Welcome

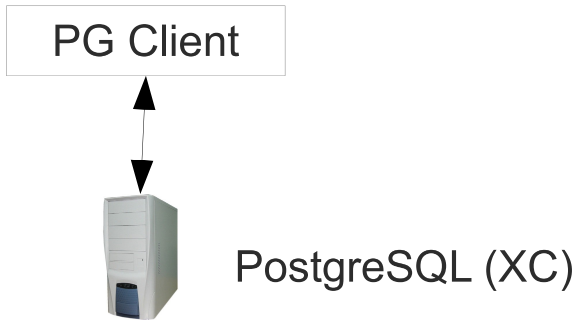
A Shared-nothing cluster system: Postgres-XC

- Amit Khandekar

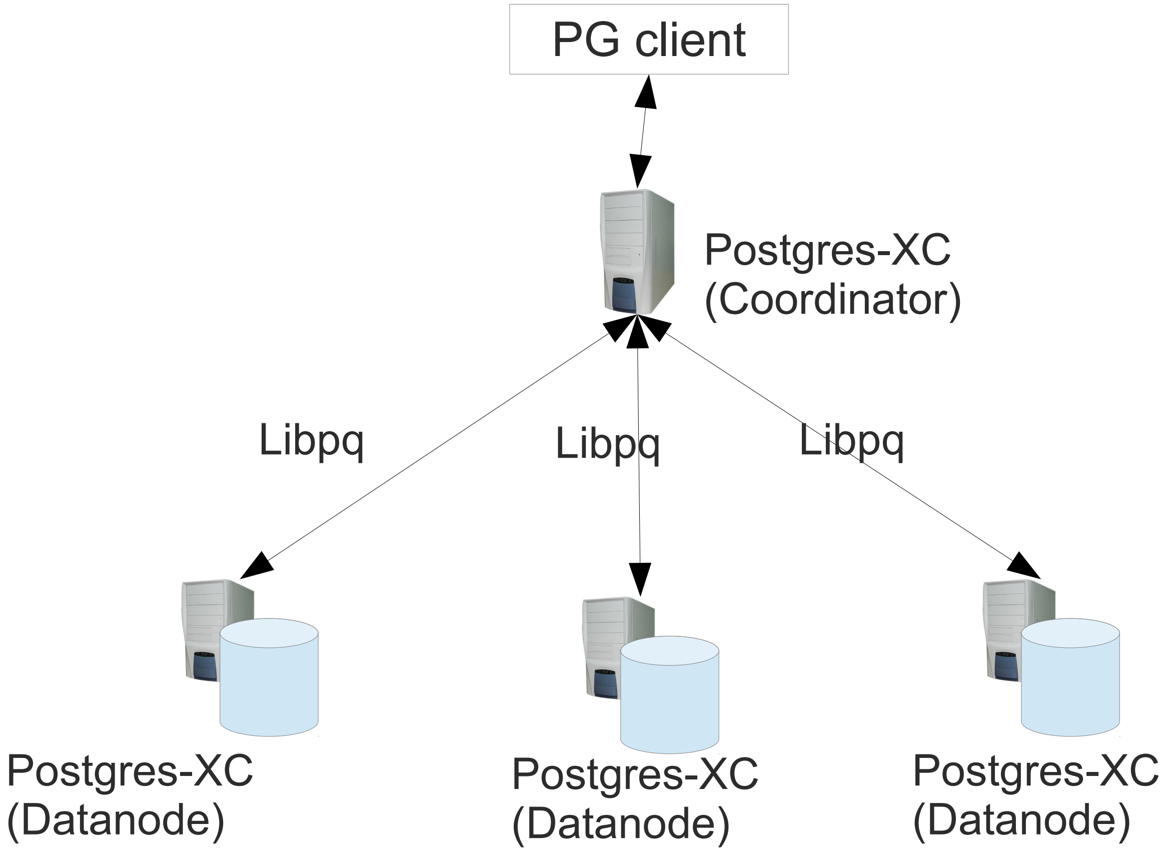
Agenda

- Postgres-XC Configuration
- Shared-nothing architecture applied to Postgres-XC
- Supported functionalities: Present and Future

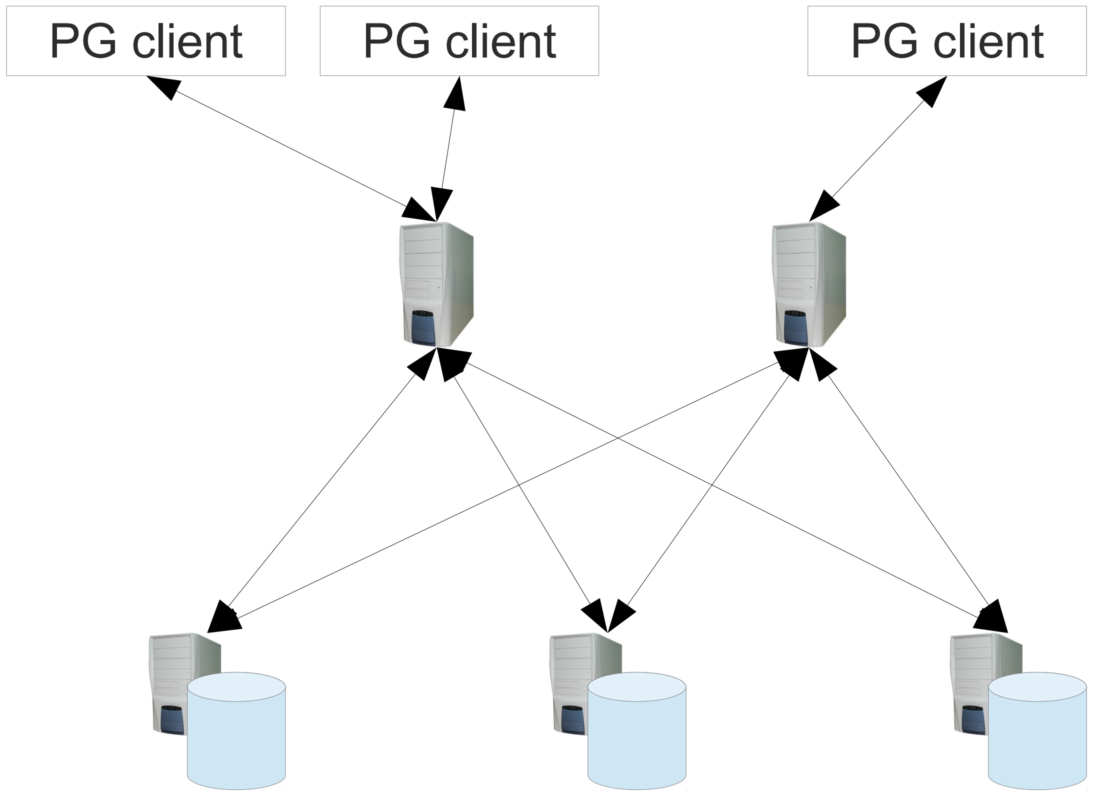
Configuration (User view)



