PGConfchina 2012

Configuring Write-Scalable PostgreSQL Cluster

Postgres-XC Primer and More

by
Koichi Suzuki

June 15th, 2012

Agenda

- A – Postgres-XC introduction
- B - Cluster design, navigation and configuration
- C - Distributing data effectively
- D - Backup and restore, high availability
- E - Postgres-XC as a community

Chapter A
Postgres-XC Introduction

A-1
What Postgres-XC is and what it is not

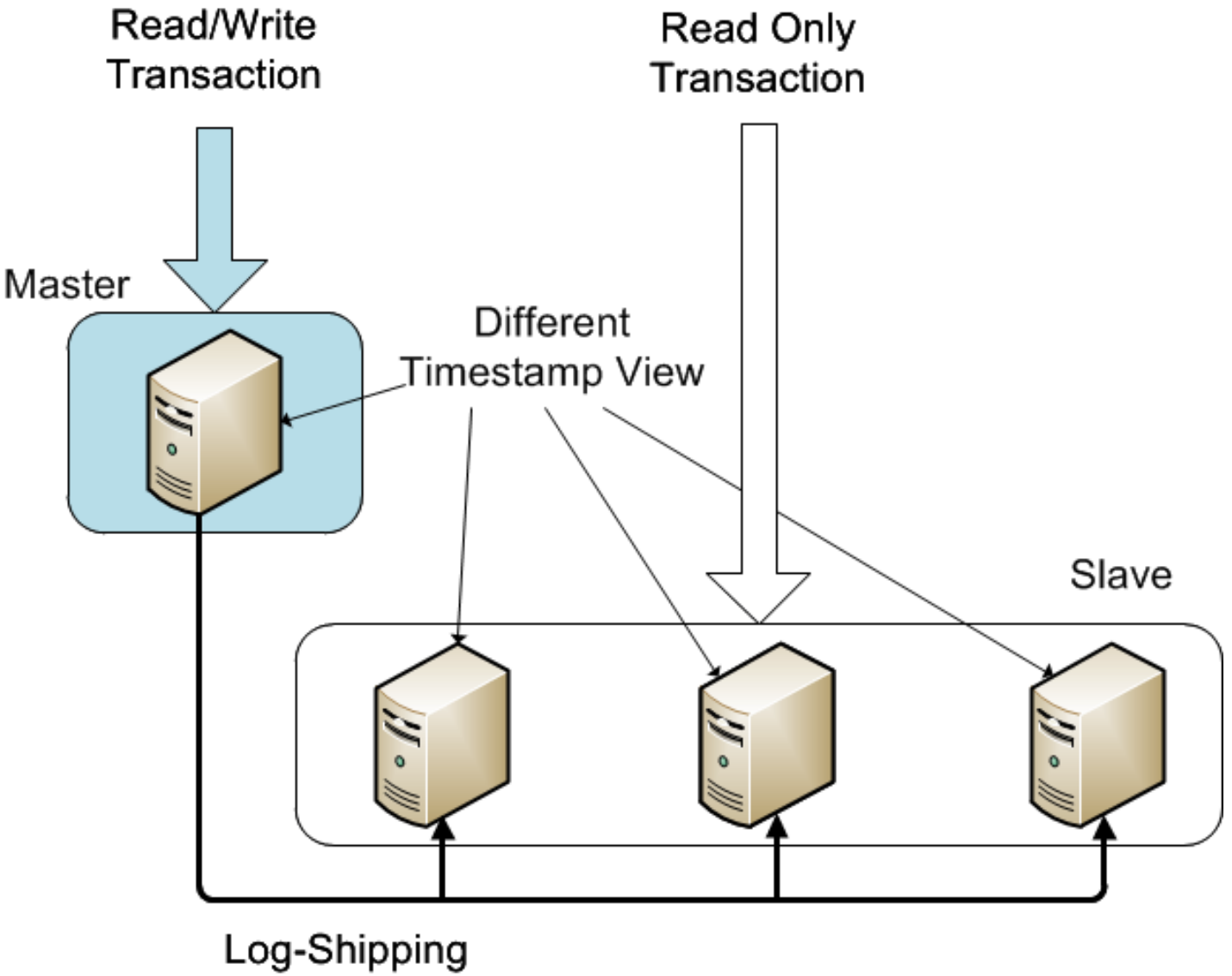
Summary (1)

- PostgreSQL-based database cluster
  - Binary compatible applications
    - Many core extension
  - Catches up latest PostgreSQL version
    - At present based upon PG 9.1. Soon will be upgraded to PG 9.2.
- Symmetric Cluster
  - No master, no slave
    - Not just PostgreSQL replication.
    - Application can read/write to any server
  - Consistent database view to all the transactions
    - Complete ACID property to all the transactions in the cluster
- Scales both for Write and Read

Summary (2)

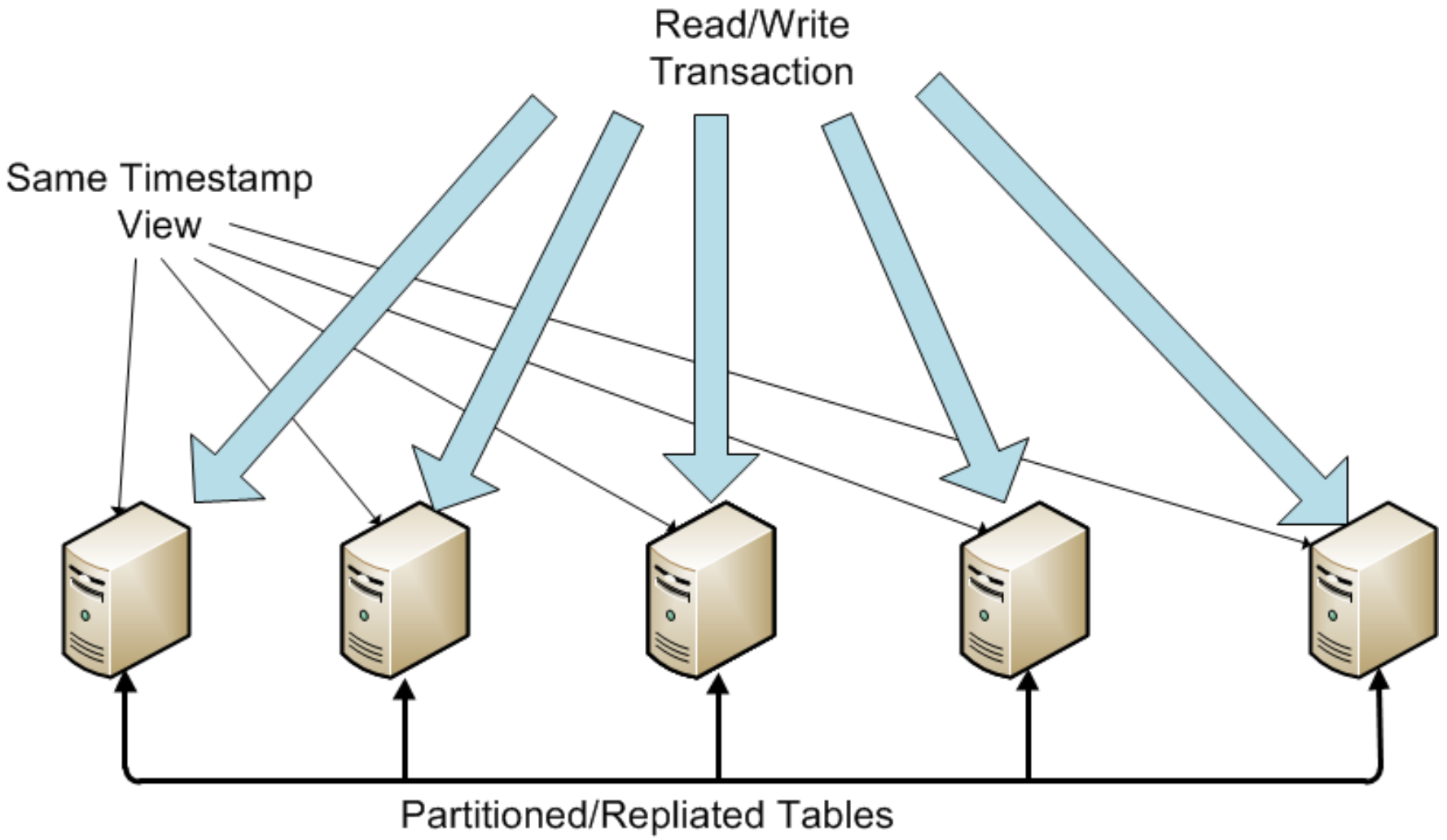
- Not just a replication
  - Configured to provide parallel transaction/statement handling.
  - HA configuration needs separate setups (explained later)

