Postgres-XC Architecture, Implementation and Evaluation Version 0.900

NTT Open Source Software Center
EnterpriseDB Corporation

Mar. 25^{th}, 2010

Contents

1 What Is Postgres-XC? 6

2 Postgres-XC's Goal 6

3 How To Scale Out Both Reads And Writes? 8

3.1 Parallelism In Postgres-XC 8

3.2 Postgres-XC's Global Transaction Management 10

4 Postgres-XC Key Components 10

4.1 GTM (Global Transaction Manager) 10

4.1.1 How PostgreSQL Manages Transactions 10

4.1.2 Making Transaction Management Global 12

4.2 Coordinator 14

4.3 Data Node 14

4.4 Interaction Between Key Components 16

5 Isn't GTM A Performance Bottleneck? 16

5.1 Primitive GTM Implementation 18

5.2 GTM Proxy Implementation 19

5.3 Coordinator And Data Node Connection 21

6 Performance And Stability 21

6.1 DBT-1-Based Benchmark 21

6.2 Test Environment 24

7 Test Result 24

7.1 Throughput And Scalability 25

7.2 CPU Usage 27

7.3 Network Workload 28

8 Remarks Of Current Implementation **29**

8.1 Development Status 29

8.2 Development History & Approach 29

8.3 Limitations 30

8.4 Connection Handling 30

8.5 The Postgres-XC Code 31

8.6 Noteworthy Changes to Existing PostgreSQL Code 31

9 Roadmap of Postgres-XC **32**

Revision Log

Mar. 25, 2010, Version 0.900 Initial release

1 What Is Postgres-XC?

Postgres-XC is an open source project to provide write-scalable, synchronous multi-master, transparent PostgreSQL cluster solution. It is a collection if tightly coupled database components which can be installed in more than one hardware or virtual machines.

Write-scalable means Postgres-XC can be configured with as many database servers as you want and handle much more writes (updating SQL statements) which single database server can not do. 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 stored in more than one database servers internally[1].

You can configure Postgres-XC to run on multiple hardware. They store your data in a distributed way, that is, partitioned or replicated way at your choice for each table. [2] When you issue queries, Postgres-XC determines where the target data is stored and issue corresponding queries to servers with the target data as shown in Figure 1.

In typical web systems, you can have as many web servers or application servers to handle your transactions. However, you cannot do this for a database server in general because all the changing data 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.

Postgres-XC is based upon PostgreSQL database system and reuses most of existing modules including interface to applications, parser, rewriter and executor. In this way, Postgres-XC's application interface is compatible to existing PostgreSQL. (As described later, at present, we provide limited SQL statement, which will be improved in the future).

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:

1. Postgres-XC should provide multiple servers to accept transactions and statements from applications, which is know as "master" server in general. In Postgres-XC, this

[1] Of course, you should use how tables are stored internally when you design the database physically to get most from Postgres-XC.

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

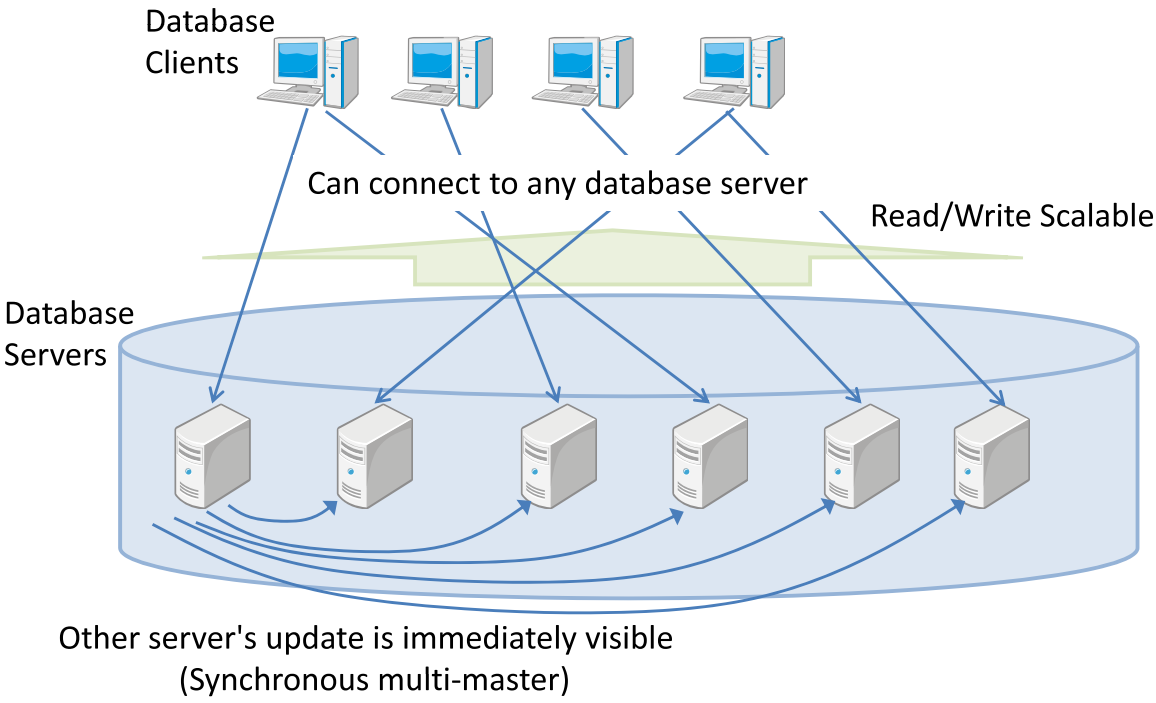

Figure 1: Postgres-XC stores your data in a distributed fashion

is called "coordinator".

2. Postgres-XC should provide more than one masters.
3. Any "master" 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.
4. Tables should be able to be stored in the database in replicated or distributed way (know as fragment 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.
5. Postgres-XC provide compatible PostgreSQL API to applications.
6. 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 achievements are as follows:

1. Transaction management is almost complete. Postgres-XC provides complete "Read Committed" and "Serializable" transaction isolation level which behaves exactly the

same as single PostgreSQL server. Savepoint and two-phase commit from the client should be added in the future.

2. Simple SQL statements are available, which do not need cross-node operation such as cross-node joins. Details will be described later.

3. Views, subqueries, rules, aggregate functions and and cross-node joins are not available.

We're planning to extend the SQL statement support toward general PostgreSQL SQL statements, including views, subqueries, rules, stored functions and triggers, as well as supporting dynamic reconfiguration.

3 How To Scale Out Both Reads And Writes?

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

We'll compare Postgres-XC'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.

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 "data node". 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 data nodes are involved in handling each statement.

In this way, as seen in Figure 2, statements are handled in parallel by Postgres-XC servers, which scales transaction throughput. As reported 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-2 shows that present Postgres-XC is suitable for transactional use case as described in PostgreSQL Wiki page. By improving supported SQL statements, we're expecting that Postgres-XC can be suitable for analytic use case.