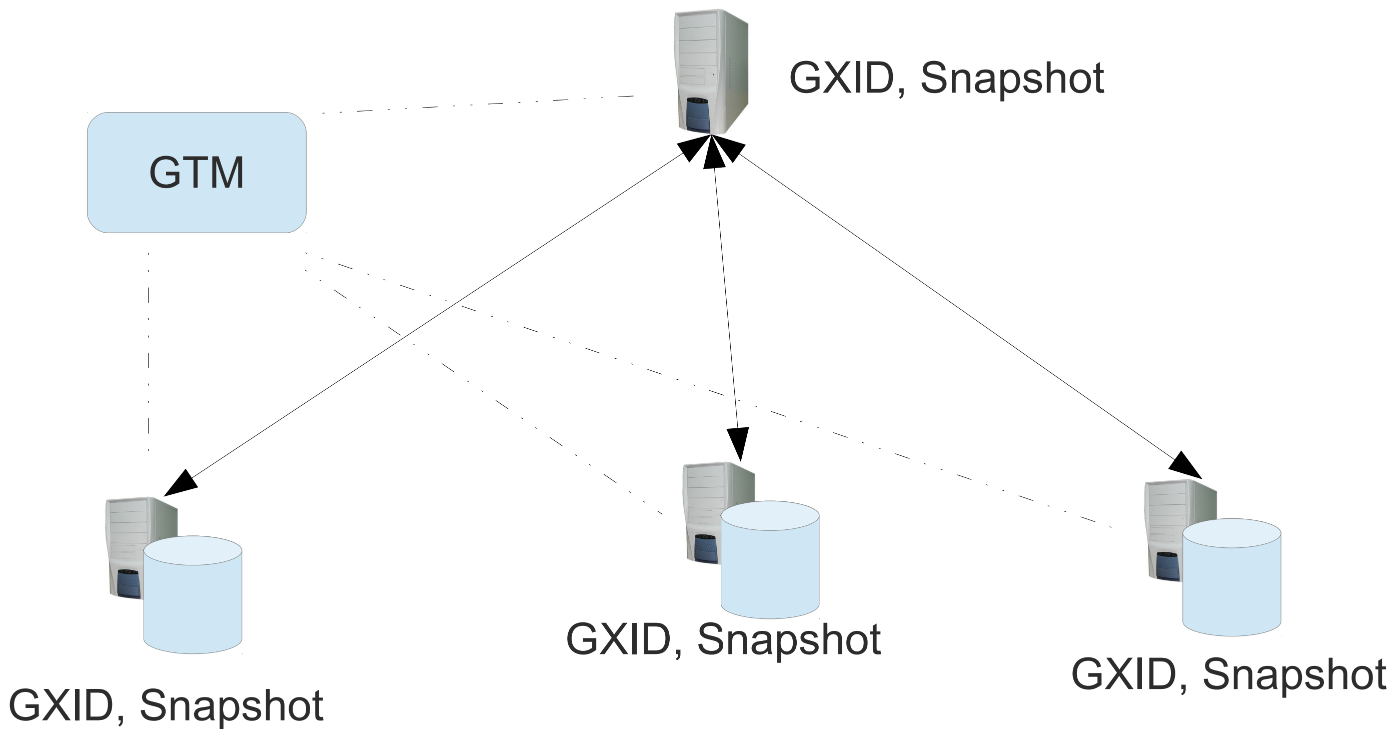
Configuration (Shared-nothing)



Configuration (Synchronous and Symmetric)



What is shared ?



Shared-nothing Cluster

Benefits

- Scalability and performance (no CPU, disk, memory bottleneck)
- Lower Hardware Cost (Commodity hardware)
- Scope for data redundancy (High availability)

Efforts required

- Accessing non-local data
- Implementing distributed data access
- Node addition/removal requires reorganizing database