Symmetric Cluster (1)
PostgreSQL replication

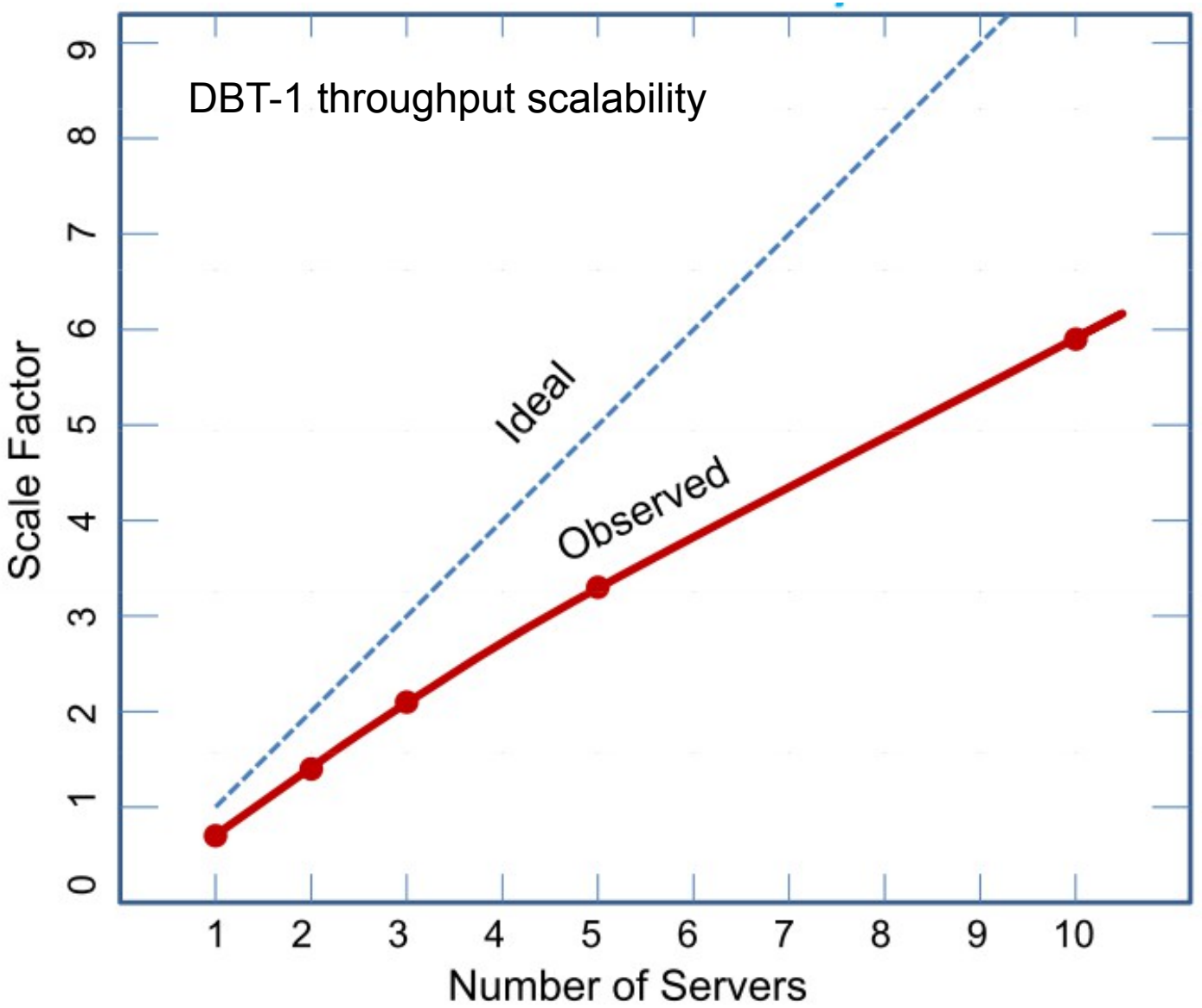


Symmetric Cluster (2)

Postgres-XC Cluster



Read/Write Scalability



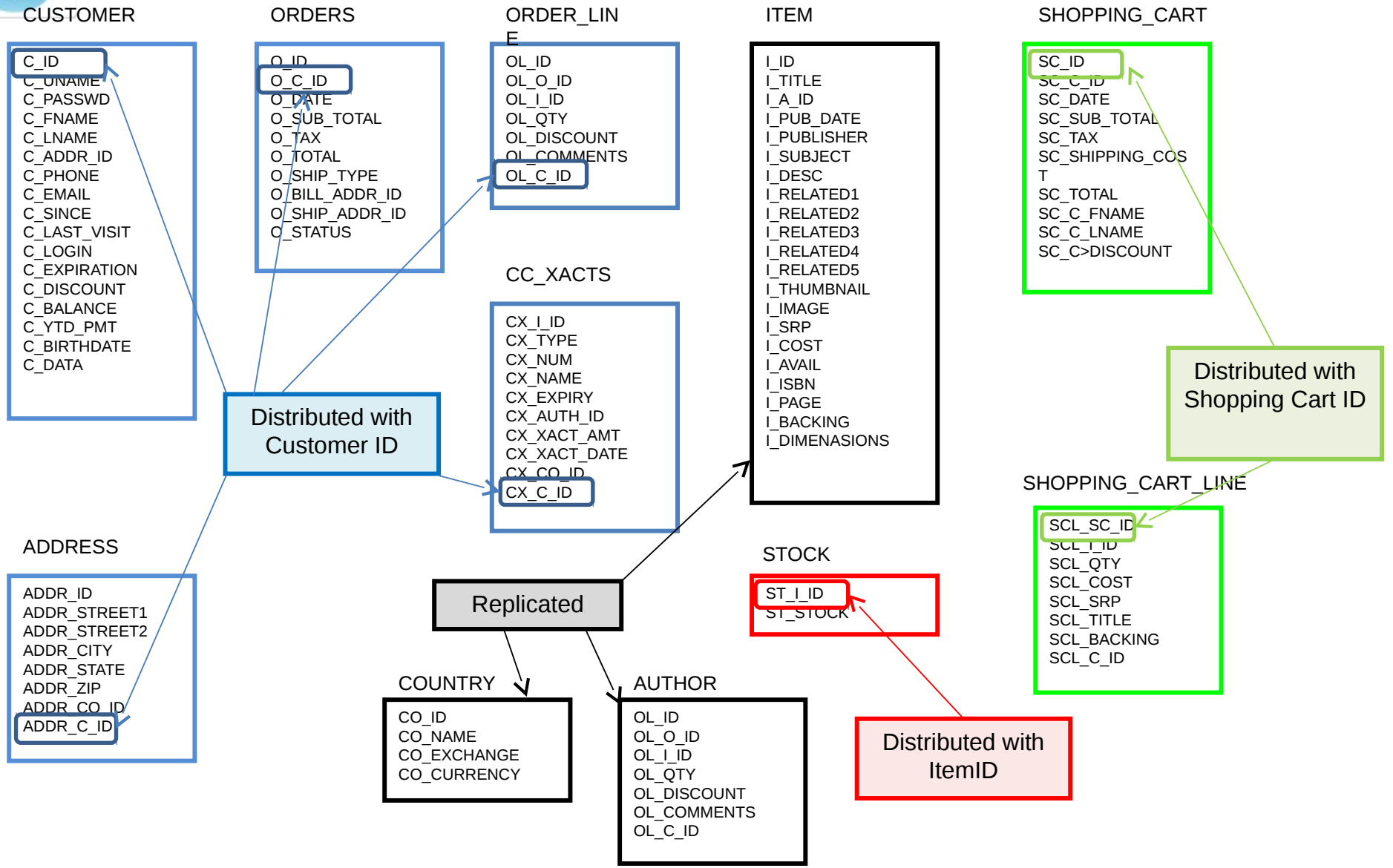
Present Status

- Project/Developer site
  - `http://postgres-xc.sourceforge.net/`
  - `http://sourceforge.net/projects/postgres-xc/`
- Now Version 1.0 available
  - Base PostgreSQL version: 9.1
    - Promptly merged with PostgreSQL 9.2 when available
  - 64bit Linux on Intel X86_64 architecture
    - Tested on CentOS 5.3 and ubuntu 10.4
    - Tested on different platform by community members

How to achieve R/W scalability
Table distribution and replication

- Each table can be distributed or replicated
  - Not a simple database replication
  - Well suited to distribute star schema structure database
    - Transaction tables → Distributed
    - Master tables → Replicate
  - Join pushdown
  - Where clause pushdown
  - Parallel aggregates

DBT-1 Example



Major Difference from PostgreSQL

- Table distribution/replication consideration
  - `CREATE TABLE tab (…) DISTRIBUTE BY HASH(col) | MODULO(col) | REPLICATE | ROUND ROBIN`
- Configuration
  - Purpose of this tutorial
- Some missing features (many may come with V2.0 though)
  - WHERE CURRENT OF
  - Trigger
  - Savepoint ...

A-2
Postgres-XC Components

Summary

- Coordinator
  - Connection point from Apps
  - SQL analysis and global planning
  - Global SQL execution
- Datanode (or simply “NODE”)
  - Actual database store
  - Local SQL execution

Postgres-XC Kernel, based upon vanilla PostgreSQL

Share the binary

Recommended to configure as a pair in OLTP Apps.

- GTM (Global Transaction Manager)
  - Provides consistent database view to transactions
    - GXID (Global Transaction ID)
    - Snapshot (List of active transactions)
    - Other global values such as SEQUENCE

Different binaries

- GTM Proxy
  - integrates server-local transaction requirement for performance

Difference from vanilla PostgreSQL

- More than one Postgres-XC kernel (almost PostgreSQL kernel)
- GTM and GTM-Proxy
- May connect to any one of the coordinators
  - Provide single database view
  - Full-fledged transaction ACID property
    - Some restrictions in SSI (serializable)

How single database view is provided

- Pick up vanilla PostgreSQL MVCC mechanism
  - Transaction ID (Transaction timestamp)
  - Snapshot (list if active transactions)
  - CLOG (whether given transaction is committed)
- Made the former two global → GTM
  - CLOG is still locally maintained by coordinators and datanodes
  - Every coordinator/datanode shares the same snapshot at any given time
- 2PC is used for transactions spanning over multiple coordinators and/or datanodes
  - Has some performance penalty and may be improved in the future
- SSI is still a challenge.

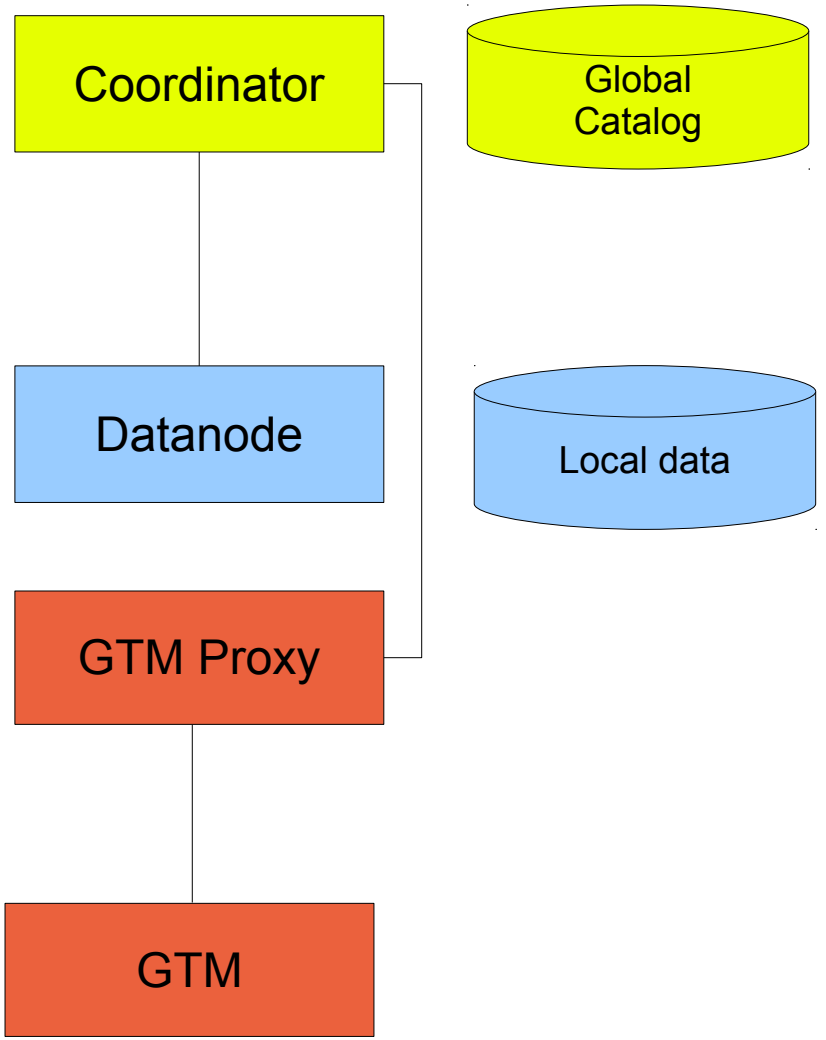
How each component works

Act as just PostgreSQL
Determines which datanode(s) to go
Distributed query planning
Provide GXID and snapshot to datanode(s)