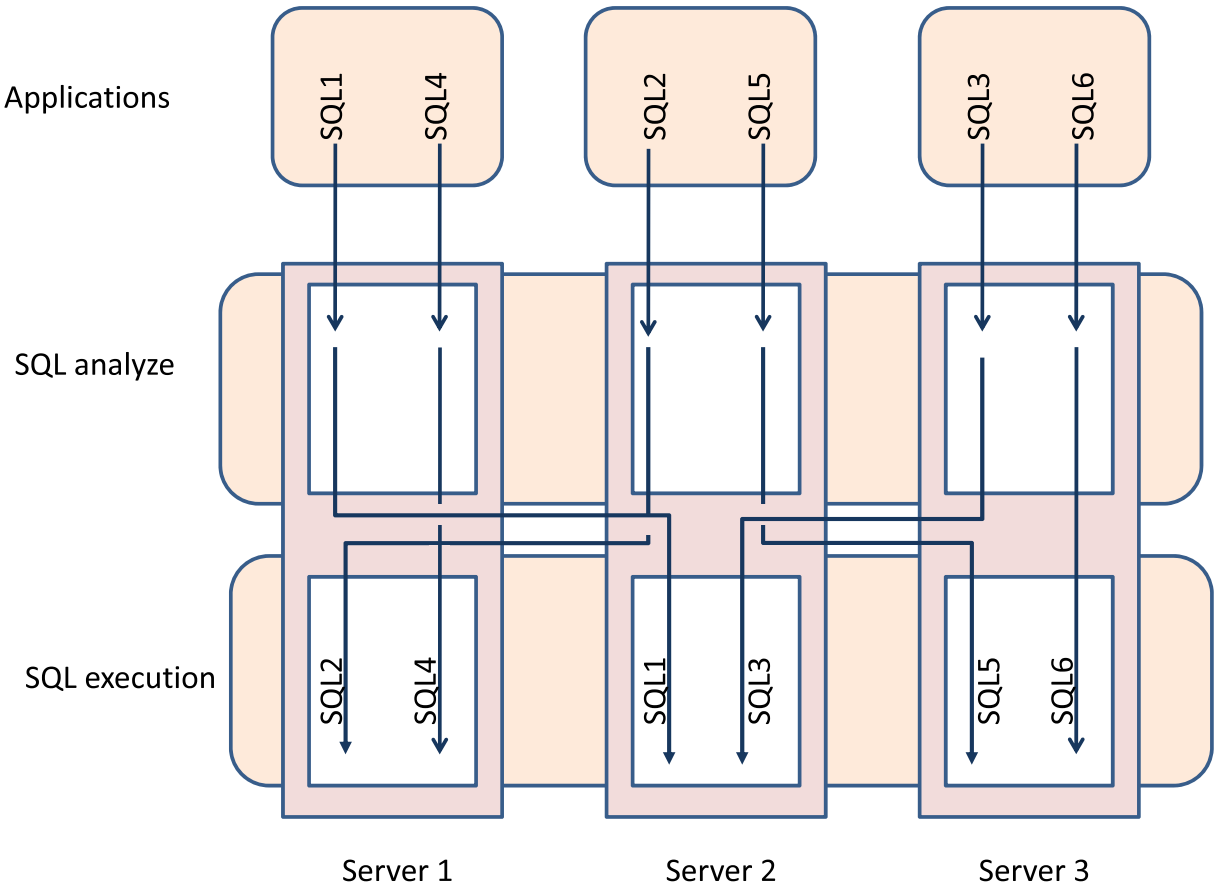

Figure 2: Postgres-XC can handle statements in parallel in multiple data nodes.

3.2 Postgres-XC's Global Transaction Management

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.

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 data node.

Details of the background and the algorithm will be given in later sections.

4 Postgres-XC Key Components

In this section, we will show main components of Postgres-XC.

Postgres-XC is composed of three major components, called GTM (Global Transaction Manager), Coordinator and Data Node as shown in Figure 3. Their features are given in the following sections.

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 updating data.

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 distinguish which transaction is older/newer[3]. Please let us describe a little in detail how it is done. [4]

[3] More precisely, XID is 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.

[4] 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.

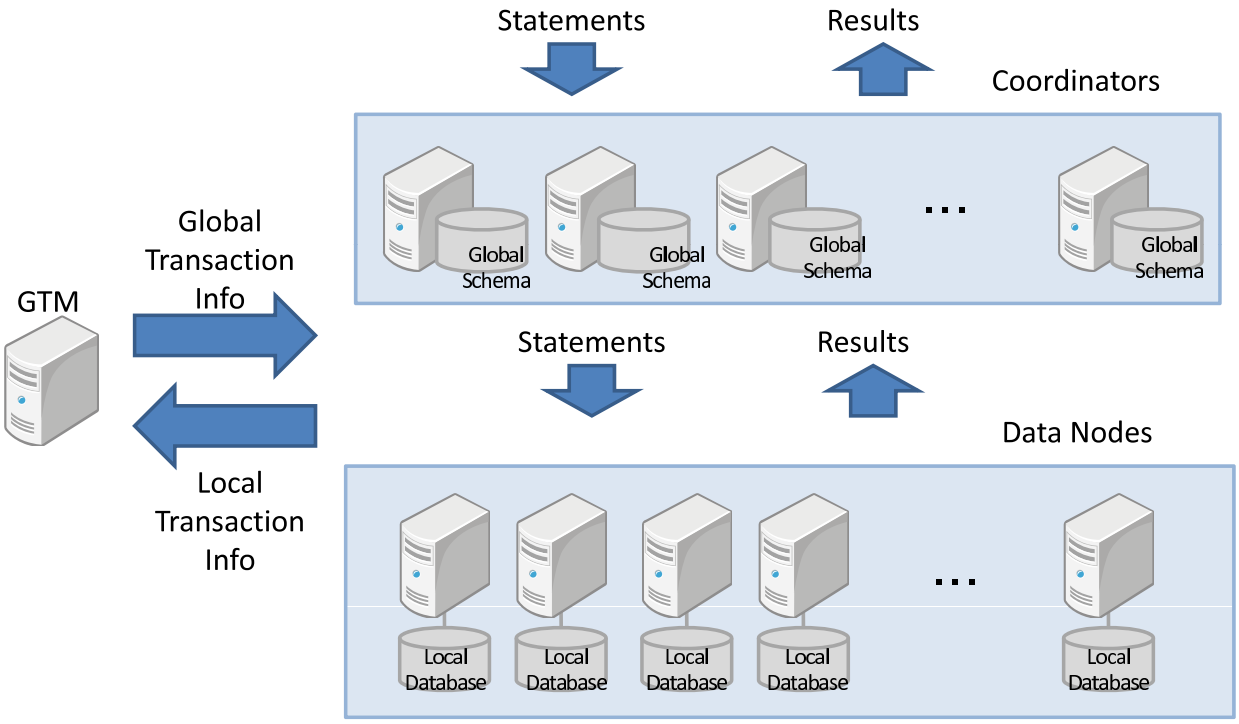

Figure 3: Interaction between Postgres-XC components

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 the transaction should ignore such tuple. In such way (in practice, this is done by versup 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 on the other server.

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" the 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, which is called the "*snapshot*".

If the creating transaction 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 running*" and "*if an old transaction was committed or aborted*."

The former information is 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.

4.1.2 Making Transaction Management Global

In Postgres-XC, we picked the following features of transaction management and visibility checking:

1. Assigning XID globally to transactions (GXID, Global Transaction ID). This can be done globally to identify each Transactions in the system.
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 initiated by other server in Figure 1 or Figure 2. 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 take place.

To do this, Postgres-XC introduced a dedicated component called GTM (Global Transaction Manager). GTM runs on one of the servers and provide unique and ordered transaction id to each transaction running on Postgres-XC servers. Because this is globally unique ID, we call this GXID (Global Transaction 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 database consistency. Outline of this mechanism is illustrated in Figure 4. Please note how transactions included in each snapshot changes according to global transaction.