Shared-nothing : Scalability

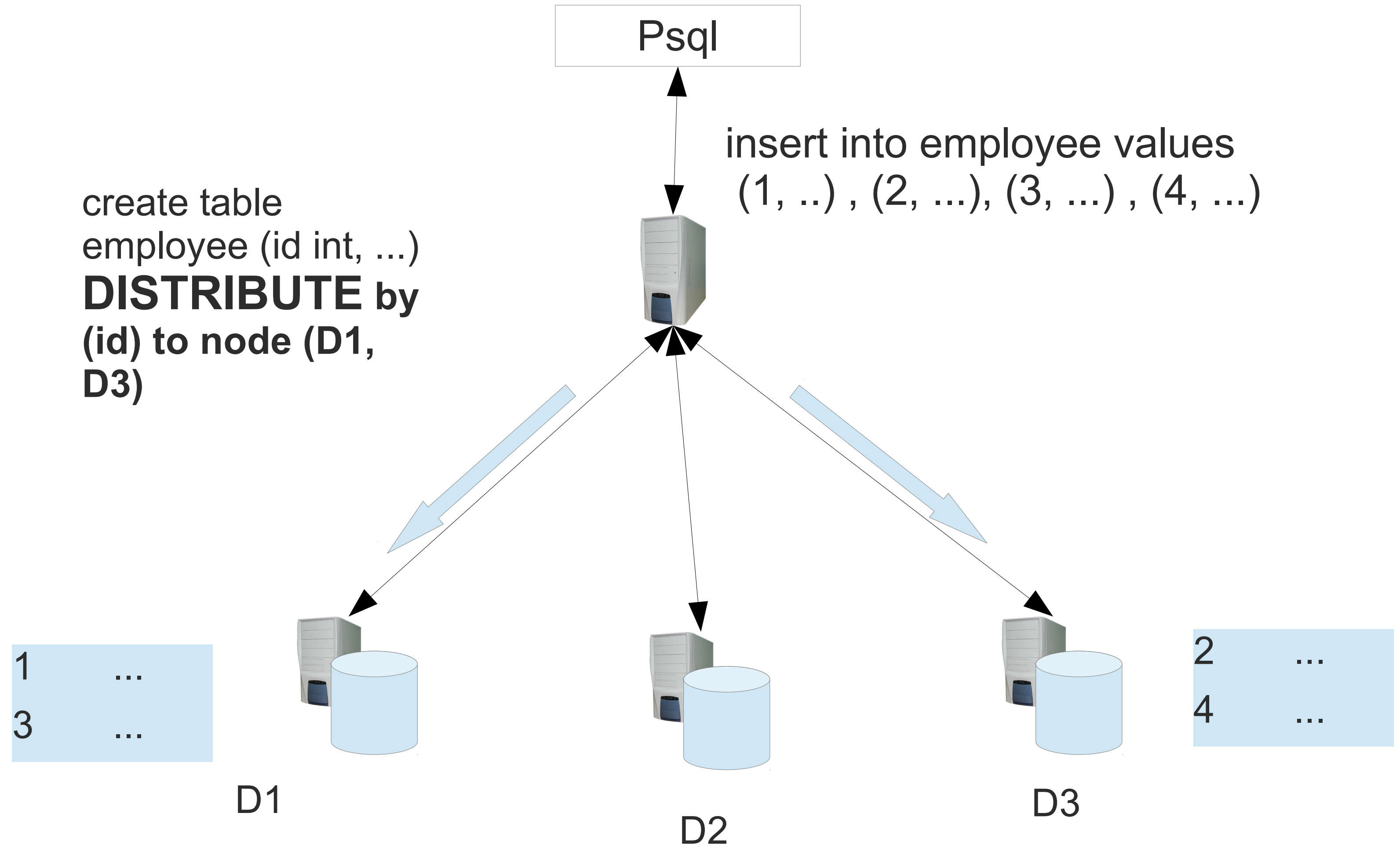
Parallelism

Load-balancing

Data distribution

Data movement

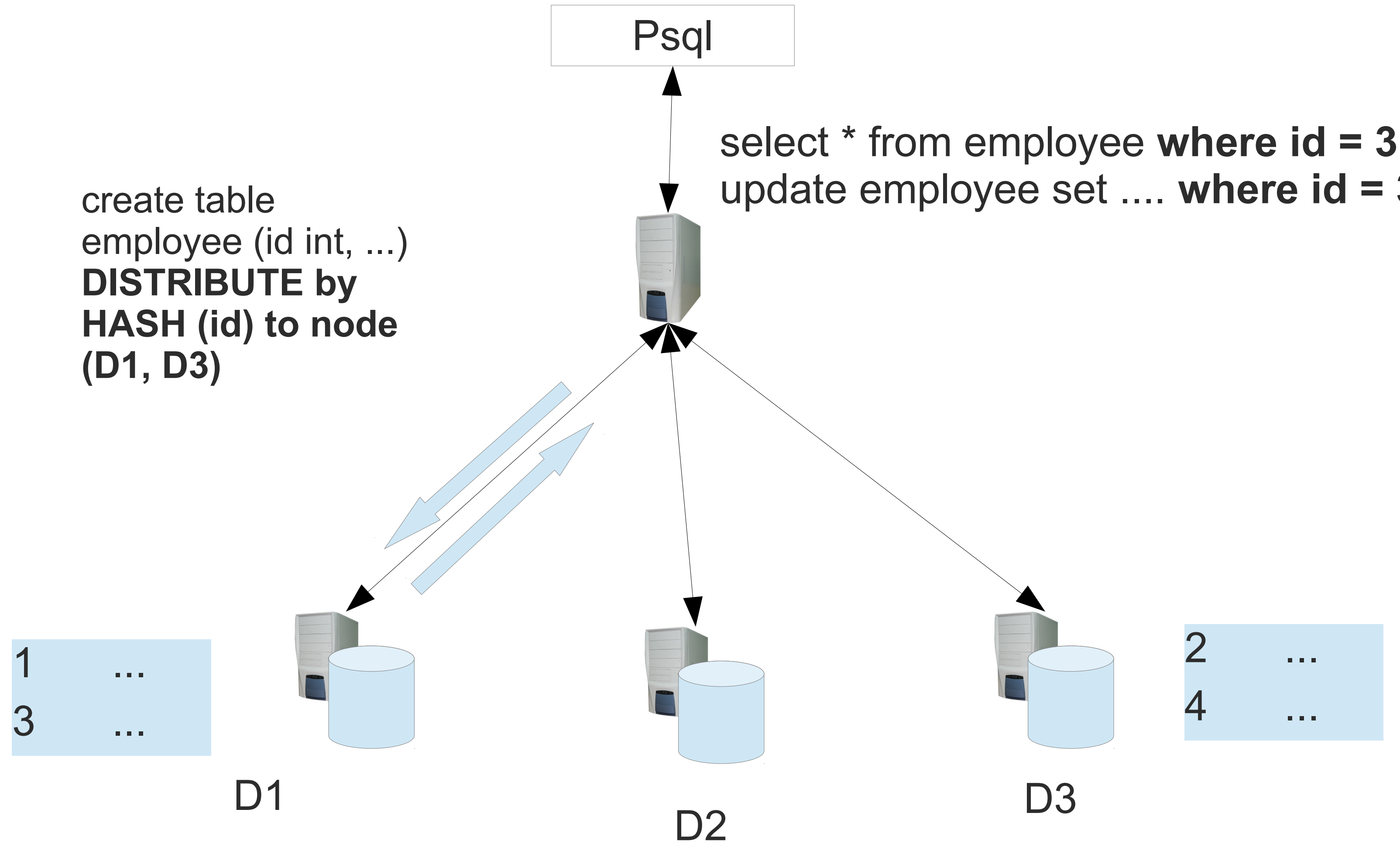
Distributed Tables


Psql
insert into employee values (1, ..) , (2, ...), (3, ...) , (4, ...)
create table employee (id int, ...) **DISTRIBUTE by (id) to node (D1, D3)**
1 ...
3 ...
D1
D2
D3
2 ...
4 ...

Distributed Tables

create table employee (id int, ...) **DISTRIBUTE by HASH (id) to node (D1, D3)**

select * from employee **where id = 3**
update employee set .... **where id = 3**