Handle local statement(s) from coordinators
More like just single PostgreSQL

Reduce GTM interaction by grouping GTM request/response
Takes care of connection/reconnection to GTM

Provides global transaction ID and snapshot
Provides sequence



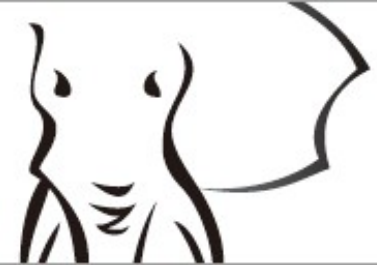
Additional Info for configuration

- Both coordinator/datanode can have their own backups using PostgreSQL log shipping replication.
- GTM can be configured as “standby”, a live backup

Explained later

Chapter B
Cluster design, navigation and configuration

Friendly approach to XC



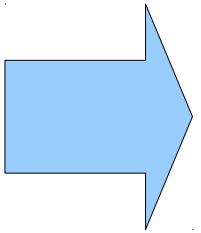
B-1
Design of cluster

About servers and applications

General advice

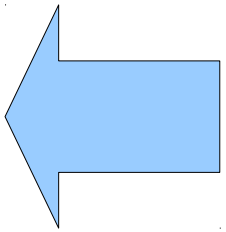
- Avoid data materialization on Coordinator

```
postgres=# explain (costs false) select * from aa,bb where aa.a = bb.a;
                 QUERY PLAN
-------------------------------------
 Data Node Scan on "__REMOTE_FQS_QUERY__"
   Node/s: dn1, dn2
(2 rows)
```



YES !

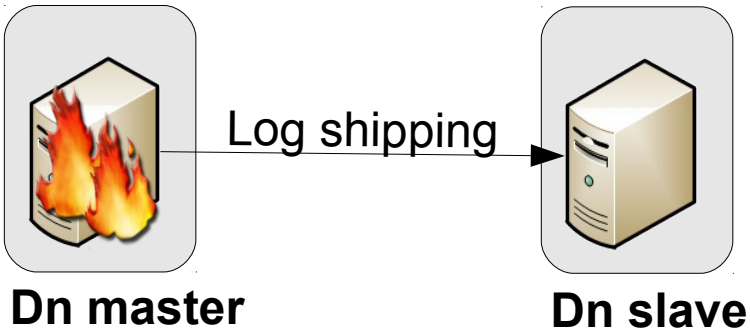
NO !

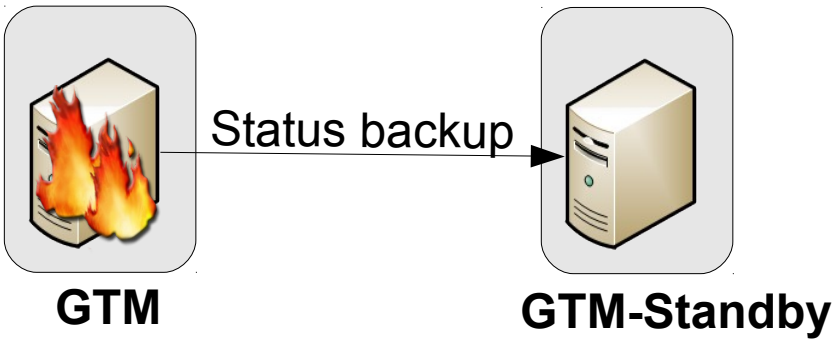


```
postgres=# explain (costs false) select * from aa,bb where aa.a = bb.a;
          QUERY PLAN
-----------------------------
 Nested Loop
   Join Filter: (aa.a = bb.a)
   ->  Data Node Scan on aa
         Node/s: dn1, dn2
   ->  Data Node Scan on bb
         Node/s: dn1
(6 rows)
```

High-availability design

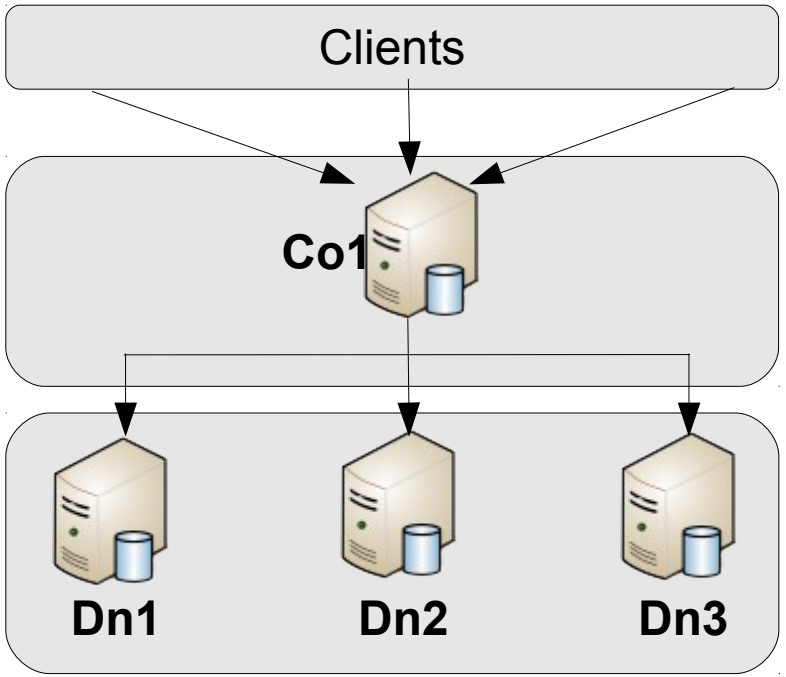
- Streaming replication on critical nodes
  - Nodes having unique data
  - Do not care about unlogged tables for example
- GTM-Standby
  - GTM is SPOF
  - Need to fallback to a standby if failure





Flexible configuration (1)
One coordinator

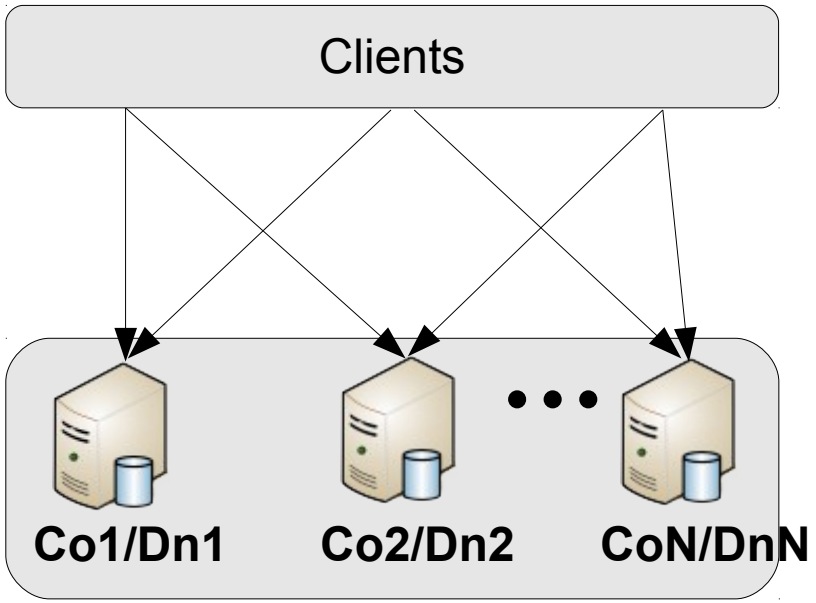
- Coordinator/Datanode CPU on same machine 30/70
- 1/3 ratio on separate servers/VMs



- Master table: replicated table used for joins
- Warehouse table of DBT-2

Coordinator/Datanode pair

- Automatic load balance between the two.
- 1Co/1Dn on same server/VM



- Maximize local joins with preferred node

B-2
Code and binaries

Code navigation and deployment

Where to get source?

- Sourceforge website

  http://sourceforge.net/projects/postgres-xc/files/Version_1.0/
- Sourcseforge (readonly)
  - git clone git://postgres-xc.git.sourceforge.net/gitroot/postgres-xc/postgres-xc
- Github (readonly)
  - git clone https://github.com/postgres-xc/postgres-xc.git

When you get source code

- If you downloaded tarball → extract it.
- If you copied GIT repository
  - Source code is extracted
  - You may check out any commit of XC history

Structure of code - 1

- Code navigation:
  - Use of flags #ifdef PGXC .. #endif
- GTM
  - Folder: src/gtm/
  - Contains GTM and GTM-Proxy code
  - Postgres-side, GTM-side and clients
- Node location management
  - Folder: src/backend/pgxc/locator
  - Node hashing calculation and determination of executing node list

Structure of code - 2

- Pooler
  - Folder: src/backend/pgxc/pool/
  - Pooler process, postgres-side management
- Node manager
  - Folder: src/backend/pgxc/nodemgr/
  - Node and node group catalogs. Local node management
- Planner
  - Folder: src/backend/pgxc/plan/
  - Fast-query shipping code
- Documentation
  - Folder: doc-xc/

Compilation and deployment

- Same as vanilla PostgreSQL
  - ./configure –prefix...
  - make html/man/world and make install
- Deployment methods:
  - Install core binaries/packages on all the servers/VMs involved
  - Or... compile once and copy binaries to all the servers/VMs
- Configurator, automatic deployment through cluster
  - Written in Ruby
  - YAML configuration file
  - Not supported since 0.9.4 :(