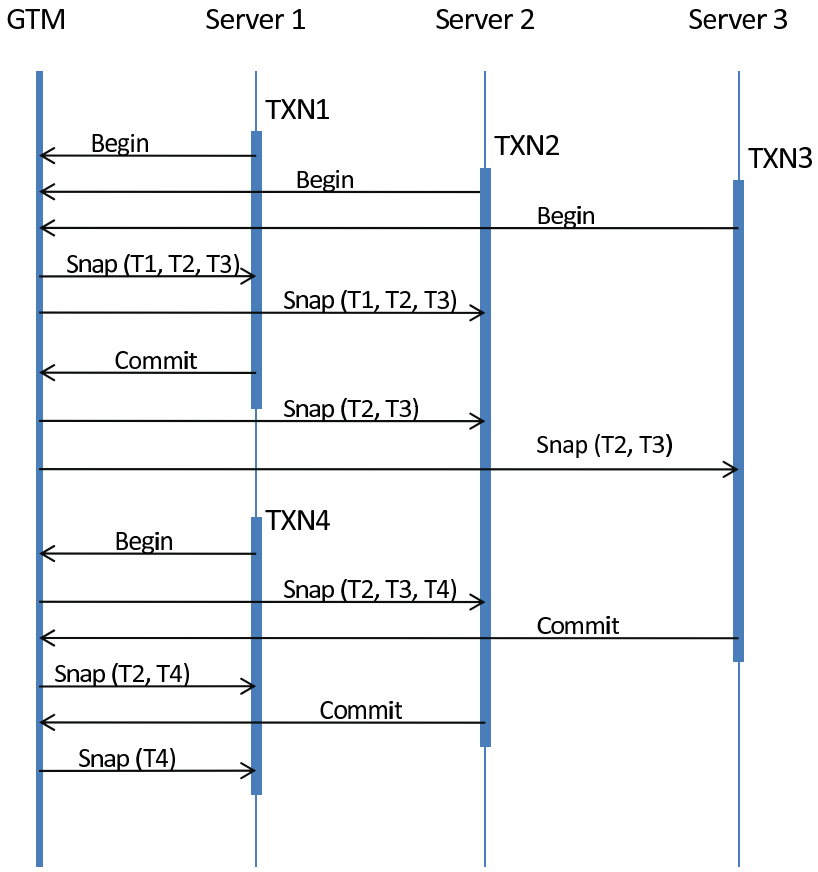

Figure 4: Outline of Postgres-XC's Global Transaction Management.

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 command in two-phase commit protocol.

Please also note that global snapshot provided by GTM includes other transactions running on different servers and reflects transaction status reports.

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 statement. If it is "serializable", transaction will request a snapshot at the beginning of transaction and reuse it thought the transaction.

GTM also provides global value such as sequence. Such global value will include timestamps, notification and so on. This will be an extension in the following releases.

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 Data Node as described below. Coordinator receives SQL statements, get Global Transaction Id and Global Snapshot as needed, determine which data node is involved and ask them to execute (a part of) statement. When issuing statement to Data Nodes, it is associated with GXID and Global Snapshot so that Data Node is not confused if it receives another statement from another transaction originated by another coordinator.

4.3 Data Node

Data Node actually stores your data. Tables may be distributed among data nodes, or replicated to all the data nodes. Because Data Node does not have global view of the whole database, it just takes care of locally stored data. Incoming statement is examined by the coordinator as described next, and rebuilt to execute at each data node involved. It is then transferred to each data nodes involved together with GXID and Global Snapshot as needed. Data Node may receive request from various coordinators. However, because each the transaction is identified uniquely and associated with consistent (global) snapshot, data node 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.5.

A coordinator receives statements from applications. When it starts new transaction, it send a request to the GTM to get new global transaction ID. GTM keeps track of such request to calculate a global snapshot. According to the transaction isolation level, it also

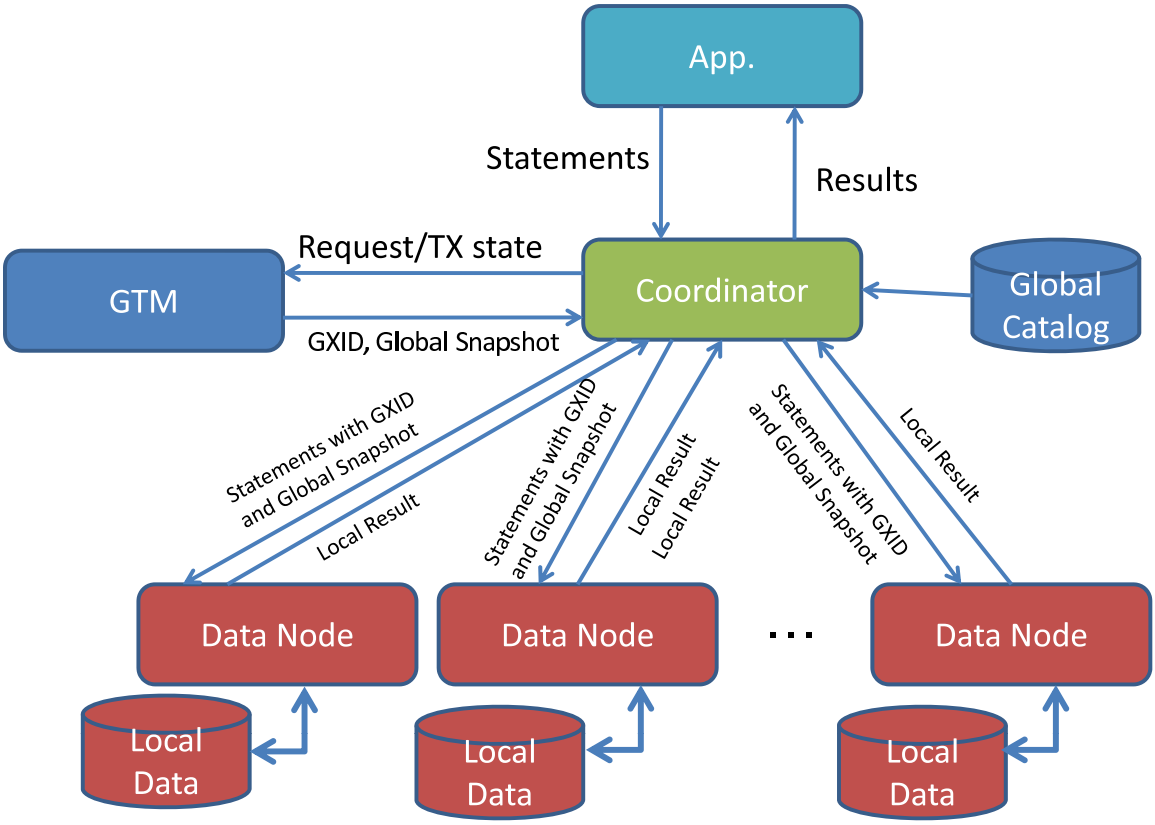

Figure 5: Interaction between Postgres-XC components

requests GTM for the global snapshot. Then the coordinator analyzes a statement and determines which data not is involved[5], rewrite the statement as needed and send it to the data node involved, with GXID and global snapshot. The involved data node may change from statement to statement. Detailed sequence will be shown in the next section.

4.4 Interaction Between Key Components

As explained in the previous section, Postgres-XC has three major components to provide global transaction control, to determine which data node should each statement go and to handle the statement.

Sequence of interactions among Postgres-XC components are given in Figure 6.

As shown in the figure, when a coordinator begins a new transaction, it inquires 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 `SERIALIZABLE`, 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 data node to go, and converted for each data node if necessary.

Please note that statements will be passed to appropriate data nodes 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 data nodes are involved in the update in the transaction, the coordinator issues `PREPARE` for 2PC, then issue `COMMIT`. These steps will be reported to GTM as well to keeps track of each transaction status to calculate global snapshots.

Please see the section 4.1 for details of this background.