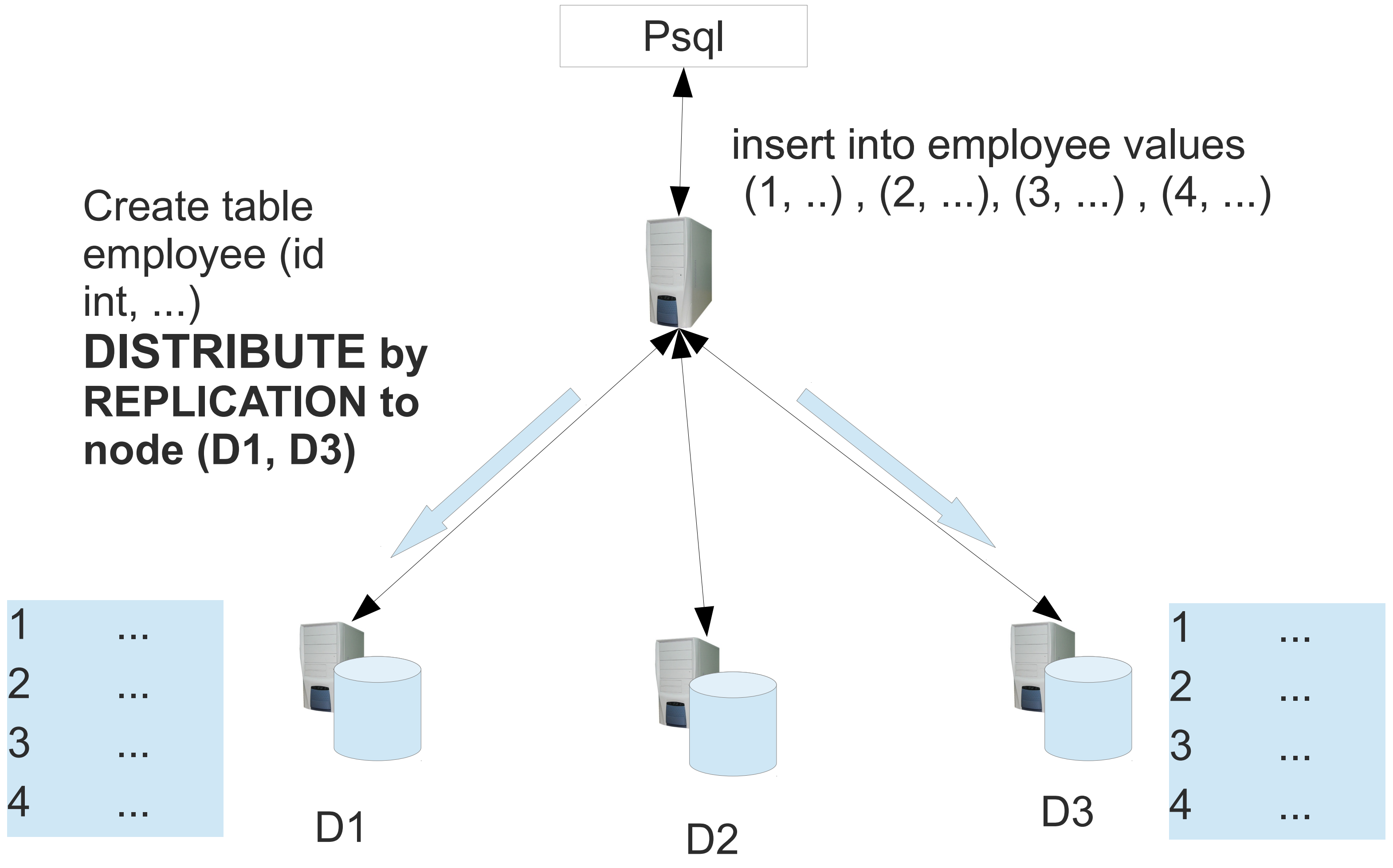
Distributed Tables

- Distribute by HASH
- Distribute by MODULO
- Distribute by Round Robin
- Distribute by Range
- Vertical fragmentation

Replicated Tables

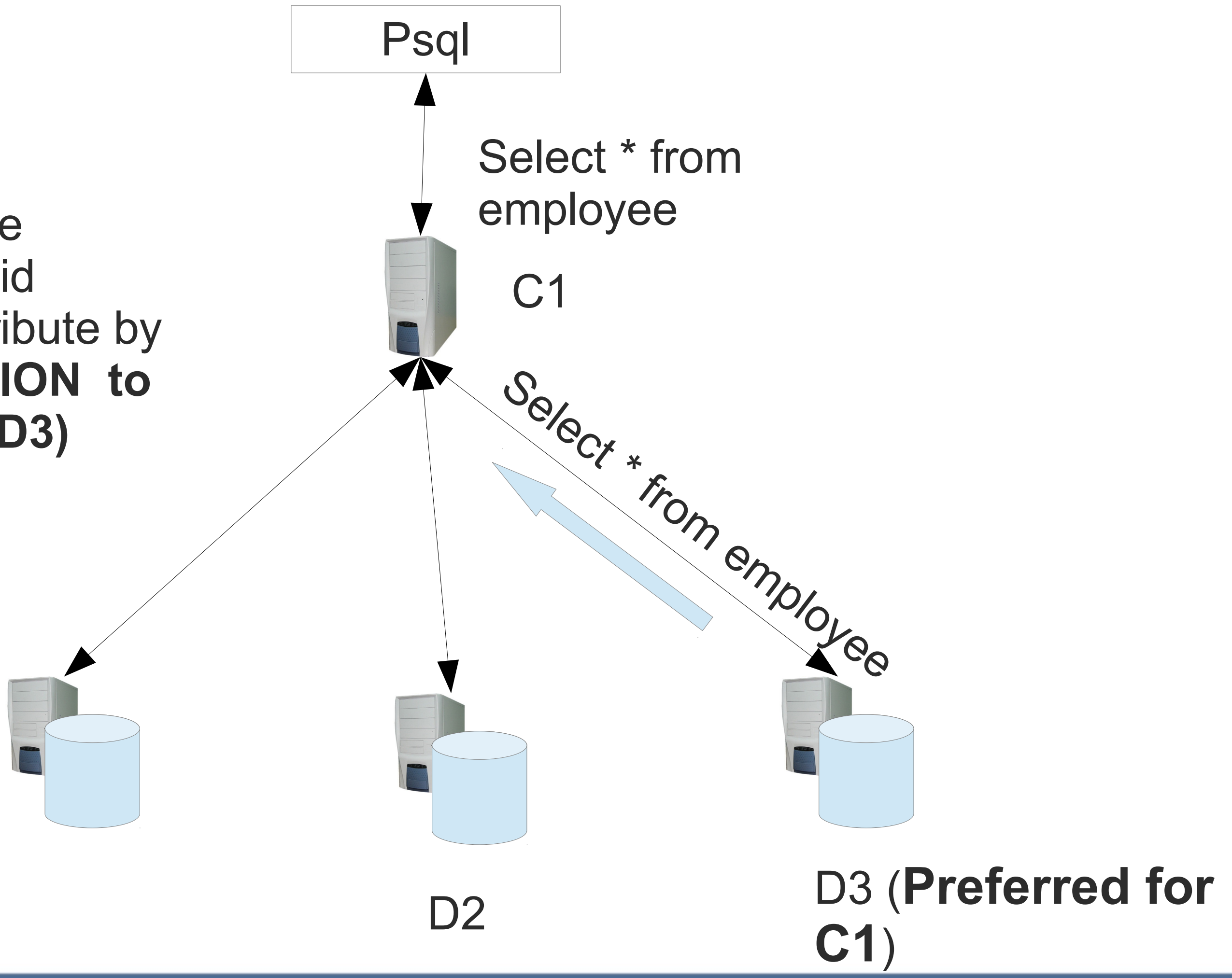
Create table employee (id int, ...) **DISTRIBUTE by REPLICATION to node (D1, D3)**

insert into employee values (1, ..) , (2, ...), (3, ...) , (4, ...)