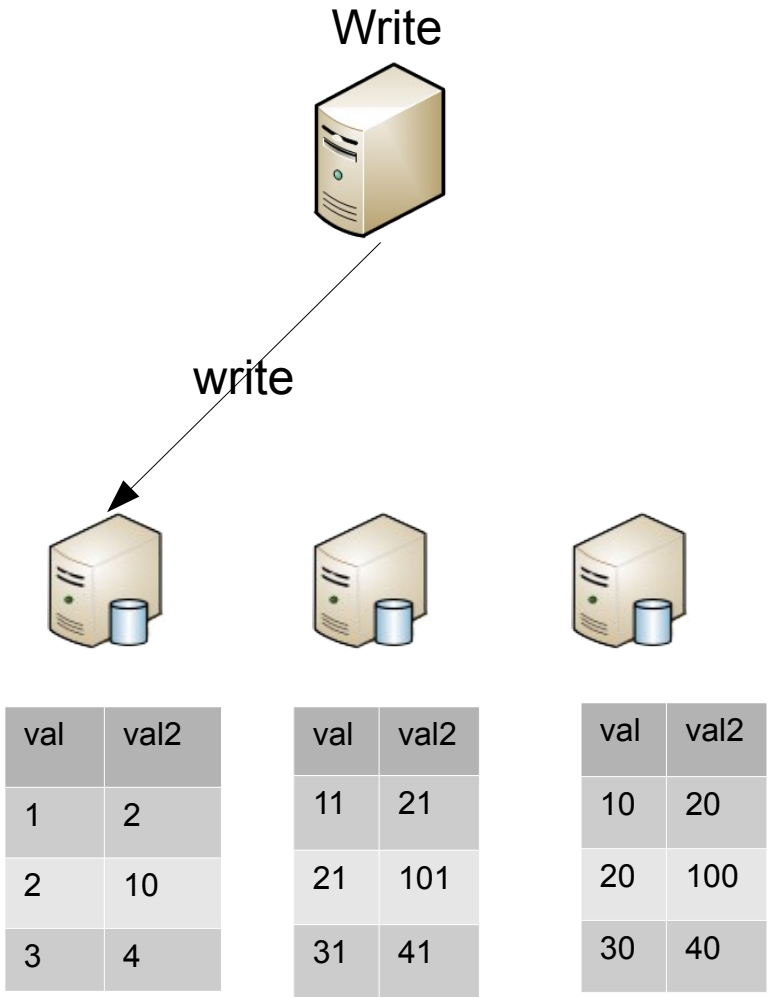
B-3
Everything about configuration

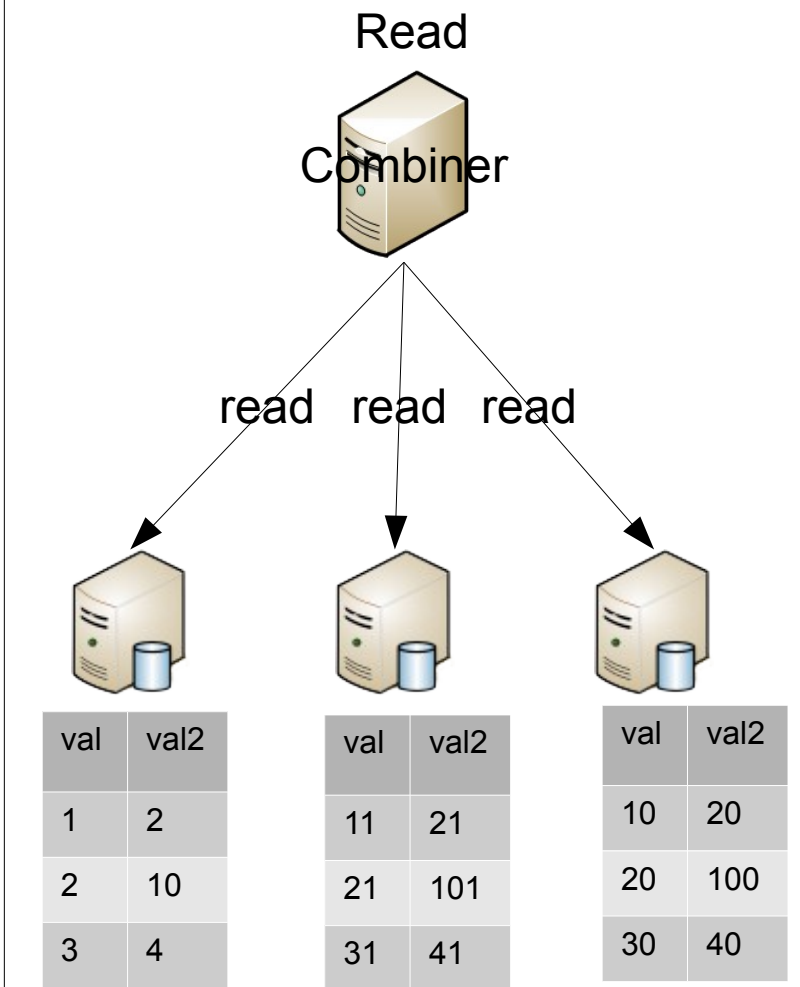
Settings and options

Fundamentals

- Each component (coordinator/datanode, etc) needs its own configuration file.
  - No central single configuration file
  - Location is specified by -D option
    - Must be visible as a part of local file system of the node, either mounted by NFS or SMB

Distributed Tables





B-4
Data management and cluster-related commands

All the core mechanisms to manage and... check your cluster

Initialization

- Initialization of GTM – creation of gtm.conf
  - Mandatory options:
    - Data folder
    - GTM or GTM-Proxy?
  - Example: `initgtm -Z gtm -D $DATA_FOLDER`
- Initialization of a node
  - Mandatory option
    - Node name => `--nodename`
    - Data folder => `-D`
  - Example:

```
initdb --nodename mynode -D $DATA_FOLDER
```

Configuration parameters

- Basics are same as vanilla Postgres
- Extra configuration for all the nodes
  - GTM connection parameters: `gtm_host/gtm_port`
  - Node name: `pgxc_node_name` for self identification
- Coordinator-only
  - Pooler parameters: `pooler_port`, `min_pool_size`, `max_pool_size`
  - `persistent_datanode_connections`: connections taken for session not sent back to pool

Parameters exclusive to XC

- `enforce_two_phase_commit` – default = on
  - Control of autocommit temporary objects
  - Turn to off to create temporary objects
- `max_coordinators` – default 16, max Coordinators usable
- `max_datanodes` – default 16, max Datanodes usable

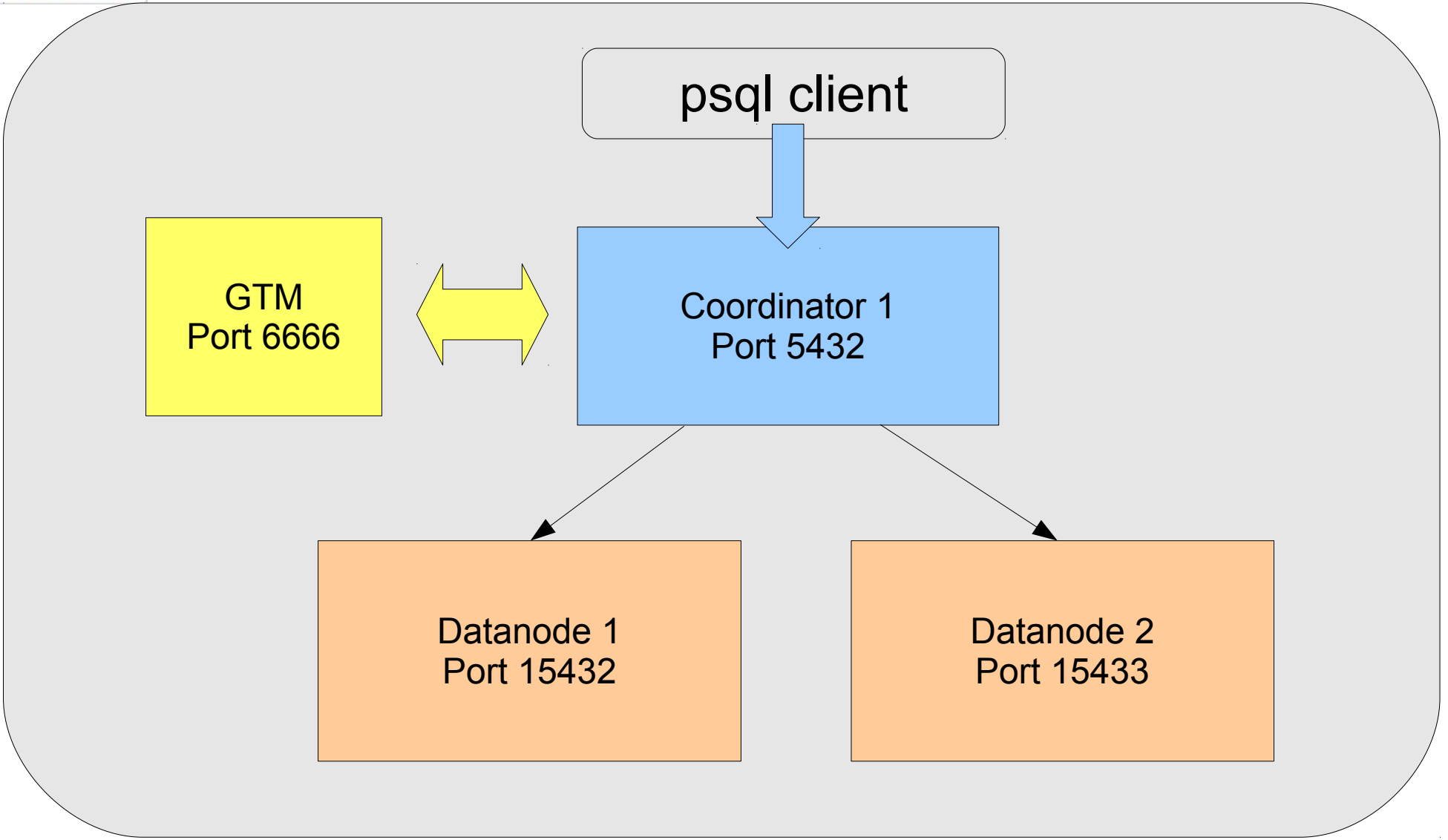
Node start-up

- Coordinator/Datanode
  - Same options as vanilla Postgres
  - Except... Mandatory to choose if node starts up as a Coordinator (`-C`) or a Datanode (`-X`)
  - Possible to set with
    ```
    pg_ctl -Z coordinator/datanode
    ```
- GTM
  - `gtm -D $DATA_FOLDER`
  - `gtm_ctl start -D $DATA_FOLDER -Z gtm`

Commands – cluster management

- `CREATE/ALTER/DROP NODE`
- `CREATE NODE GROUP`
- System functions
  - `pgxc_pool_check()`
  - `pgxc_pool_reload()`

Demonstration



CREATE TABLE extensions

- Control of table disribution
  - DISTRIBUTE BY
    - REPLICATION
    - HASH(*column*)
    - MODULO(*column*)
    - ROUND ROBIN
- Data repartition
  - TO NODE *node1*, … *nodeN*
  - TO GROUP *nodegroup*

Postgres-XC commands - Maintenance

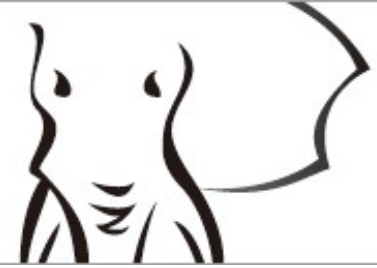
- `EXECUTE DIRECT`
  - Connect directly to a node
  - `SELECT` queries only
  - Can be used to check connection to a remote node
  - Local maintenance
- `CLEAN CONNECTION`
  - Drop connections in pool for given database or user

Point in time recovery - PITR

- `CREATE BARRIER`
  - Addition of a barrier ID in Xlogs consistent in cluster
  - Block 2PC transactions to have a consistent transaction status
- Recovery.conf – `recovery_target_barrier`
  - Specify a barrier ID in recovery.conf to recover a node up to a given barrier point