5 Isn't GTM A Performance Bottleneck?

Because GTM can be regarded as "serializing" all the transaction processing, people may think that GTM can 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 7.3. Latency to send

[5] At present, the coordinator accepts only statements just one data node is involved. For details, please see later sections.

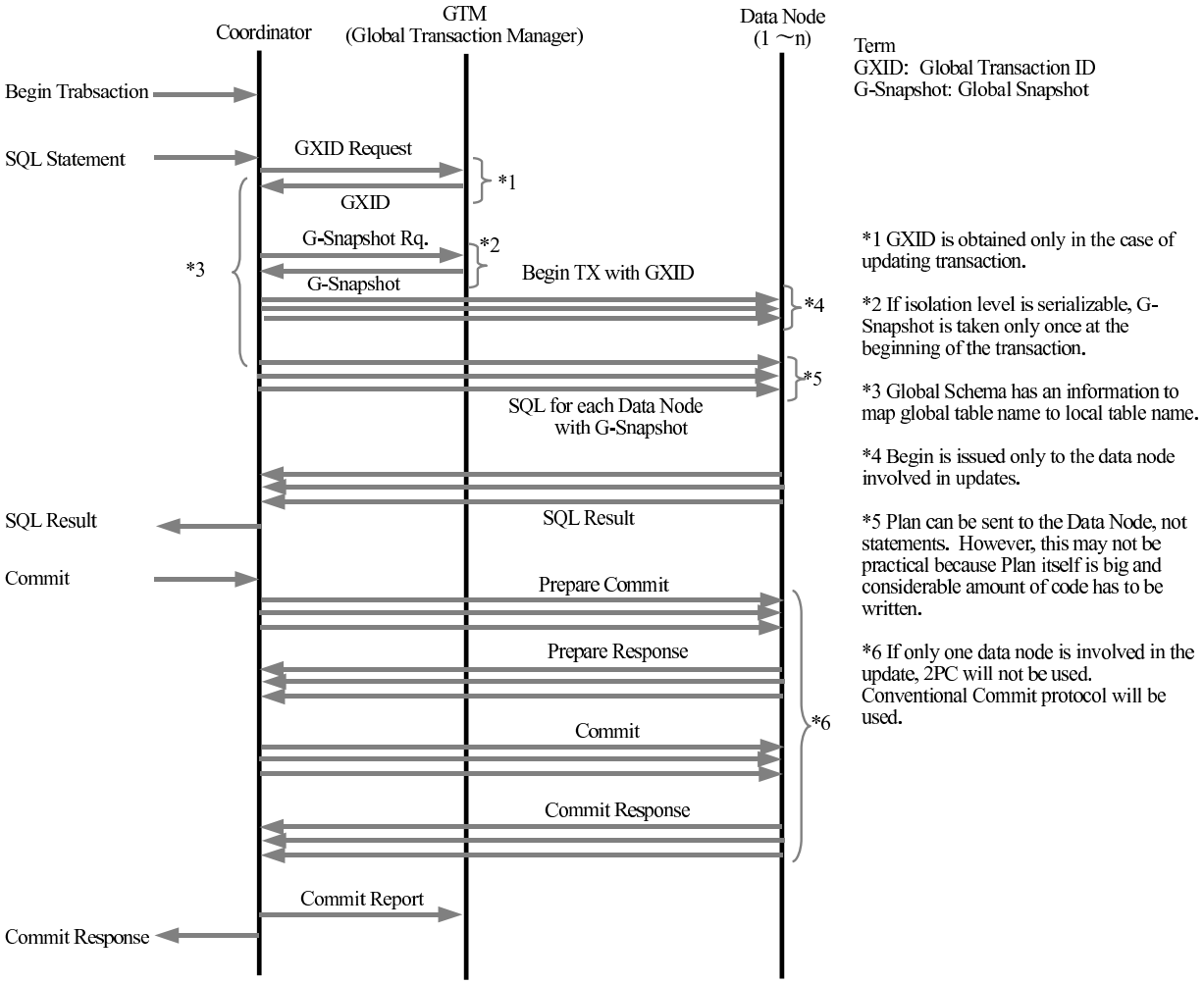

Figure 6: Interaction between Postgres-XC components

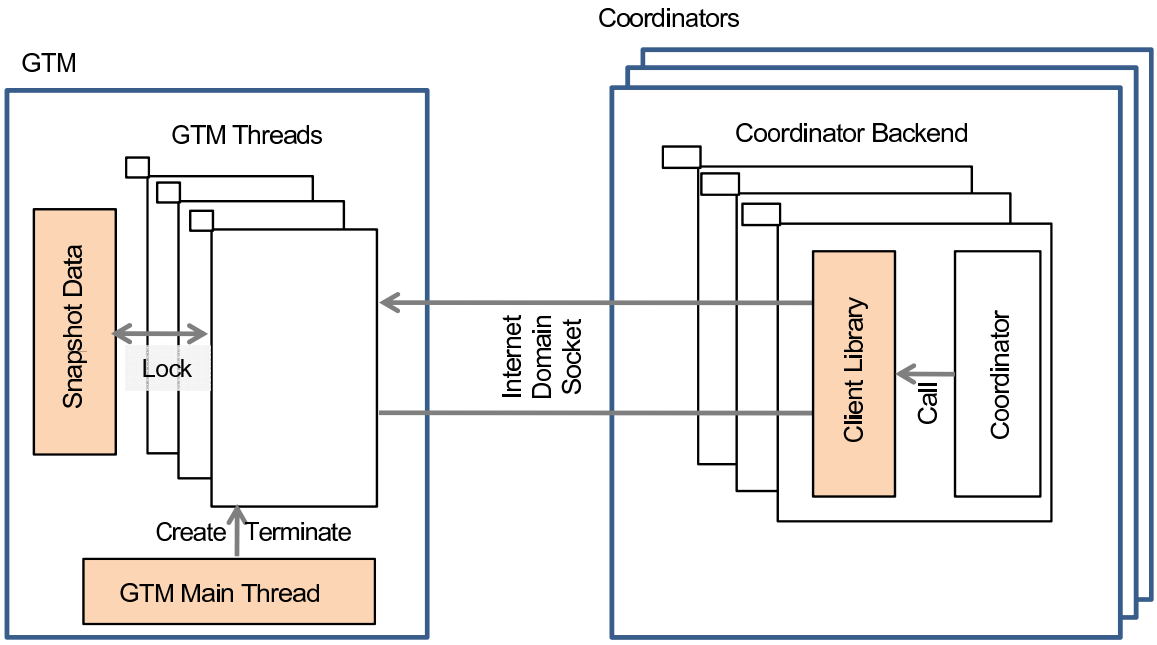

Figure 7: Primitive GTM structure

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 data nodes. Typical configuration will be shown in Figure 10.

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

5.1 Primitive GTM Implementation

As seen in Figure 6, GTM can be implemented as shown in Figure 7. 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 sends GXID, snapshot and other response to the coordinator backend.

4. It 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, 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[6]. In fact, the order of the amount of data sent from GTM is ($O(N^2)$ where N is 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.

Real network workload will be shown in later section to show 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.

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 8,

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 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 like described in Section 5.1. Now GTM can accept connections from GTM Proxies.