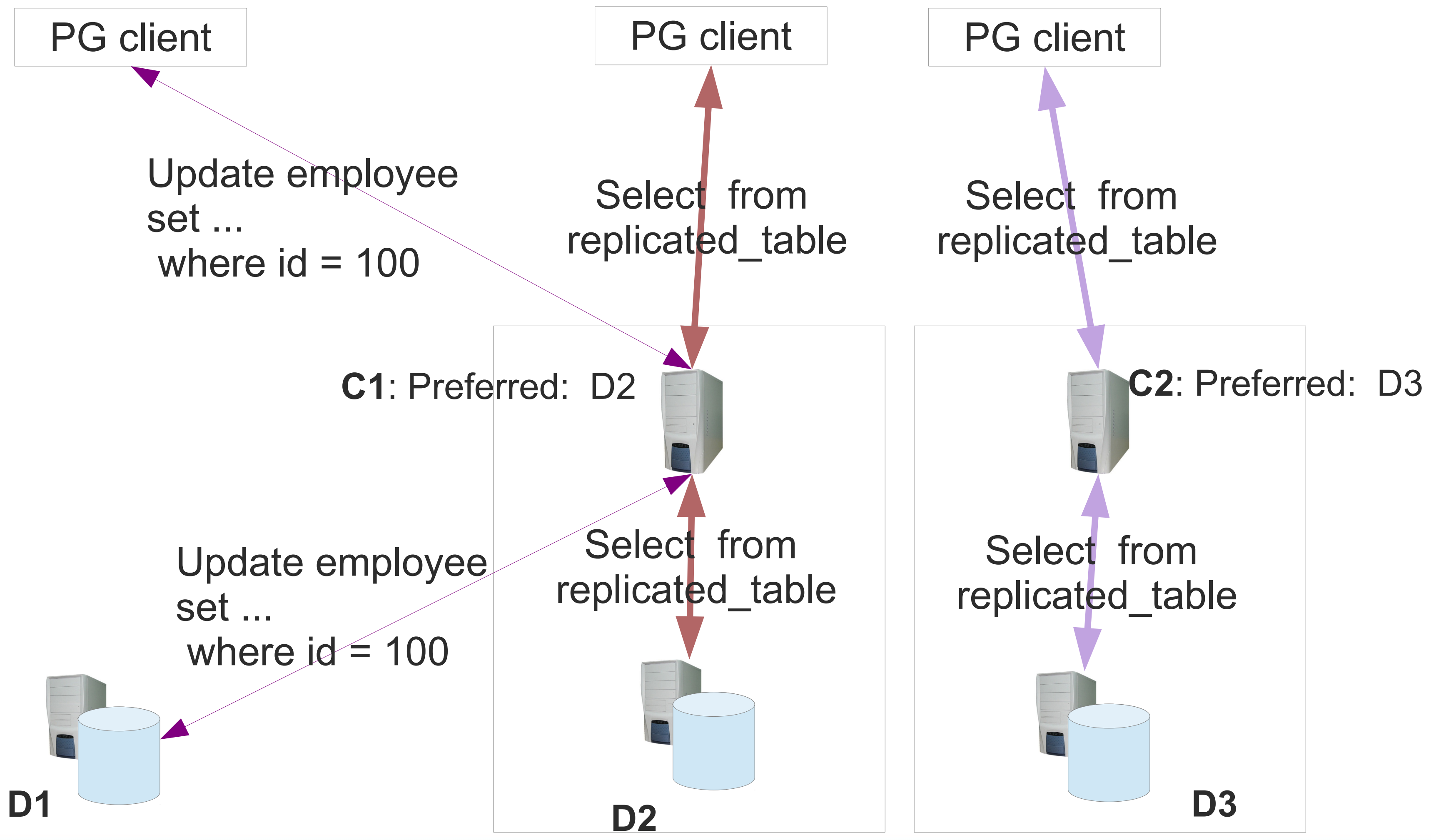


Replicated tables (Preferred node)

Create table employee (id int, ...) distribute by **REPLICATION to node (D1, D3)**



Parallelism : E.g. Configuration


PG client
PG client
PG client
Update employee set ... where id = 100
Select from replicated_table
Select from replicated_table
C1: Preferred: D2
C2: Preferred: D3
Update employee set ... where id = 100
Select from replicated_table
Select from replicated_table
D1
D2
D3

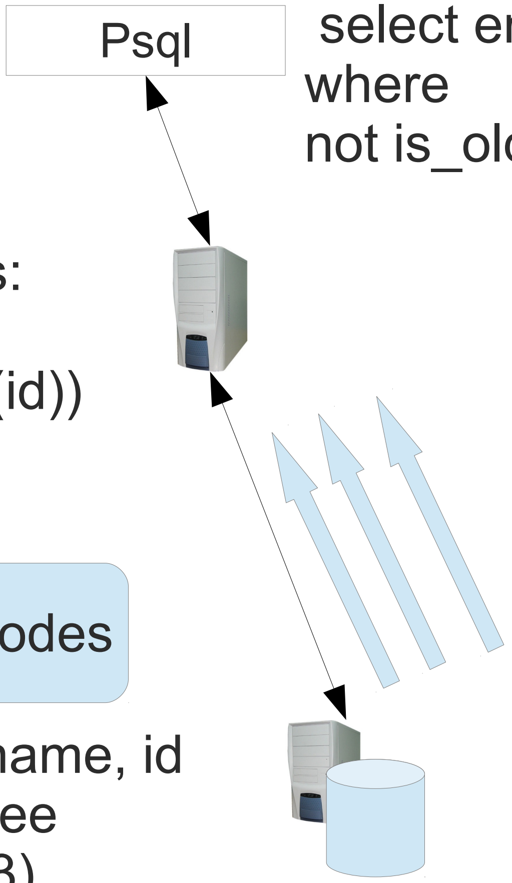
Parallelism

- Inter query
- Intra query
  - Intra plan-node
    - *Single remote table scan done in parallel on datanodes*
  - Inter plan-node
    - *Scope for future work*
    - *Join table scans running parallel*

Load balancing

- Replicated tables offer good load balancing opportunity
- Preferred node for replicated tables
- XC Randomly chooses data node if no preferred node
- DBA: Data distribution
- Coordinator load balancing
  - Requires external application to redirect client requests to particular coordinator.

Reducing data movement



select empname from employee
where
not is_old_employee(id) and id = 3;

Coordinator quals:
(NOT
is_old_employee(id))

Push work to datanodes

SELECT empname, id
FROM employee
WHERE (id = 3)

Reading Plans

```
explain verbose select empname from employee
where not is_old_employee(id) and id = 3;
```

PostgreSQL

```
 Seq Scan on public.employee  (cost=0.00..332.88 rows=4 width=32)
   Output: empname
   Filter: ((employee.id = 3) AND (NOT is_old_employee(employee.id)))
```

Postgres-XC

```
 Data Node Scan on employee  (cost=0.00..0.00 rows=1000 width=32)
   Output: employee.empname
   Node/s: data_node_2
   Remote query: SELECT empname, id FROM ONLY employee WHERE (id = 3)
   Coordinator quals: (NOT is_old_employee(employee.id))
```

Pushing work to datanodes

- Pushable :
  - Immutable functions
  - Constant expressions
  - Join involving at least one common replicated table
  - Whole query in certain scenarios (FQS)

  - Work in progress

Pushing work to datanodes

```
explain select * from employee join dept  on employee.dept = dept.deptid;

                                            QUERY PLAN
--------------------------------------------------------------------------------------------
 Hash Join  (cost=0.12..0.26 rows=10 width=76)
   Hash Cond: (employee.dept = dept.deptid)
   ->  Data Node Scan on employee   (cost=0.00..0.00 rows=1000
width=40)
         Node/s: data_node_1, data_node_2
   ->  Hash  (cost=0.00..0.00 rows=1000 width=36)
         ->  Data Node Scan on dept   (cost=0.00..0.00 rows=1000 width=36)
               Node/s: data_node_1, data_node_2
```

Pushing work to datanodes

```
explain select name from employee join dept  on employee.dept = dept.deptid;

                                        QUERY PLAN
----------------------------------------------------------------------------------------------
 Data Node Scan on "__REMOTE_FQS_QUERY__"  (cost=0.00..0.00 rows=0
width=0)
   Output: employee.name
   Node/s: data_node_1
   Remote query: SELECT employee.name FROM (employee JOIN dept ON
((employee.deptid = dept.deptid)))
(4 rows)
```

Cost estimates

- Future work
- cost estimation is not cluster-aware
  - Data transfer cost not calculated.
- No selectivity info on coordinator
  - ANALYZE command updates stats on datanodes.
    - *Does not update on coordinator.*
- Cheapest plan not chosen
- Datanodes have the usual PG cost estimation

