master with `-Z restoremode` and import the catalog exported at the step 6.
8. Stop the new datanode and start it with `-Z datanode` option as a datanode.
9. Issue `CREATE NODE` and `pgxc_pool_reload()` at all the coordinators to reflect the change.
10. Close the session opened in the step 5 to release DDL lock.

[1] In the current release, `pgxc_lock_for_backup()` is targeted to a datanode master and does not propagate to other nodes. It should have targeted to a coordinator. Fix will be committed and available at the next minor release.

25.7 Adding a datanode slave

Please note that the master datanode must be configured and running to add a datanode slave. Please also consider to install Postgres-XC binaries and configure GTM proxy if needed, as described in the section 18.2.

Adding datanode slave is quite similar to adding coordinator slave. You can do this as follows:

```
PGXC$ add datanode slave name host dir archDir
```

name, *host*, *dir* and *archDir* are the node name, host where the new datanode slave runs, its working directory and the directory to receive WAL archive from its master, respectively.

When adding a datanode slave, `pgxc_ctl` will do the following:

1. Initialize the working directory and archive WAL directory.
2. Reconfigure the master's `postgresql.conf` file to begin WAL shipping.
3. Reconfigure the master's `pg_hba.conf` file to accept WAL shipping connection from the new slave.
4. Update pgxc_ctl configuration file and backup it if specified.
5. Restart the master to reflect changes done in 2 and 3.
6. Run `pg_basebackup` to build the master's base backup at the slave's work directory to start with.
7. Update the slave's `postgresql.conf` to run as a slave.
8. Configure the slave's `recovery.conf` file to connect to the master for log shipping.
9. Start the slave.

Chapter 26

Removing nodes

As mentioned, GTM master is a vital Postgres-XC component and it is not allowed to remove it. GTM master has to be running when Postgres-XC cluster is running.

26.1 Removing GTM slave

You should stop GTM slave before removing. `pgxc_ctl` provides command to do this:

```
PGXC$ stop gtm slave
```

Then, you can remove GTM slave by:

```
PGXC$ remove gtm slave
```

To remove gtm slave, `pgxc_ctl` does the following:

1. Update pgxc_ctl configuration file and back up it if specified.

26.2 Removing GTM proxy

Before you remove a gtm proxy, you should stop it. `pgxc_ctl` provides a command to do as follows:

```
PGXC$ stop gtm_proxy name
```

where *name* is gtm_proxy name to stop.

Then, you can remove the gtm_proxy as follows:

```
PGXC$ remove gtm_proxy name
```

Please note that you should configure coordinators and datanodes connecting to this gtm proxy and restart them. It is advised that you can remove a gtm proxy if no coordinators or datanodes are connected to it any longer.

26.3 Removing coordinator master

Because a coordinator does not store user data, it is not harmful to remove a coordinator master. Please do not issue DDL while you are removing coordinator master, or such DDL could be propagated to the removing coordinator.

`pgxc_ctl` does not care if the removing coordinator master is running. If it is running, `pgxc_ctl` will stop it.

The command to remove a coordinator master is as follows:

```
PGXC$ remove coordinator master name
```

where *name* is the coordinator node name to remove.

`pgxc_ctl` will do the following to remove a coordinator master.

1. Remove the slave of the removing coordinator master if configured. See the next section for details.
2. Issue `DROP NODE` statement at all the other coordinator to remove the coordinator from all the other coordinators.
3. Stop the coordinator master if running.
4. Update pgxc_ctl configuration file and back up it if specified.

26.4 Removing a coordinator slave

You can remove a coordinator slave by following `pgxc_ctl` command.

```
PGXC$ remove coordinator slave name
```

where *name* is the coordinator name to remove.

`pgxc_ctl` will do the following to remove a coordinator slave.

1. If the coordinator slave is running, stop it.
2. Update the master's configuration to disable log shipping.
3. Restart the master.
4. Update pgxc_ctl configuration file and back up it if specified.

26.5 Removing a datanode master

Before you remove a datanode master, please be sure that the removing datanode does not contain any user data. You can check this by using `\d+` *pattern* command to `psql`. Issue `ALTER TABLE` statement to each table to remove the datanode from its replication or distribution nodes.

You can remove a datanode master with the command:

```
PGXC$ remove datanode master name
```

where *name* is the datanode node name to remove.

`pgxc_ctl` will do the following to remove a datanode master.

1. If slave is configured for this mater, remove it. See the next section for details.
2. Issue `DROP NODE` statement in all the coordinators to remove this datanode.
3. Update pgxc_ctl configuration file and back up it if specified.

26.6 Removing a datanode slave

Removing a datanode slave is quite similar to removing a coordinator slave. You can do this by the following `pgxc_ctl` command:

```
PGXC$ remove datanode slave name
```

where *name* is the datanode name to remove.

`pgxc_ctl` will do the following to remove a datanode slave.

1. If the coordinator slave is running, stop it.
2. Update the master's configuration to disable log shipping.
3. Restart the master.
4. Update pgxc_ctl configuration file and back up it if specified.