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

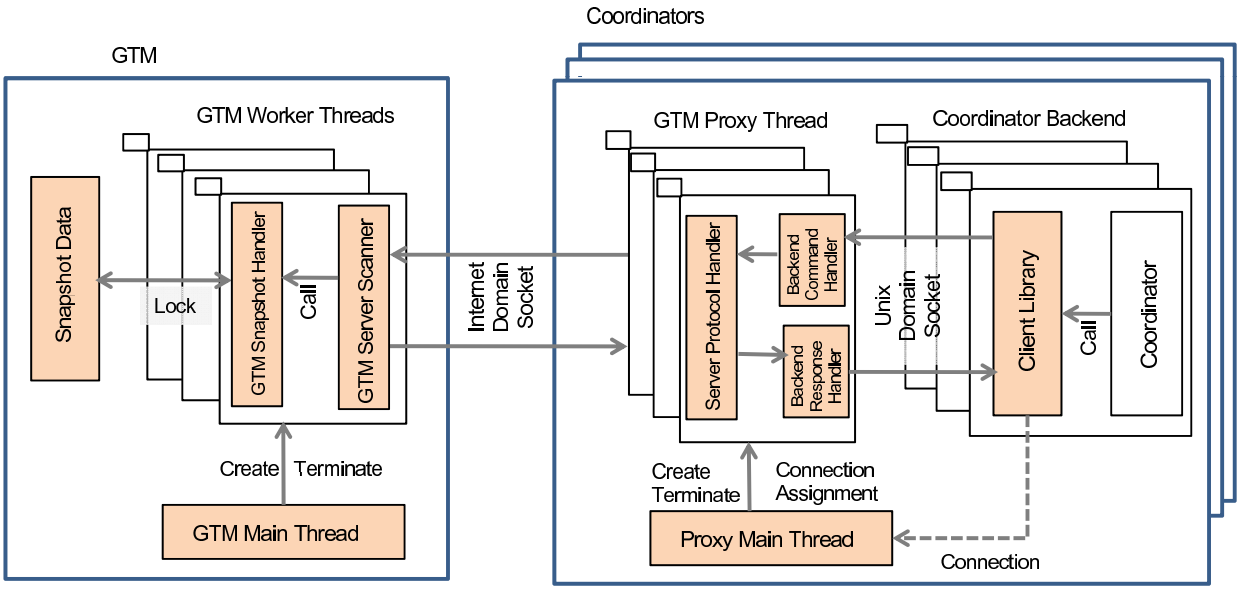

Figure 8: GTM structure with proxy

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 handles multiple coordinator backends. If a coordinator has one hundred coordinator backends and one GTM Proxy Thread, this thread takes care of one hundred coordinator backend.

Then GTM Proxy Thread scans[7] all the requests from coordinator backend. If coordinator is more busy, 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

[7] Poll(2) will be available, for example.

structure. When the workload to Postgres-XC is quite light, the interaction will be as same as the case in Section 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 $O(N)$.

5.3 Coordinator And Data Node Connection

As seen in Figure 5, you may think that the number of connection between coordinator and data node may increase from time to time. This may leave unused connection and waste system resources. Repeating real connect and disconnect requires data node backend initialization which increases latency and also wastes system resources.

For example, as in the case of GTM, if each coordinator has one hundred connections to applications and we have ten coordinators, after a while, each coordinator may have connection to each data node. It means that each coordinator backend has ten connections to coordinators and each coordinator has one thousand (100×10) connections to coordinators.

Because we consume much more resources for locks and other control information per backend and only a few of such connection is active at a given time, it is not a good idea to hold such unused connection between coordinator and data node.

To improve this, Postgres-XC is equipped with connection pooler between coordinator and data node. When a coordinator backend requires connection to a data node, the pooler looks for appropriate connection from the pool. If there's an available one, the pooler assigns it to the coordinator backend. When the connection is no longer needed, the coordinator backend returns the connection to the pooler. Pooler does not disconnect the connection. It keeps the connection to the pool for later reuse, keeping data node backend running.

6 Performance And Stability

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 benchmark available in public for transactional use case.
- Tables are designed so that it cannot simply be partitioned. We need more than one partitioning.

At present, Postgres-XC does not support cross-node joins [8] and aggregate functions. It does not support views, rules or subqueries either. It is not allowed to update the key value

[8] Cross node joins are joins which needs material from more than one data node to calculate.

used to determine the location of rows of tables. Although it is possible to use semantic dependency among keys to determine smarter table distribution, it is not available yet and we should do this manually[9][10].

We will show how we modified DBT-1 to best tune to Postgres-XC.

To localize statement target, we modified DBT-1 tables as follows[11].

1. Customer-ID is added to order-line table. Because Customer ID is the external key of order table to customer table and order ID is the external key of order line table to order table, it is obvious that order line table refers to the customer table through order ID. We expect this addition can be generated automatically be adding key dependence description in DDL.
2. Customer-ID is added to address table because in reality it is obvious that personal information belongs to each customer and it is common practice not to share such information among different customers.
3. table is divided into two tables, item and stock, as in the latest TPC-W specification.

Also, we changed connection from ODBC to libpq.

Table configuration of DBT-1, is illustrated in Figure 9 with modification in Postgres-XC evaluation. 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 the data nodes.

Please note shopping cart and shopping cart ID tables. It is more favorable if shopping cart and shopping card ID can be distributed using customer ID. However, because DBT-1 uses these table even before a customer is not assigned customer ID, 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.

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

[9] Present Postgres-XC is focused on features which can demonstrate outcome of Postgres-XC in most effective way and these remarks in database design should be noticed in the future extension as well.

[10] Current implementation remarks will be summarized in later sections.

[11] For details of DBT-1 table configuration, please refer to DBT-1 material.