Cluster catalogs

- `pgxc_node` – Information of remote nodes
  - Connection info: host/port
  - Node type
  - Node name
- `pgxc_group` – node group information
- `pgxc_class` – table distribution information
  - Table Oid
  - Distribution type, distribution key
  - Node list where table data is distributed



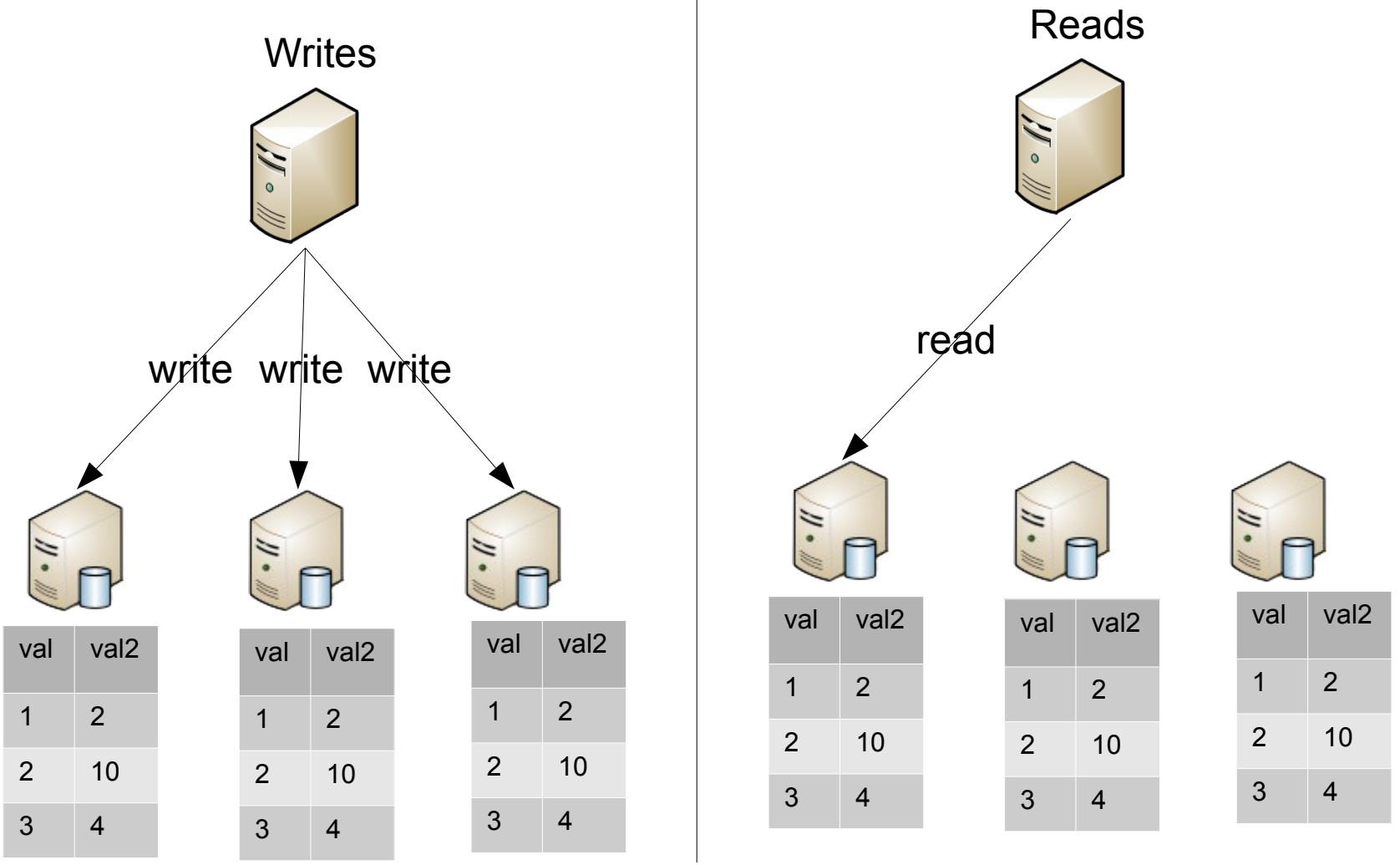
Chapter C
Distributing data effectively

Distribution strategies
Choosing distribution strategy
Transaction Management

Distributing the data

- Replicated table
  - Each row in the table is replicated to the datanodes
  - Statement based replication
- Distributed table
  - Each row of the table is stored on one datanode, decided by one of following strategies
    - Hash
    - Round Robin
    - Modulo
    - Range and user defined function – TBD

Replicated Tables



Replicated Tables

- Statement level replication
- Each write needs to be replicated
  - writes are costly
- Read can happen on any node (where table is replicated)
  - Reads are faster, since reads from different coordinators can be routed to different nodes
- Useful for relatively static tables, with high read load

Querying replicated tables

- Example: simple SELECT on replicated table

```
CREATE TABLE tab1 (val int, val2 int)
DISTRIBUTE BY REPLICATION TO NODE datanode_1, datanode_2;
EXPLAIN VERBOSE SELECT * FROM tab1 WHERE val2 = 5;
                          QUERY PLAN
-------------------------------------------------------------
 Result
   Output: val, val2
   ->  Data Node Scan on tab1 -- Queries the datanode/s
         Output: val, val2
         Node/s: datanode_1 -- one node out of two is chosen
         Remote query: SELECT val, val2 FROM ONLY tab1 WHERE (val2 = 5)
```

Replicated tables - multi-row operations

- Example: aggregates on replicated table

```
EXPLAIN VERBOSE SELECT sum(val) FROM tab1 GROUP BY val2;
                        QUERY PLAN
-----------------------------------------------------------
 HashAggregate -- Groups rows on coordinator, N(groups) < N(rows)
   Output: sum(val), val2
   ->  Data Node Scan on tab1
         -- Brings all the rows from one datanode to coordinator.
         Output: val, val2
         Node/s: datanode_1
         Remote query: SELECT val, val2 FROM ONLY tab1 WHERE true
```

Replicated tables – mult-irow operation

- Similarly push DISTINCT, ORDER BY etc.

```
EXPLAIN VERBOSE SELECT sum(val) FROM tab1 GROUP BY val2;
                              QUERY PLAN
-----------------------------------------------------------------
 Data Node Scan on "__REMOTE_FQS_QUERY__"
   Output: sum(tab1.val), tab1.val2
   Node/s: datanode_1
   Remote query: SELECT sum(val) AS sum FROM tab1 GROUP BY val2
   -- Get grouped and aggregated results from datanode
```

- Pushing aggregates to the datanodes for better performance

Distributed Table

- Write to a single row is applied only on the node where the row resides
  - Point read performance is good
- Multiple rows can be written in parallel
  - Write performance is good
- Scanning rows spanning across the nodes (e.g. table scans) can hamper performance
- Good for tables with high point-write and point-read load

Querying distributed table

- Example: simple SELECT on distributed table

```
CREATE TABLE tab1 (val int, val2 int)
DISTRIBUTE BY HASH(val)
TO NODE datanode_1, datanode_2, datanode_3; -- distributed table
EXPLAIN VERBOSE SELECT * FROM tab1 WHERE val2 = 5;
                          QUERY PLAN
------------------------------------------------------------------
 Data Node Scan on "__REMOTE_FQS_QUERY__"
   -- Gathers rows from the nodes where table is distributed
   Output: tab1.val, tab1.val2
   Node/s: datanode_1, datanode_2, datanode_3
   Remote query: SELECT val, val2 FROM tab1 WHERE (val2 = 5)
```

Distributed tables – multi-row operations (1)

- Example: aggregates on distributed table

```
EXPLAIN VERBOSE SELECT sum(val) FROM tab1 GROUP BY val2;
                        QUERY PLAN
-----------------------------------------------------------------
 HashAggregate -- Groups rows on coordinator, N(groups) < N(rows)
   Output: sum(val), val2
   ->  Data Node Scan on tab1
         -- Brings all the rows from the datanode to coordinator.
         Output: val, val2
         Node/s: datanode_1, datanode_2, datanode_3
         Remote query: SELECT val, val2 FROM ONLY tab1 WHERE true
```

Distributed tables – multi-row operation (2)

- Example: aggregates and GROUP BY on distributed table

```
EXPLAIN VERBOSE SELECT sum(val) FROM tab1 GROUP BY val2;
                        QUERY PLAN
-----------------------------------------------------------------
 HashAggregate
   Output: pg_catalog.sum((sum(tab1.val))), tab1.val2
   -- Finalise the grouping and aggregation at coordinator
         ->  Data Node Scan on "__REMOTE_GROUP_QUERY__"
             Output: sum(tab1.val), tab1.val2
             Node/s: datanode_1, datanode_2, datanode_3
             Remote query: SELECT sum(group_1.val), group_1.val2
                           FROM (SELECT val, val2 FROM ONLY tab1
                           WHERE true) group_1 GROUP BY 2
   -- Get partially grouped and aggregated results from datanodes
```

JOINs

- Example: Join on Coordinator with an integer

```
EXPLAIN VERBOSE SELECT * FROM tab1, tab2 WHERE tab1.val = tab2.val;
                        QUERY PLAN
-----------------------------------------------------------------
 Nested Loop – Perform JOIN on coordinator.
   Output: tab1.val, tab1.val2, tab2.val, tab2.val2
   Join Filter: (tab1.val = tab2.val)
                Queries to datanodes to fetch the rows from tab1 and tab2
   ->  Data Node Scan on tab1
         Output: tab1.val, tab1.val2
         Remote query: SELECT val, val2 FROM ONLY tab1 WHERE true
   ->  Data Node Scan on tab2
         Output: tab2.val, tab2.val2
         Remote query: SELECT val, val2 FROM ONLY tab2 WHERE true
```

JOINs

- Performing JOIN on coordinator won't be efficient if
  - Number of rows selected << size of cartesian product of the relations
  - The size of the JOIN result tuple is not large compared to the rows from the joining relations
- It will be efficient if
  - Number of rows selected ~= size of cartesian product of the relations
  - The size of JOIN result tuple is very large compared to the individual row-sizes

Perfoming JOINs on datanodes

- Example: Join on Datanode with an integer key

```
EXPLAIN VERBOSE SELECT * FROM tab1, tab2 WHERE tab1.val = tab2.val;
                                  QUERY PLAN
------------------------------------------------------------------------
 Data Node Scan on "__REMOTE_FQS_QUERY__"
   Output: tab1.val, tab1.val2, tab2.val, tab2.val2
   Node/s: datanode_1
   Remote query: SELECT tab1.val, tab1.val2, tab2.val, tab2.val2
                        FROM tab1, tab2 WHERE (tab1.val = tab2.val)
```