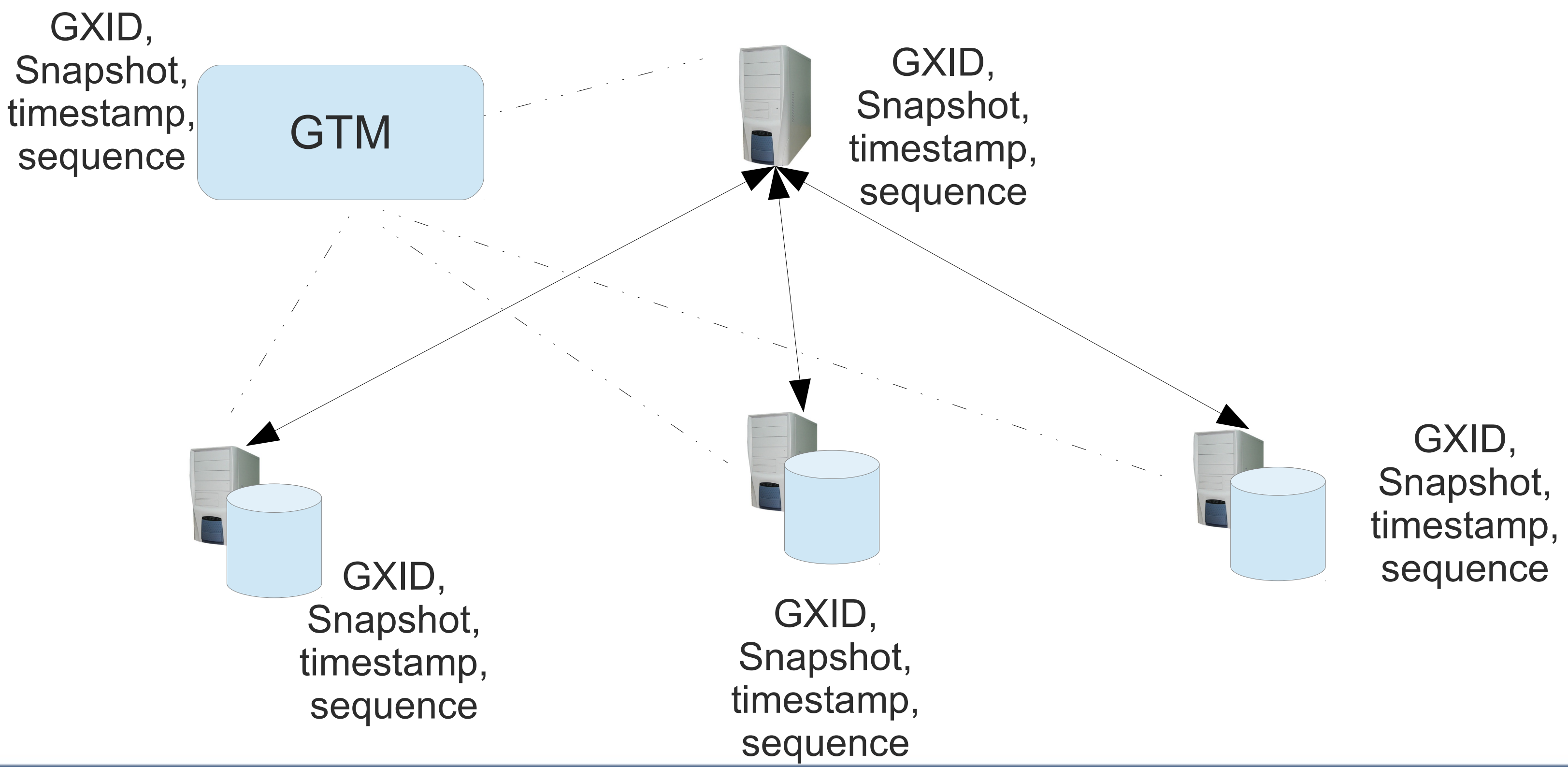
Deadlocks

- No cluster-wide deadlock detection
- Updates on replicated tables : deadlocks more likely
  - Two parallel updates on same row of replicated table
    - Q1 has row lock on node1, Q2 has row lock on node2
    - Now Q1 waits on Q2 lock on node2, and Q2 waits on Q1 lock on node1
  - Assign same primary data node on each coordinator
    - *Might even not need to do this in the future*

ACID Properties

ACID properties (Consistency)

- Consistent view of database throughout the cluster using Global transaction ID, and Global Snapshot
- MVCC takes care of the rest.



ACID properties (Consistency)

- Global Constraints not supported yet
  - Constraint check is done only on individual node; not done across datanodes.
  - Hence, attempt to create table with a constraint that requires cluster-wide constraint check is not allowed.
  - E.g. distributed table not allowed to have unique constraint on a column unless that column is distribution key, etc.
  - Will keep this restriction until we support global constraint check.
- Updating distribution key column not supported
  - TIP: Explicitly choose distribution key while creating table

ACID properties (Isolation)

- Transaction isolation
  - read committed
  - repeatable read
  - serializable (>= 9.1) falls back to repeatable read

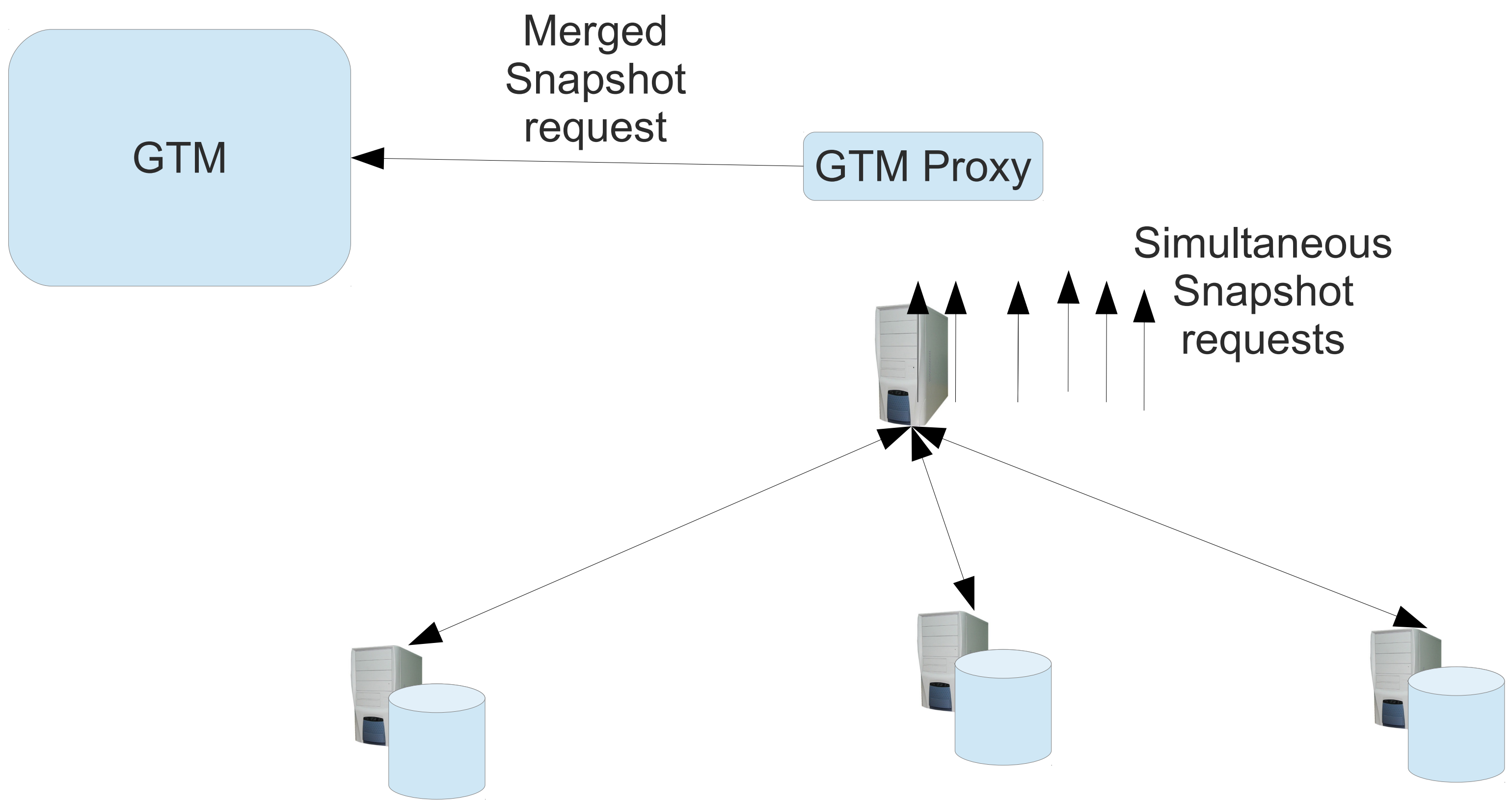
ACID properties (Atomicity and durability)