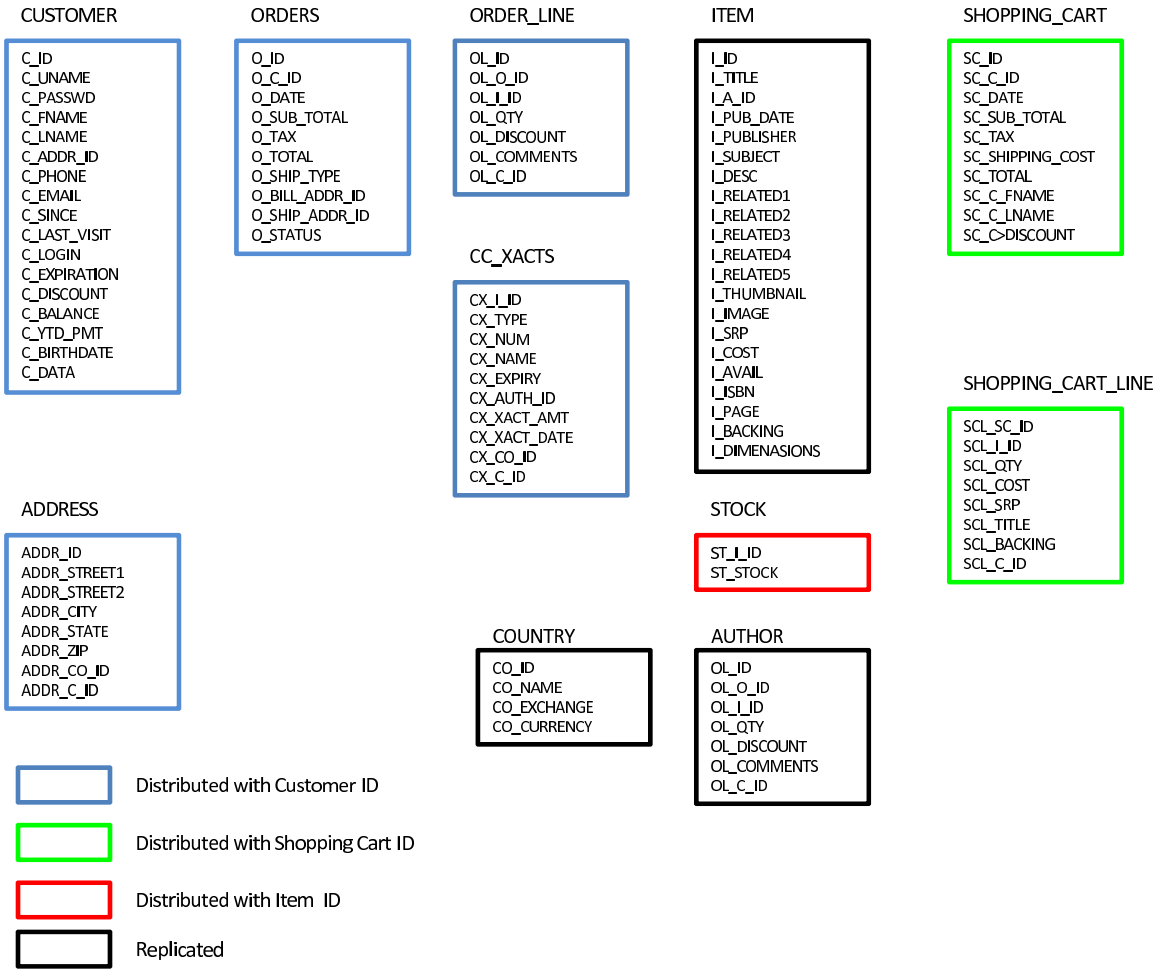

Figure 9: DBT-1 Based Tables Used in the Evaluation.

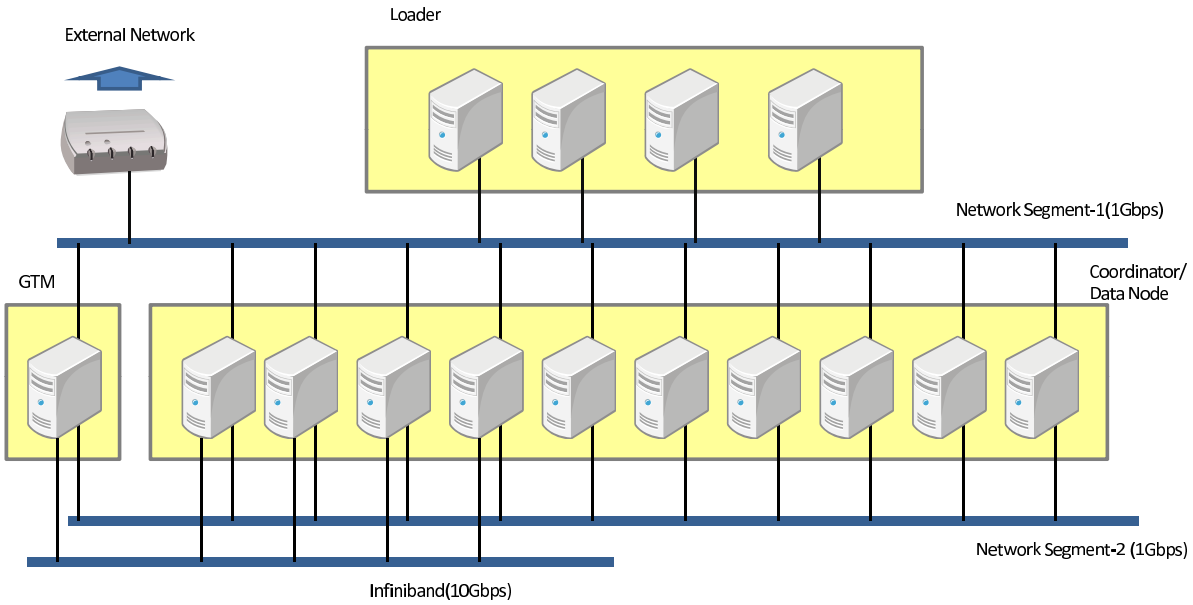

Figure 10: Postgres-XC Test Environment

6.2 Test Environment

Test environment is shown in Figure 10. We have one GTM, and up to ten database servers. Each database server is equipped with one coordinator and one data node. Although we can install coordinator and data node in separate servers, we used this configuration because it is simpler to balance the workload of coordinator and data node.

We have four servers 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. (As shown later, Infiniband is not necessary in our case).

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.
2. Postgres-XC with one server.

Table 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

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 data node were installed in each server. GTM was installed in a separate server.

We ran the test with two kinds of workload as follows:

1. Full load. Measured throughput and resource usage 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 usage and network workload of the benchmark.

7.1 Throughput And Scalability

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

Table 1 shows the result of full load throughput for various configurations. Figure 11 is the chart of Postgres-XC scale factor vs. number of servers, based on the result in Table 1.

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

We also run Postgres-XC with five coordinators/data nodes for a week with 90% workload of the full load with five coordinator/data nodes. In this period, GTM, coordinators and

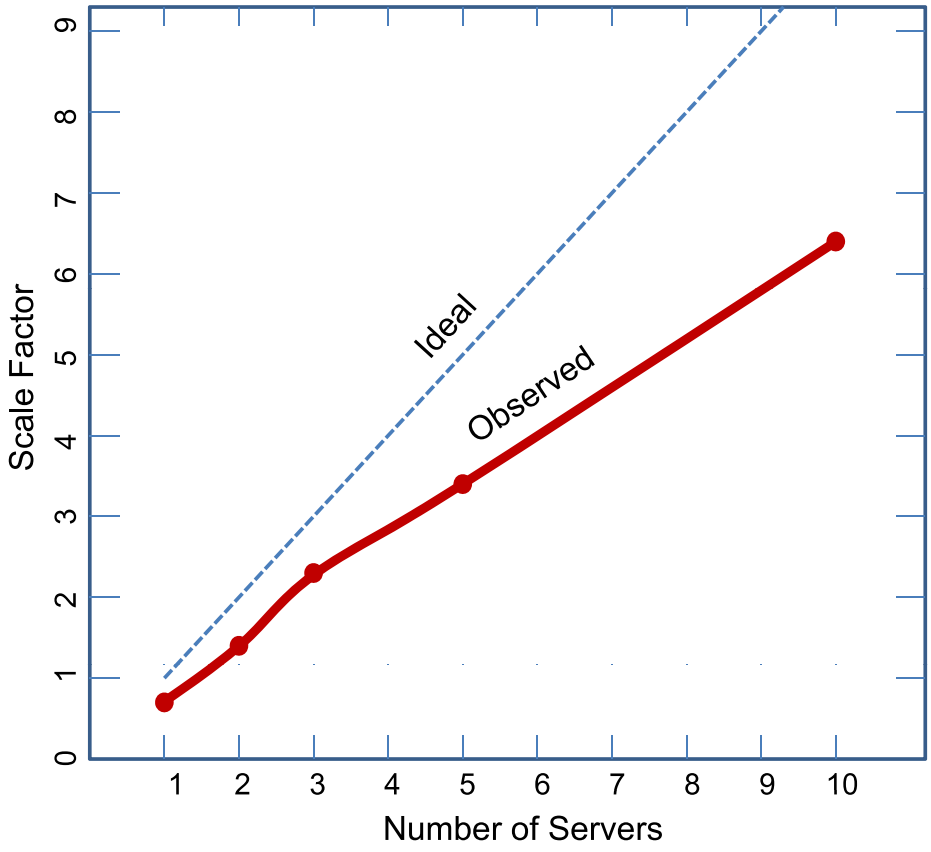

Figure 11: Postgres-XC Full Load Throughput

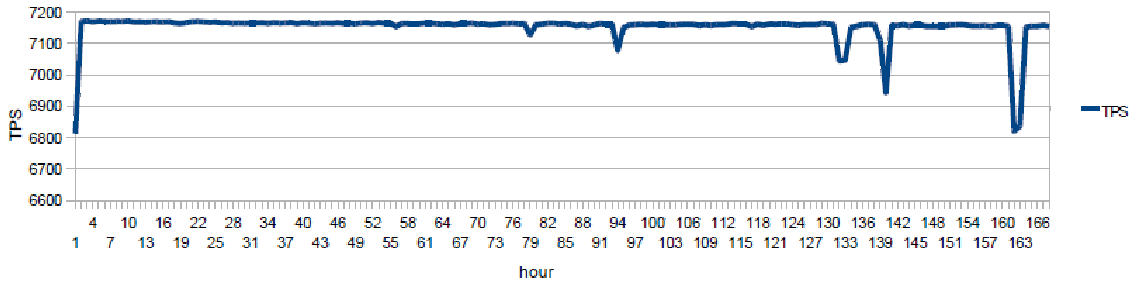

Figure 12: Postgres-XC Full Load Throughput in One Week Test

Table 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/Data Node

** 2 loaders were used.

*** Indicates number of Coordinator/Data Node and loader respectively.

data nodes handled GXID wraparound and vacuum freeze successfully[12].