– JOIN performed on datanode

Performing JOINs on datanodes

- Indexes can help to perform JOIN faster
  - Indexes available only on datanode
- Aggregates, grouping, sorting can as well be performed on datanode
- Always perform JOINs on the datanode
  - In XC, coordinators do not have correct statistics, so can't predict the selectivity
  - Proper costing model is yet to be implemented

Shippability of Joins

	Hash/Module distributed	Round Robin	Replicated
Hash/Modulo distributed	Inner join with equality condition on the distribution column with same data type and same distribution strategy	NO	Inner join if replicated table's distribution list is superset of distributed table's distribution list
Round Robin	No	No	Inner join if replicated table's distribution list is superset of distributed table's distribution list
Replicated	Inner join if replicated table's distribution list is superset of distributed table's distribution list	Inner join if replicated table's distribution list is superset of distributed table's distribution list	All kinds of joins

Constraints

- XC does not support Global constraints – i.e. constraints across datanodes
- Constraints within a datanode are supported

Distribution strategy	**Unique, primary key constraints**	**Foreign key constraints**
Replicated	Supported	Supported if the referenced table is also replicated on the same nodes
Hash/Modulo distributed	Supported if primary OR unique key is distribution key	Supported if the referenced table is replicated on same nodes OR it's distributed by primary key in the same manner and same nodes
Round Robin	Not supported	Supported if the referenced table is replicated on same nodes

Choosing distribution strategy (1)

- Replication - if
  - The table is less-frequently written to
  - The table's primary key is referenced by many distributed tables
  - The table needs to have a primary key/unique key which is can not be distribution key
  - The table is part of JOIN for many queries – makes easier to push the JOIN to the datanode
  - Data redundancy

Choosing distribution strategy (2)

- Hash/Modulo distributed – if
  - There are high point-read/write loads
  - The queries have equality conditions on the distribution key, such that the data comes from only one node (essentially it becomes equivalent to replicated table for that query)
- Round Robin – if
  - There is no definite distribution key, but still want to balance the write load across the cluster

Example DBT-1 (1)

- `author`, `item`
  - Less frequently written
  - Frequently read from
  - Author and item are frequently JOINed
  - Hence replicated on all nodes

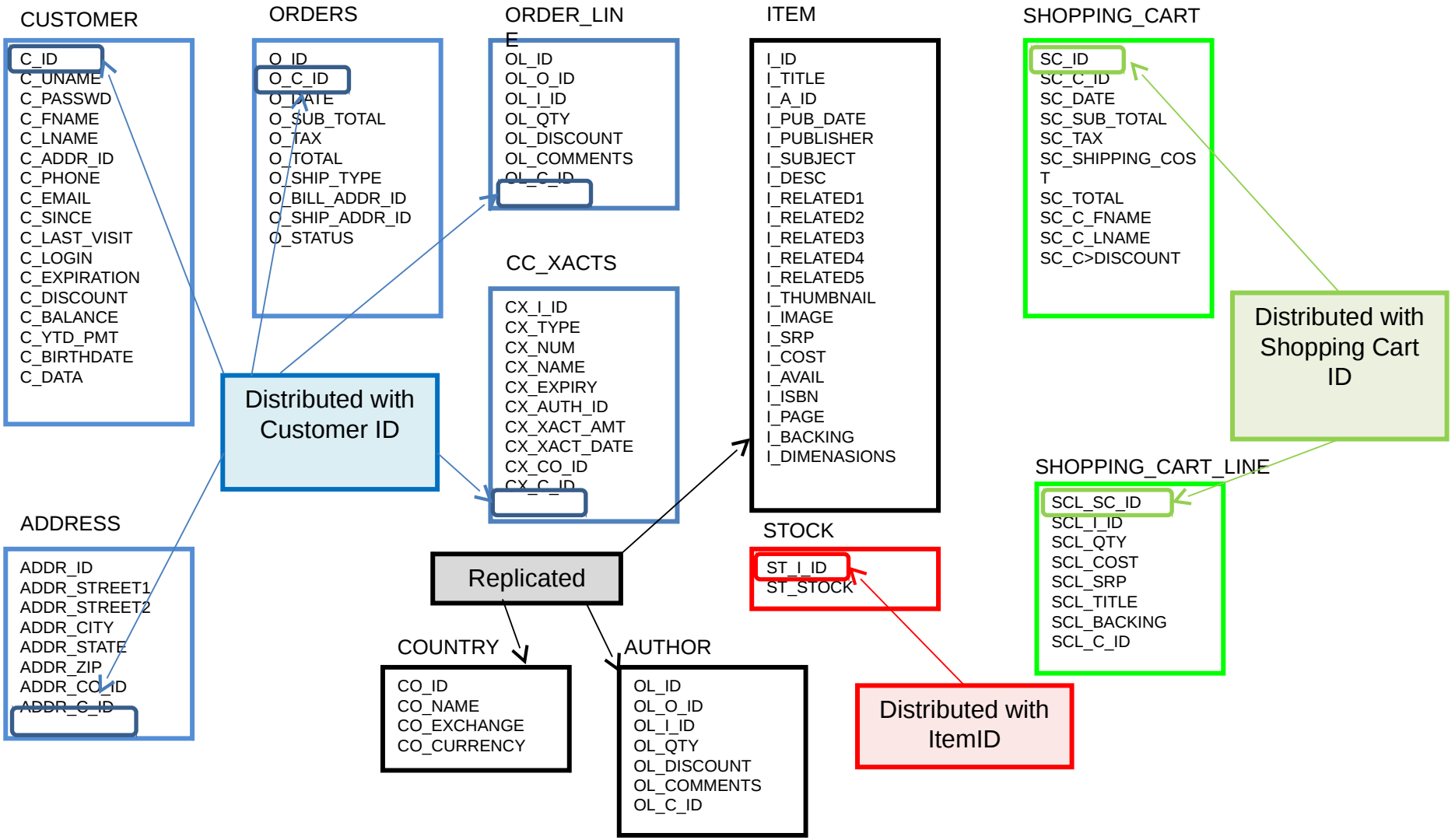
Example DBT-1 (2)

- customer, address, orders, order_line, cc_xacts
  - Frequently written
    - hence distributed
  - Participate in JOINs amongst each other with customer_id as JOIN key, point SELECTs based on customer_id
    - hence diistributed by hash on customer_id so that JOINs are shippable
  - Participate in JOINs with item
    - Having item replicated helps pushing JOINs to datanode

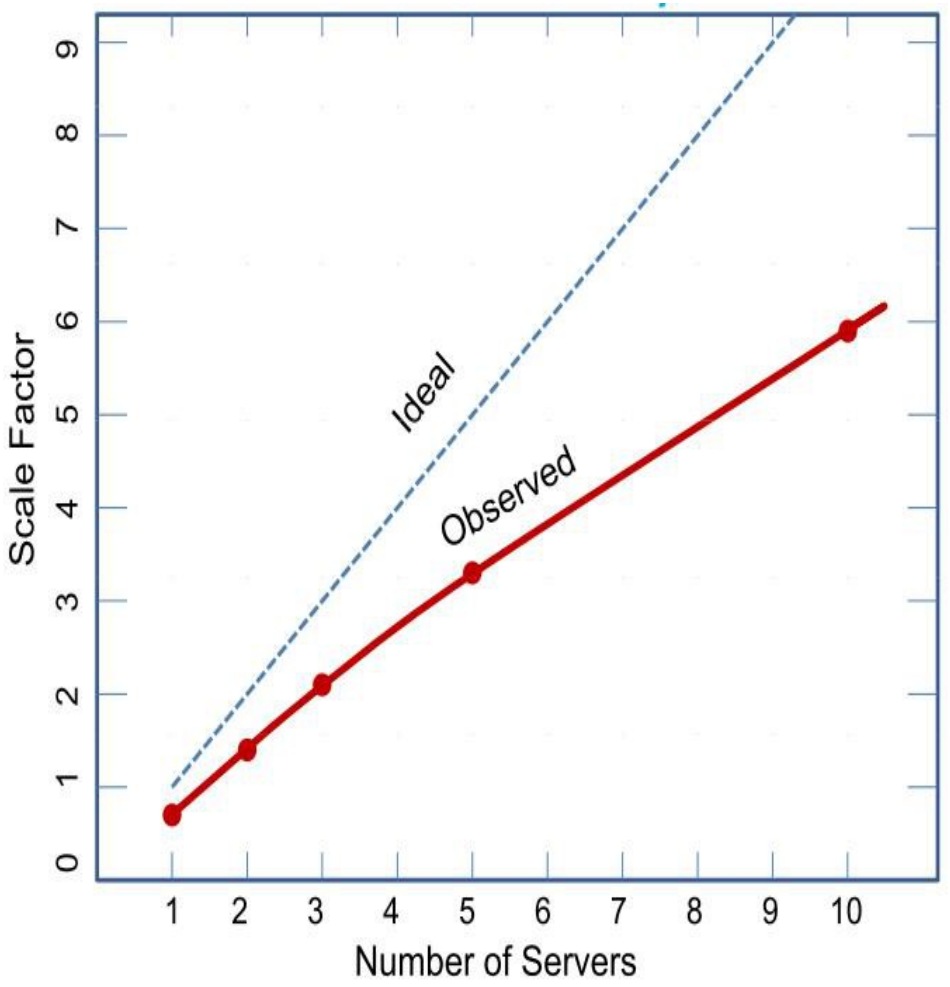
Example DBT-1 (3)

- Shopping_cart, shopping_cart_line
  - Frequently written
    - Hence distributed
  - Point selects based on shopping_cart_id
    - Hence distributed by hash on shopping_cart_id
  - JOINs with item on item_id
    - Having item replicated helps pushing JOINs to datanode

Example DBT-1 (4)



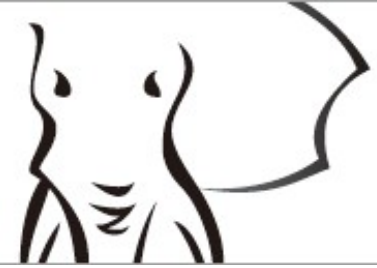
DBT-1 scale-up



- Old data, we will publish bench-marks for 1.0 soon.
- DBT-1 (TPC-W) benchmark with some minor modification to the schema
- 1 server = 1 coordinator + 1 datanode on same machine
- Coordinator is CPU bound
- Datanode is I/O bound

Transaction management

- 2PC is used to guarantee transactional consistency across nodes
  - When there are more than one nodes involved OR
  - When there are explicit 2PC transactions
- Only those nodes where write activity has happened, participate in 2PC
- In PostgreSQL 2PC can not be applied if temporary tables are involved. Same restriction applies in Postgres-XC
- When single coordinator command needs multiple datanode commands, we encase those in transaction block



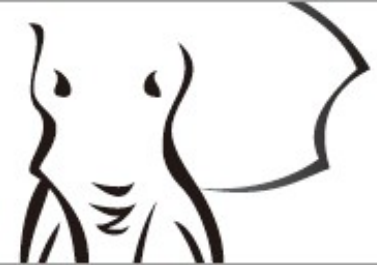
Chapter D
Backup, restore, recovery and high availability