- Two-phase protocol
  - Coordinator uses this transparently on nodes involved in write activity.
  - This ensure either all nodes commit, or all nodes abort the transaction; even if a node crashes.
  - Always used when explicitly requested from application using PREPARE TRANSACTION
  - Needs to be disabled if temp tables are involved: PG restriction.
    - *set enforce_two_phase_commit = off*
  - Because datanodes are PostgreSQL-based servers, datanodes have their own CLOG, so can be individually recovered after a crash.

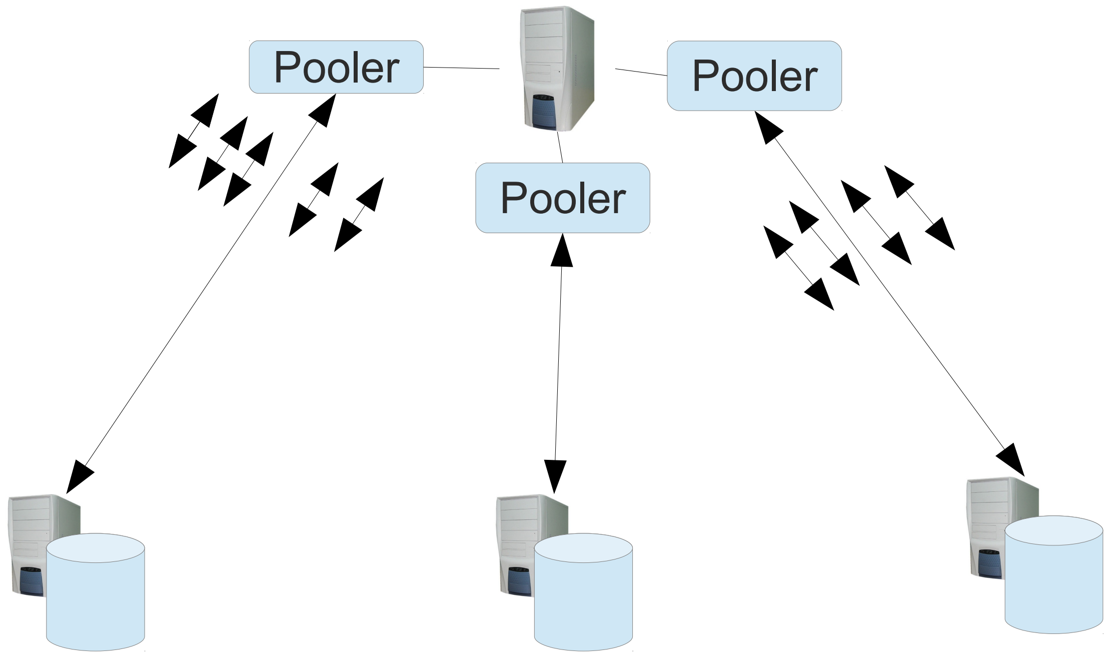
ACID properties (Durability)

- pg_prepared_xacts
  - All nodes have executed PREPARED TRANSACTION
  - Coordinator is about to send abort/commit when a node crashes
  - pg_prepared_xacts will show such transactions
- pgxc_clean utility
  - Cleans up such transactions on the nodes that are recovered
  - Issues COMMIT PREPARED