Figure 12 shows the throughput chart. At average, the throughput is quite stable, except that spikes are observed periodically and spike grows with time.

We observed that vacuum analyze is running for a long period at the spike. We're analyzing this and trying to find workarounds. We think GTM has nothing to do with the spike and GTM runs stably in long period.

7.2 CPU Usage

We've measured CPU usage in the benchmark test above to see if Postgres-XC reasonably uses hardware resource. Table 2 shows CPU usage (100% − idle) for various configuration and nodes with full workload.

[12]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`

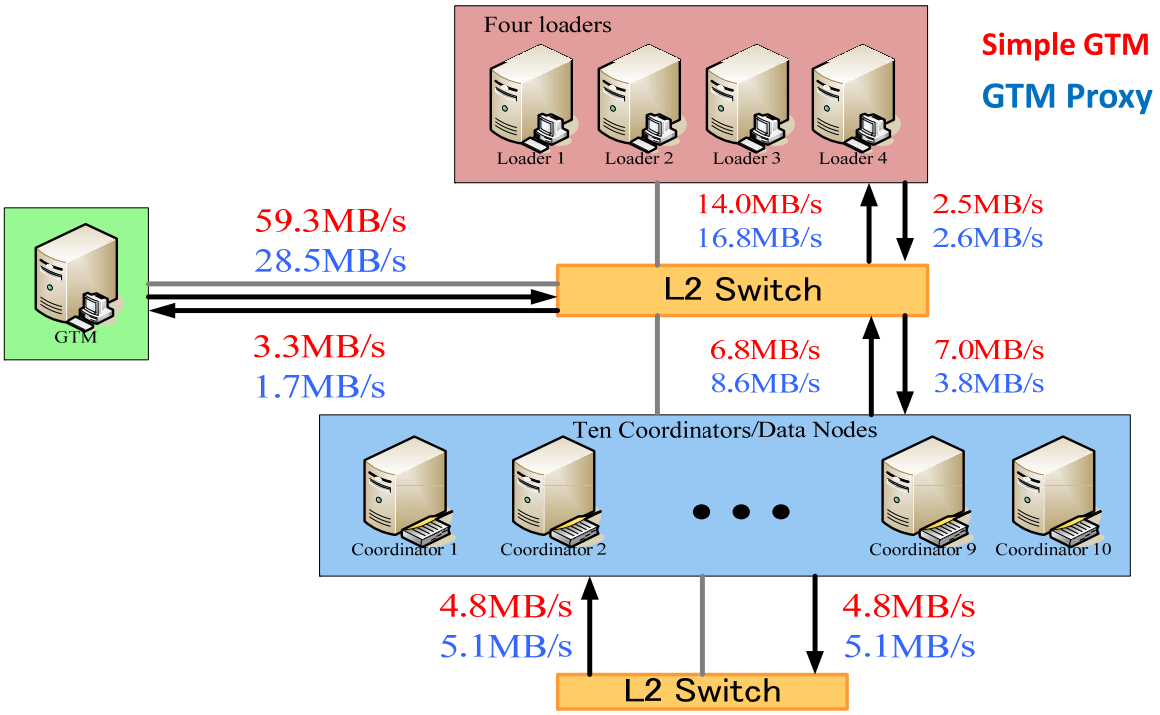

Figure 13: Network Data Transfer Rate of Each Server

7.3 Network Workload

We have also measured the network workload.

Figure 13 summarizes the data transfer rate between server. Please note the measurement in this figure is network read/write rate for each servers.

This is also summarized in Table 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.
2. Other server's network workload is very light. Conventional Gigabit network is sufficient.

Because current Postgres-XC doesn't support cross node join, no cross-node tuple transfer is involved in this measurement. In the future, we should make similar measurement for cross-node joins to estimate additional network workload.

Table 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/Data Node ⇔ Loader/GTM	7.0MB/s	3.8MB/s	6.8MB/s	8.6MB/s
Coordinator/Data Node ⇔ Coordinator/Data Node	4.8MB/s	5.1MB/s	4.8MB/s	4.8MB/s

8 Remarks Of Current Implementation

This section contains information regarding the status of the project and some general information regarding development of PG-XC.

8.1 Development Status

Postgres-XC is in the early stages of development, but has produced useful results for demonstrating scalability. The SQL commands currently supported is limited, but in the coming months we will work on expanding on that. The focus will be on increasing the usability and features of the cluster.

8.2 Development History & Approach

A goal at the beginning of the project was write scalability for transactions. User-created tables may be distributed across multiple nodes (or replicated), to spread the load. In accessing these tables, we needed to guarantee that the data across nodes can be accessed with a consistent view of the data.

Therefore a decision was made to focus on transaction management initially. If we could not achieve good performance in globally supporting MVCC[13], the project could not get very far.

We first developed a Global Transaction Manager (GTM) that is responsible for assigning transaction ids and snapshots. This worked well for hundreds of connections, but for greater scalability, we also developed a GTM Proxy. Basically, it collects GTM requests from coordinator processes and groups them together more efficiently. Network messaging is improved, the GTM need not handle as many connections, and time spent in critical sections in the GTM is reduced.

[13]MVCC (Multi-Version Concurrency Control) is a center of PostgreSQL's transaction management. For details, please refer to PostgreSQL reference manual.

We modified the other components of the cluster to make use of the new GTM. Clients connect to a coordinator. The coordinator communicates with GTM for transaction id and snapshot information, and the coordinator passes it down as needed to the data node connections.

All of this adds some latency for any given transaction, but we are interested in achieving good total throughput for the entire cluster handling many sessions.

As of this writing in March 2010, we will be improving the supported SQL and PostgreSQL features. We currently block some statements that we determine to be unsafe to execute in the cluster or are not supported, like creating a foreign key constraint that cannot be enforced locally on a single node. In the near term we may loosen restrictions on some features to improve usability, but the DBA needs to understand the danger and be cognizant of how tables are distributed (example: stored functions).

8.3 Limitations

We only execute SQL statements that can be safely executed in the current cluster. For example, we exclude `SELECT` statements where data from node must be joined with data on another. We also do not yet support multi-node aggregates or multi-node `DISTINCT`.

8.4 Connection Handling

We want to avoid unnecessarily involving data nodes in a transaction that do not need to be involved. That is, we do not simply obtain a connection to every data node 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 data node involved.

Assume in our example, only 2 data nodes were involved, one for the first `UPDATE` statement, and one for the second `UPDATE`. 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 data node 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.

8.5 The Postgres-XC Code

This section discusses the Postgres-XC code. Details about compiling and installing Postgres-XC can be found in PostgreSQL reference manual.