Example configuration (1)

- Coordinator x 2
  - coord1
    - -D=/home/koichi/pgxc/nodes/coord1
    - port: 20004
  - coord2
    - -D=/home/koichi/pgxc/nodes/coord2
    - port: 20005
- Datanode x 2
  - datanode1
    - -D=/home/koichi/pgxc/nodes/datanode1
    - port: 20006
  - datanode2
    - -D=/home/koichi/pgxc/nodes/datanode2
    - port: 20007

Example configuration (2)

- GTM
  - -D=/home/koichi/pgxc/nodes/gtm
  - port: 20001
- GTM-Standby
  - -D=/home/koichi/pgxc/nodes/gtm_standby
  - port: 20000
- GTM Proxy X 2
  - gtm_pxy1
    - -D=/home/koichi/pgxc/nodes/gtm_pxy1
    - port: 20002
    - Connects: coord1, datanode1
  - gtm_pxy2
    - -D=/home/koichi/pgxc/nodes/gtm_pxy2
    - port: 20003
    - Connect: coord2, datanode2



D-1
Backup

General

- Coordinator and Datanode
  - No difference in principle from vanilla PostgreSQL
  - pg_dump, pg_dumpall works globally
  - Streaming replication should be configured for each coordinator and datanode
  - Restoration should be consistent in all the nodes
    - Barrier
- GTM proxy
  - No dynamic data → Only static configuration files needs backup
- GTM
  - Need to backup current status → GTM Standby (explained later)

pg_dump, pg_dumpall, pg_restore

- You don't have to backup each node.
- Connect to one of the coordinators and issue pg_dump or pg_dumpall to backup.
- Connect to one of the coordinators and issue pg_restore to restore.
- pg_dump extended to support table distribution

Cold backup of each database cluster (static backup) (1)

- After you stop all the components, you can backup all the physical files of each component.
- Restore them and simply restart all the components.
- To maintain whole cluster consistent, you must backup them all at the same occasion (after you stopped the whole cluster).

Cold backup of each database cluster (static backup) (2)

- First, stop whole cluster

```
$ pg_ctl stop -Z coordinator -D /home/kochi/pgxc/nodes/coord1 # -m fast
$ pg_ctl stop -Z coordinator -D /home/koichi/pgxc/nodes/coord2 # -m fast
$ pg_ctl stop -Z datanode -D /home/koichi/pgxc/nodes/datanode1 # -m fast
$ pg_ctl stop -Z datanode -D /home/koichi/pgxc/nodes/datanode2 # -m fast
$ gtm_ctl stop -S gtm_proxy -D /home/koichi/pgxc/nodes/gtm_pxy1
$ gtm_ctl stop -S gtm_proxy -D /home/koichi/pgxc/nodes/gtm_pxy2
$ gtm_ctl stop -S gtm -D /home/koichi/pgxc/nodes/gtm
```

- You should run the above commands on each machines where each coordinator or datanode are running.
- You may not need -m fast option if you disconnect all the connections from coordinator to datanode with `CLEAN CONNECTION` statement.
- As simple as vanilla PostgreSQL, but you should take care of all the running components.

Physical backup of each database cluster (static backup) (3)

- Then, backup everything (tar is used in this case)

```
$ cd /home/koichi/pgxc/node
$ tar cvzf somewhere/gtm.tgz gtm
$ tar cvzf somewhere/gtm_pxy1.tgz gtm_pxy1
$ tar cvzf somewhere/gtm_pxy2.tgz gtm_pxy2
$ tar cvzf somewhere/coord1.tgz coord1
$ tar cvzf somewhere/coord2.tgz coord2
$ tar cvzf somewhere/datanode1.tgz datanode1
$ tar cvzf somewhere/datanode2.tgz datanode2
```

- Again, as simple as vanilla PostgreSQL
- You can use your favorite backup tools. tar, rsync, whatsoever.
- Just you should take care of all the components.

Hot backup (1) General

- Similar to vanilla PostgreSQL
- You need to synchronize restoration point
  - Barrier: `CREATE BARRIER 'barrier_id'`
    - Advise to issue this command periodically from psql
    - Will propagate to all the component
    - Can be used as restoration point

Hot backup (2)

- Other are similar to vanilla PostgreSQL
- Again, you should take care of all the running nodes.
- Setup hot backup (base backup and WAL archiving) for each coordinator and datanode.
- GTM hot backup needs dedicated standby
  - Common to HA configuration

Coordinator/Datanode hot backup

- Like standalone PG backup.

  For each coordinator/datanode, you should

  1. Configure WAL archiving
  2. Take base backups
- Again, simple but have to take care of all.

Hot backup – coord1 example

- Configure WAL archiving

```
$ cat /home/koichi/pgxc/nodes/coord1/postgresql.conf

wal_level = archive
archive_mode = on
archive_command = 'cp -i %p /somewhere/%f </dev/null'
```

- Take base backup

```
$ pg_basebackup -D backup_dir -h hostname -p 20004 -F tar
```

-x option of pg_basebackup

- This option includes WAL segments in the backup and enable to restore without WAL archiving.
- In XC, we should restore all the nodes to the same timestamp.
- You can use this option without WAL archiving
  - If you are quite sure that -x option includes your target “barrier” in backups of all the coordinator/datanode
  - If you are not, you should configure WAL archiving to be sure that all the WAL files includes your target “barrier”.

GTM Hot Backup – GTM Sandby

- GTM has dedicated synchronous backup called GTM-standby.
- Just like PostgreSQL synchronous replication.
- GTM standby shares the binary with GTM.
- Start GTM, then GTM-Standby. GTM-Standby copies every request to GTM and maintains GTM's internal status as a backup.
- GTM-Standby can be promoted to GTM with gtm_ctl command.
- GTM-proxy can reconnect to promoted GTM.

Running GTM-Standby

- Run GTM as

```
$ cat gtm_data_dir/gtm.conf

nodename = 'gtm'          # node name of your choice
listen_addresses = 'gtm_ip_address'
port = 20001              # port number of your choice
startup = ACT             # specify ACT mode to start

$ gtm_ctl start -S gtm -D gtm_data_dir
```

- Run GTM standby as

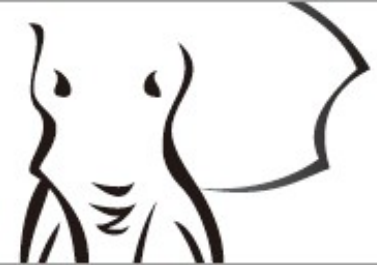
```
$ cat gtm_sby_data_dir/gtm.conf

nodename = 'gtm_standby' # node name of your choice
listen_addresses = 'gtm_standby_ip_address'
port = 20000              # port # of your choice
startup = STANDBY         # specfy to start as standby
active_port = 20001       # ACT gtm port numer
active_host = 'gtm_ip_address'    # ACT gtm ip address

$ gtm_ctl start -S gtm -D gtm_sby_data_dir
```

GTM-Standby backs up

- GTM standby backs up every status of GTM
  - To restore GTM with GTM-Standby, you should promote GTM-standby (explained later)



D-2
Restore

Restore from pg_dump and pg_dumpall

- Initialize whole Postgres-XC cluster
  - Have been covered at “configuration”
- Run pg_restore
  - You don't have to run pg_restore for each coordinator/datanode
  - Select one of the coordinator to connect and run pg_restore, that's all.
  - Takes care of table distribution/replication

Restore from the cold backup (1)

- Restore all the cold backups

```
$ cd /home/koichi/pgxc/node
$ tar xvzf somewhere/gtm.tgz
$ tar xvzf somewhere/gtm_pxy1.tgz
$ tar xvzf somewhere/gtm_pxy2.tgz
$ tar xvzf somewhere/coord1.tgz
$ tar xvzf somewhere/coord2.tgz
$ tar xvzf somewhere/datanode1.tgz
$ tar xvzf somewhre/datanode2.tgz
```

Restore from the cold backup (2)

- Start all the components again

```
$ gtm_ctl start -S gtm -D /home/koichi/pgxc/nodes/gtm
$ gtm_ctl start -S gtm_proxy -D /home/koichi/pgxc/nodes/gtm_pxy1
$ gtm_ctl start -S gtm_proxy -D /home/koichi/pgxc/nodes/gtm_pxy2
$ pg_ctl start -Z datanode -D /home/koichi/pgxc/nodes/datanode1 -o “-i”
$ pg_ctl start -Z datanode -D /home/koichi/pgxc/nodes/datanode2 -o “-i”
$ pg_ctl start -Z coordinator -D /home/koichi/pgxc/nodes/coord1  -o “-i”
$ pg_ctl start -Z coordinator -D /home/koichi/pgxc/nodes/coord2  -o “-i”
```

- Restoration is done.

Restoration from the hot backup Coordinator and Datanode

- Restore the hot backup of each coordinator and datanode just like single PostgreSQL
- Make restore point consistent among coordinators and datanodes.
  - Specify Barrier ID as the restore point

Recovery.conf settings coord1 example

```
$ cat /home/koichi/nodes/coord1/reconvery.conf

restore_command = 'cp /somewhere/%f “%p”'
recovery_target_barrier = 'barrier_id'

$ pg_ctl start -Z coordinator -D /home/koichi/pgxc/nodes/coord1 -o “-i”
```

- You can specify other `recovery.conf` options such as `archive_cleanup_command` if needed.
- Configure recover.conf file for every coordinator and datanode and start them with `pg_ctl`.
- Specify the same *barrier_id* value.

D-3
Recovery

When coordinator or datanode crashes

- Any transaction involved in crashed coordinator or datanode will fail.
- If crash recovery runs, do it as vanilla PostgreSQL.
- If crash recovery does not run, you need archive recovery.
  - You should do archive recovery for every coordinator and datanode to maintain cluster-wide data consistensy.

When GTM-Proxy crashes

- GTM-Proxy does not have any dynamic data.
- Restore backup configuration file and restart.
- You may need to restart coordinator/datanode connected to the failed gtm-proxy

If GTM crashes

- You must have run gtm_standby to recover from GTM crash.
- You must configure XC with gtm_proxy.
- Promote gtm_standby as gtm.
- Then reconnect gtm_proxy to the promoted gtm.
- You don't stop coordinators and/or datanodes.
- No transaction loss.