Bottlenecks



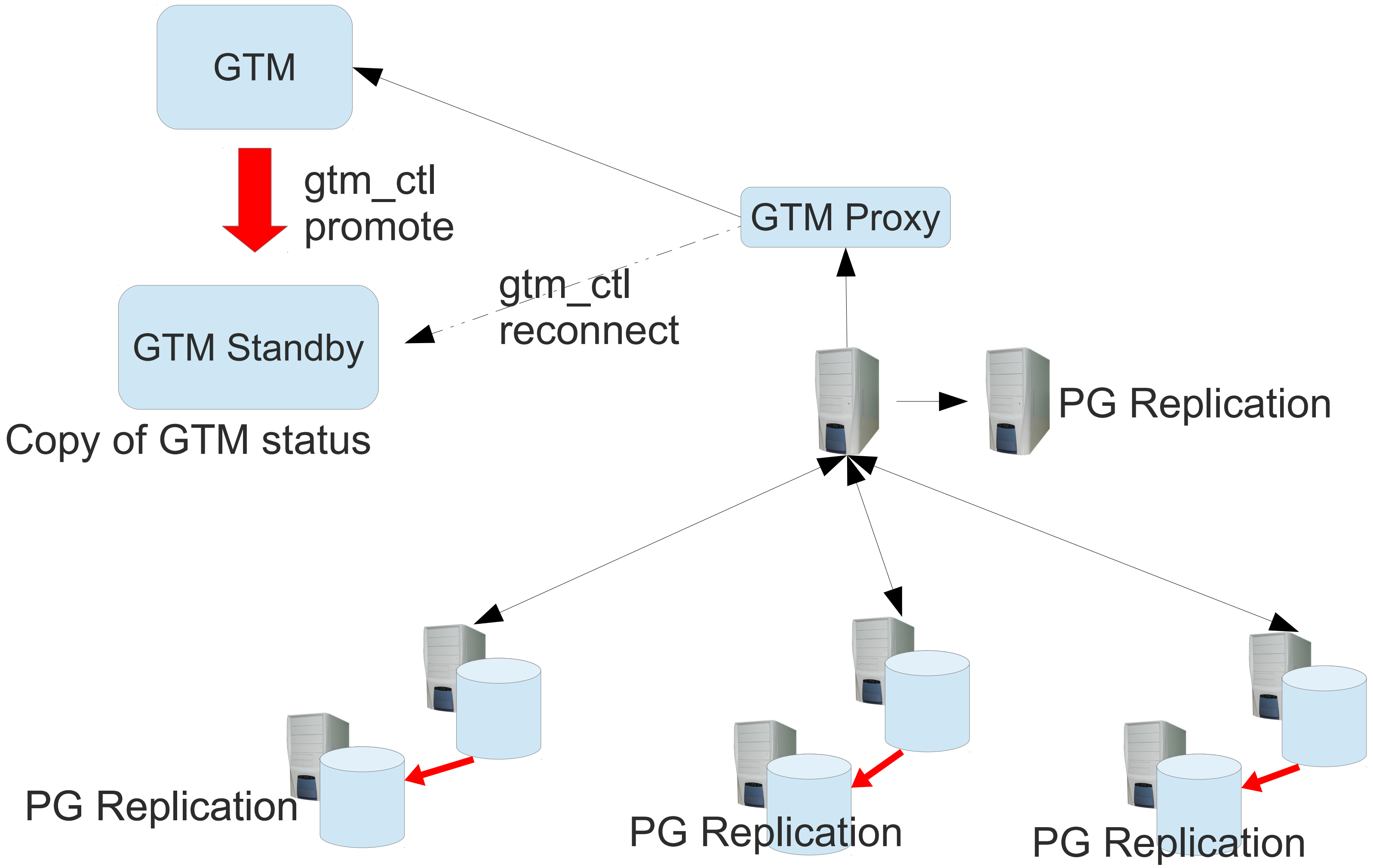
Bottlenecks



High availability : In-built ?

- Redundancy possible using replicated tables
- Queries not accessing failed node keep on executing
- If a node having all replicated data crashes, data is available on other nodes
  - but it is not HA : coordinator does not automatically failover to other replicated node.
- Scope for further research

High availability



High availability

- For automatic failover, integrate Postgres-XC with HA middleware such as Pacemaker.
  - Continuously monitor each component including GTM, coordinator and datanode
  - Write Pacemaker resource agents for Postgres-XC
    - *Implement start, stop, status, promote, etc*
  - May still need manual intervention
    - *ALTER NODE for new IP.*
    - *pgxc_clean()*
- Linux-HA Japan team actively working for the above

Recovery

For PITR, the whole cluster should be recovered upto the same point on all nodes

- CREATE BARRIER 'barrier_id' from any coordinator
  - *Waits for all the transactions to complete*
  - *Creates an XLOG entry for barrier recovery on each node*
- In recovery.conf, set recovery_target_barrier 'barrier_id', just like we set recovery target xid or timestamp
- Recovery takes place by rolling forward the xlog up to this point: 'barrier_id'

Catalog objects

- Queries on catalogs are always run locally.
- All nodes have the same copy of catalogs.
  - DDL statements are propagated to all nodes.
- Views/Rules
  - Rule rewrite happens on coordinator
- Sequences
  - Fetched from GTM
- User Functions
  - Definitions are everywhere. Coordinator chooses whether it should be called on datanode.
- System tables
  - Has local information.
- Triggers (Under development)

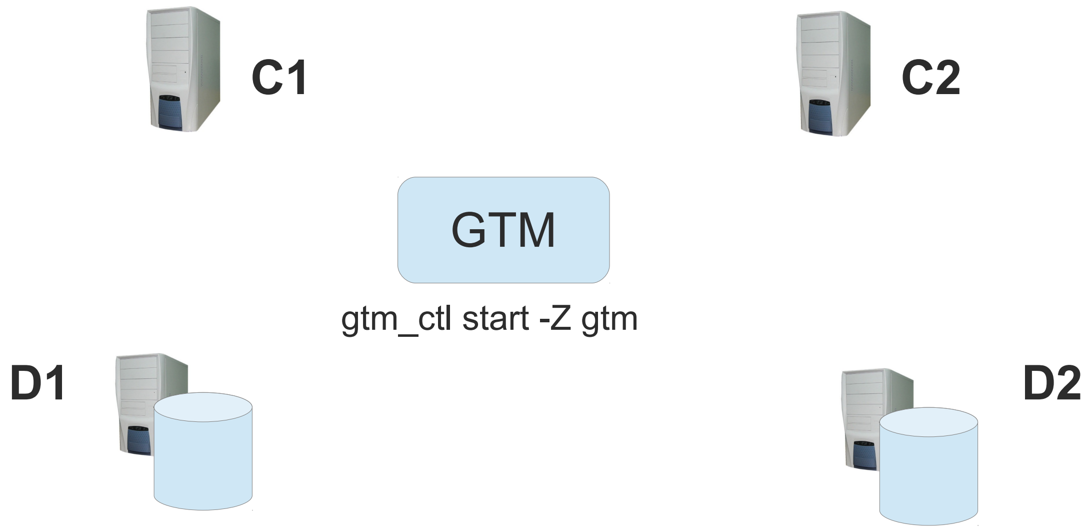
Cluster initialization

CREATE NODE **C2** WITH (HOST = '238.12.34.11', type = 'coordinator');

CREATE NODE **D1** WITH (HOST = 'localhost', type = 'datanode', **preferred**);
CREATE NODE **D2** WITH (HOST = '238.12.88.11', type = 'datanode');

CREATE NODE **C1** WITH (HOST = '238.12.34.12', type = 'coordinator');

CREATE NODE **D1** WITH (HOST = '238.12.88.12', type = 'datanode');
CREATE NODE **D2** WITH (HOST = 'localhost', type = 'datanode', **preferred**);



```
gtm_ctl start -Z gtm
```

```
initdb ... –nodename=D1;
pg_ctl start -Z datanode
```

```
initdb ... –nodename=D2;
pg_ctl start -Z datanode
```

Cluster management

- Each coordinator needs to run CREATE NODE for all other nodes including other coordinators.
- Node configuration is static. Should be changed offline.
  - pg_dump from any one coordinator
  - Stop cluster, add and reinitialize all nodes again, including new node
  - pg_restore on any coordinator
- Online node addition/removal (TODO)

Cluster management

- Online data redistribution
  - Used to change distribution strategy
    - ALTER TABLE tab1 DISTRIBUTE BY REPLICATION ...
  - Can also be used to redistribute the data onto newly added nodes.

- Online data redistribution concurrently (TODO)
  - ALTER TABLE … CONCURRENTLY

Features support (< 1.0)

- Postgres-XC 0.9.6
  - HAVING clause
  - GROUP BY optimization for pushing down
  - Temporary objects
  - PREPARE/EXECUTE
- Postgres-XC 0.9.7
  - Cluster node management with DDLs
  - SELECT INTO/CREATE TABLE AS
  - INSERT … SELECT
  - Window functions
  - Views, correlated subquery, Common table expression

Features support (>= 1.0)

- Postgres-XC 1.0
  - Based on PostgreSQL 9.1
  - Stabilization
  - SERIAL types
  - TABLESPACE
  - Advisory locks
  - Fast Query Shipping
  - Cursors
- Development branch
  - Merged with PostgreSQL 9.2
  - Data redistribution with ALTER TABLE
  - Planner improvements
  - RETURNING clause
  - WHERE CURRENT OF
  - TRIGGERS

Future

- Online node addition and removal
- Ongoing query processing improvements
- SAVEPOINT
- Serializable Snapshot Isolation
- HA improvements

… and many others

Thank you

- Project Web Page:
  - http://postgres-xc.sourceforge.net/
- Help at:
  - postgres-xc-general@lists.sourceforge.net
  - postgres-xc-developers@lists.sourceforge.net