The current Postgres-XC code is applied to PostgreSQL 8.4.2. To make it easier to view the changes, most of the changes to existing PostgreSQL modules are easily identifiable by blocks of code that use "`#ifdef PGX`". (The original idea was to try and make the code easy to remerge with PostgreSQL as its development continues and have the changes easy to identify.) As a result, before executing "configure" when building the code, please set `CFLAGS=-DPGXC`.

Note that the same PostgreSQL executable is used for both the coordinator and data node components. They are just started differently. You will typically see some code comments indicating whether the particular block of code was added for the coordinator side or data node side. Having these components use the same source modules makes development and maintenance easier.

8.6 Noteworthy Changes to Existing PostgreSQL Code

GTM, transaction handling, and related changes can be found in code relating to transaction ids and snapshots (`varsup.c`, `xact.c`, `procarray.c`, `postgres.c`). Sequences live on and are assigned from the GTM, like transaction ids and snapshots. In contrast to stand-alone PostgreSQL, in Postgres-XC, every node will not necessarily be involved in every transaction. Some existing logic in clog.c had to therefore be modified to detect if we need to extend clog. Similar code appears in `subtrans.c`.

We also added special handling for transaction id wrap-around. There is a new `pg_catalog` table, `pgxc_class` that stores table distribution information. We considered modifying `pg_class`, but decided to try and keep the existing catalog tables as-is and create new ones instead, thinking this may make remerging code with future versions of PostgreSQL easier. We will likely update the structure of the table and possibly break this out into multiple tables, but it is sufficient for now.

There is special handling for auto-vacuum. We want to exclude those transaction ids from other snapshots, so we notify GTM specially when requesting a transaction id for auto-vacuum. The data nodes typically receive transaction ids and snapshots from the coordinator. Auto-vacuum worker threads, however, connect directly to GTM.

A possible future optimization is to have autovacuum worker threads that are performing an analyze have their transaction ids appear only in snapshots of statements executing on the same node (we need not worry about that transaction id appearing on other nodes).

We avoid writing to WAL on the coordinator if it was not involved in a transaction. If it is involved, such as for DDL, we do write to WAL. At initdb time, we just use local transaction ids and do not involve GTM.

9 Roadmap of Postgres-XC

Because current Postgres-XC supports fundamental transaction management, tuple visibility, connection pooling infrastructure and the outcome is very reasonable, future effort should be payed to coordinator extension and utilities.

Coordinator extension with higher priority includes the following:

Backup and Restore We will begin with `COPY TO` feature and then, `pg_dump` and `pg_restore`. We will extend `pg_restore` to allow inputs from multiple coordinators.

Then, we will implement physical backup and restore using base backup and archive log. As you may notice, we cannot simply do this because we need another means to *synchronize* the recovery point among data nodes and coordinators. We have already done preliminary research how GTM can help to synchronize recovery point. This will be disclosed elsewhere.

Statement Extension First of all, we'd like to extend supported statements by analyzing them more precisely. We may use PostgreSQL's planner code[14].

Then we'll begin to support cross-node join. Cross-node join is very complicated feature, especially global planning/optimizing is the key. We're planning to begin with basic execution with general tuple transfer among data nodes and coordinators.

Although most of the statements can be executed at data nodes, most general execution is to collect every material to the coordinator (this could be very slow, though). Efficient tuple transfer is common infrastructure both for cross-node join and cross-node aggregation.

In addition, more logs should be available to keep track how each statement was handled within the coordinator.

Prepared Statement This is especially useful at batch processing.

Stored functions As you may imagine, we need three kinds of stored function, (1) one runs only on coordinators, (2) one runs only on data nodes and (3) one can run both on coordinator and data node.

When defining functions, you may have to specify where it can run and the coordinator uses this information when it rewrites input statement for each data node, as well as additional plan inside the coordinator.

[14] So far, Postgres-XC use PostgreSQL's parser and rewriter, but not planner.

DDLs Many DDLs can be handled simply by forwarding them to all the coordinator and data nodes. `CREATE TYPE` is a typical example. We ill make these DDLs safe to run in Postgres-XC.

Other complicated DDLs, such as `CREATE VIEW` or `CREATE RULE` will be implemented afterwords. We need preliminary research and design work for the Rule.

Order By and Distinct They're one of the most popular clause. We're planning to implement them with highest priority.

Aggregate Functions Handling aggregate functions is rather complicated and we're planning to support this in several steps. First, we will support simple usage which appears at the top level of `SELECT` clause without `GROUP BY` clause. Then we will support `GROUP BY` clause, as well as aggregates in expression. User-defined aggregate function will be the next issue.

Session Parameter Current implementation does not manage session parameters. Managing session parameters in the connection pooling will be addressed in the near future.

Tuple Transfer Infrastructure For more general support of cross-node join and aggregates (and `ORDER BY` possibly), a data node needs tuples stored in other data node. Depending upon the plan, a coordinator may have to collect all such information to execute query at the coordinator. For this purpose, we're planning to do research and investigation work. Issues are (1) how to collect tuples at remote data nodes using the context of current transaction, (2) avoid writing to WAL and (3) use existing PostgreSQL module as much as possible.

Implementation will be done afterwords.

Savepoint GTM needs an extension for this. Implementation is straightforward and is an issue of resource assigning.

External 2PC This is to allow applications to issue 2PC command as statements (Postgres-XC is already using 2PC command internally to commit transactions involved by multiple data node). It is also straightforward.

More Global Values Most typical example is timestamp. This is an extension to GTM to provide such value. It is similar to `SEQUENCE`.

Drivers Some of the major PostgreSQL drivers will be tested. This has high priority. Candidates are JDBC, ODBC and PHP. Because Postgres-XC does not have proprietary interface to applications, this work will involve to check if statements which uses there drivers implicitly work well.

Cursor We're planning to support cursor in four steps. First, we will support forward cursor without `ORDER BY`, then forward cursor with `ORDER BY`. Third, we will do some architectural and design work to support backward cursor before implementation.

The second step may need releasing old record lock because we may need to move from data node to data node.

Batch Support If multiple statement is included in single input, separated by '`;`', it improves the performance if we don't wait for the response of each statement but keep forwarding the rest of statements. In Postgres-XC, successive statements may be assigned to different data node and such asynchronous response handling may bring great performance gain.

It is rather straightforward and will be addressed according to real requirements.

Primary Node To update replicated table safely, it is important to update table at a fixed data node first then update one on other data nodes. It has the most highest priority and will be done promptly.

Catalog Synchronization Similar to replicated tables, it is also desirable if any DDL can be issued from any coordinator at any time, without causing conflict.

For this purpose, we can use Primary Node idea here too. We may need some architectural and design work for some of DDLs, for example, `ALTER TABLE` before implementation.

Triggers Trigger is a complicated issue. If trigger is fired at a data node and the action is closed in such a data node, there will be almost nothing to worry. However, we should support the case a trigger action has to run on the coordinator. We also need delayed action at the end of transaction.

Constraints The most difficult constraint to enforce in distributed table is "`UNIQUE`." To enforce this strictly, we may need some global index. Similar difficulty is involved in reference integrity.

We may have to ease up this to several levels to avoid check overhead among the nodes.

Tuple Relocation We may need to allow to update distribution column values. It may need tuple transfer from data node to data node. This case is a bit different from tuple transfer infrastructure because the target data is not for work. It is persistent data.

Performance Spike You may have seen we have some performance *spike* in long period test result. So far, we're thinking that it is caused by vacuum analyze which tend to run longer as we run Postgres-XC longer. The cause will be identified and fixed.

Pool for coordinator This will be useful when we synchronize DDL among coordinators as well.

[End of Document]