Running GTM-Standby (again)

- Run GTM as

```
$ cat gtm_data_dir/gtm.conf

nodename = 'gtm'          # node name of your choice
listen_addresses = 'gtm_ip_address'
port = 20001              # port number of your choice
startup = ACT             # specify ACT mode to start

$ gtm_ctl start -S gtm -D gtm_data_dir
```

- Run GTM standby as

```
$ cat gtm_sby_data_dir/gtm.conf

nodename = 'gtm_standby' # node name of your choice
listen_addresses = 'gtm_standby_ip_address'
port = 20000              # port # of your choice
startup = STANDBY         # specfy to start as standby
active_port = 20001       # ACT gtm port numer
active_host = 'gtm_ip_address'    # ACT gtm ip address

$ gtm_ctl start -S gtm -D gtm_sby_data_dir
```

Run GTM Proxies

```
$ cat /home/koichi/pgxc/nodes/gtm_pxy1/gtm_proxy.conf

nodename = 'gtm_pxy1'
listen_addresses = 'gtm_pxy1_ip_address'
port = 20002
gtm_host = 'gtm_ip_address'
gtm_port = 20001

$ gtm_ctl start -S gtm_proxy

$ (do the same thing for gtm_pxy2 as well)
```

GTM recovery procedure

```
$ (GTM crash found)

$ gtm_ctl promote -S gtm \
   -D /home/koichi/nodes/gtm_standby

$ gtm_ctl reconnect -S gtm_proxy \
   -D /home/koichi/pgxc/nodes/gtm_pxy1 \
   -o “-s gtm_standby_ip_addr -t 20000”

$ gtm_ctl reconnect -S gtm_proxy \
   -D /home/koichi/pgxc/nodes/gtm_pxy2 \
   -o “-s gtm_standby_ip_addr -t 20000”
```

Additional gtm_proxy options

- Timer to controls to deal with communication failure and wait for reconnect.

```
err_wait_idle       # timer to wait for the first reconnect
                    # (in second)
err_wait_count      # counts to wait for reconnect
err_wait_interval   # timer to wait next reconnect (in second)

gtm_connect_retry_idle
    # timer to retry connect to the current gtm when error is
    # detected (in second).
gtm_connect_retry_count
    # number of connect retries
gtm_connect_retry_interval
    # interval of connect retry to the current gtm when error is
    # detected (in second).
```

Running GTM-Standby

- Run GTM as

```
$ cat gtm_data_dir/gtm.conf

nodename = 'gtm'        # node name of your choice
listen_addresses = 'gtm_ip_address'
port = 20001            # port number of your choice
startup = ACT           # specify ACT mode to start

$ gtm_ctl start -S gtm -D gtm_data_dir
```

- Run GTM standby as

```
$ cat gtm_sby_data_dir/gtm.conf

nodename = 'gtm_standby' # node name of your choice
listen_addresses = 'gtm_standby_ip_address'
port = 20000            # port # of your choice
startup = STANDBY       # specfy to start as standby
active_port = 20001     # ACT gtm port numer
active_host = 'gtm_ip_address'    # ACT gtm ip address

$ gtm_ctl start -S gtm -D gtm_sby_data_dir
```

Running GTM-Standby

- Run GTM as

```
$ cat gtm_data_dir/gtm.conf

nodename = 'gtm'        # node name of your choice
listen_addresses = 'gtm_ip_address'
port = 20001            # port number of your choice
startup = ACT           # specify ACT mode to start

$ gtm_ctl start -S gtm -D gtm_data_dir
```

- Run GTM standby as

```
$ cat gtm_sby_data_dir/gtm.conf

nodename = 'gtm_standby' # node name of your choice
listen_addresses = 'gtm_standby_ip_address'
port = 20000            # port # of your choice
startup = STANDBY       # specfy to start as standby
active_port = 20001     # ACT gtm port numer
active_host = 'gtm_ip_address'    # ACT gtm ip address

$ gtm_ctl start -S gtm -D gtm_sby_data_dir
```

D-4
High Availability

HA - General

- Postgres-XC's HA feature should be achieved by integration with other HA middleware such as pacemaker (resource agent).
- GTM, GTM-standby and GTM-proxies provides fundamental HA feature.
- Each coordinator and datanode should be configured with synchronous replication.
- This tutorial will focus on Postgres-XC configuration for HA middleware integration.

GTM configuration

- GTM should be configured with GTM-standby and GTM-proxy.
- Monitoring GTM, GTM-standby and GTM-proxy can be done by process monitoring.
- Dedicated monitoring command can be implemented using direct interface to each of them.
  - Need to be familiar with their internal structure.
  - May be developed elsewhere.

Coordinator/datanode configuration

- Again, same as vanilla PostgreSQL.
- Configure each coordinator and datanode synchronous replication as vanilla PostgreSQL.
- At failover, other coordinator/datanode must be notified new connection point.
  - Use `ALTER NODE` command after the failover

```
ALTER NODE datanode1 WITH
   (HOST = 'new_ip_addr', PORT = 20010);
```

  - Then, `SELECT pgxc_pool_reload()`
  - Need to run `pgxc_clean` to cleanup 2PC status.
  - Other procedure is the same as vanilla PostgreSQL standby server settings and operation.

Why pgxc_clean?

- When a node is failed over or recovered, 2PC status could be inconsistent.
  - At some nodes, transaction has been committed or aborted but at other nodes, it might remain prepared.
  - pgxc_clean collects such outstanding 2PC transaction status and correct them.

Cleaning-up outstanding 2PC

- Collects all the prepared transactions at all the coordinator and datanode.
- Check if prepared transactions are committed or aborted at other nodes.
- If committed/aborted, then commits/aborts the transaction at prepared nodes.
- If implicit 2PC transaction is only prepared, it is intended to be committed and commit it.
- If outstanding 2PC transaction is committed and aborted in other nodes, it is an error. pgxc_clean will notifiy and remain this to manual actions.

Chapter E
Postgres-XC as a community

How to enter the sect...

Sites to know

- Main management in Source Forge
  - URL: http://sourceforge.net/projects/postgres-xc/
  - Bug tracker, mailing lists...
- Other GIT repository in Github
  - URL: https://github.com/postgres-xc/postgres-xc
  - Mirror of official repository in Source Forge
- Project webpage
  - URL: http://postgres-xc.sourceforge.net/
  - Roadmap, mailing list details, members, docs

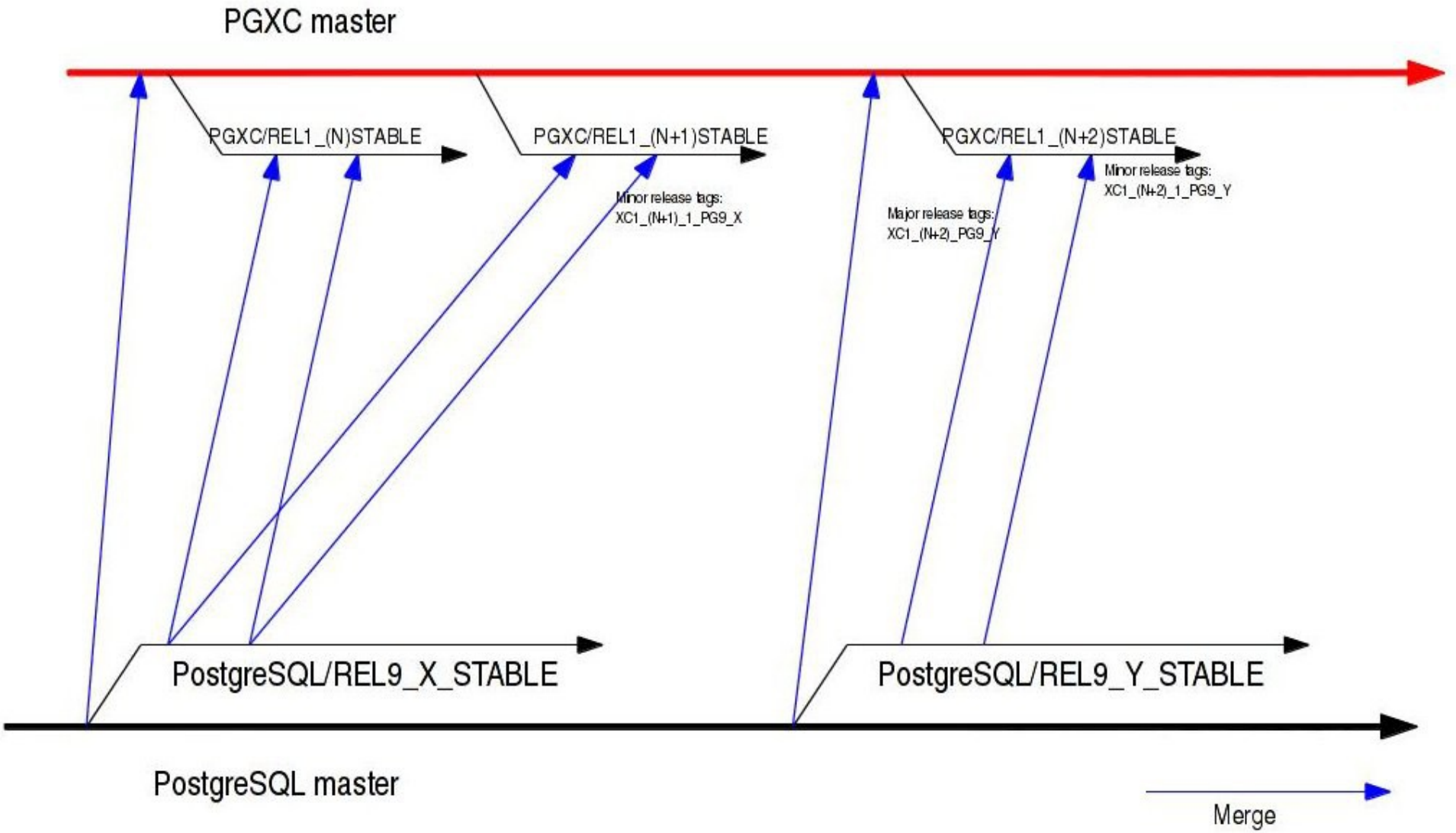
Mailing lists

- All the details in project webpage
  - http://postgres-xc.sourceforge.net/
  - Section Mailing list, with subscription links
  - postgres-xc-XXX@lists.sourceforge.net
- General: postgres-xc-general
- Hackers: postgres-xc-developers
- GIT commits: postgres-xc-committers

Documentation

- Published in http://postgres-xc.sourceforge.net
  - Section Documentation
- Concerned release and branches
  - master (automatically daily uploaded)
  - Stable releases from 1.0

Release policy and code merge



How to contribute

- As a tester - bug report
- As a packager
  - Debian, RPM, pkg...
- As a coder
  - Why not writing new stuff? => Feature requests in SF tracker
  - Bug correction and stabilization? => Bug tracker in SF
- As a documentation correcter
- Anything I am forgetting here...

What next?

- First major release, based on Postgres 9.1 => 1.0
- Next move
  - Merge with Postgres 9.2
  - Data redistribution
  - Node addition and deletion
  - Triggers
  - Global constraints
- 2.0 at the end of April 2013 (?)

Thank You Very Much!

NTT

EnterpriseDB®
The Enterprise PostgreSQL Company