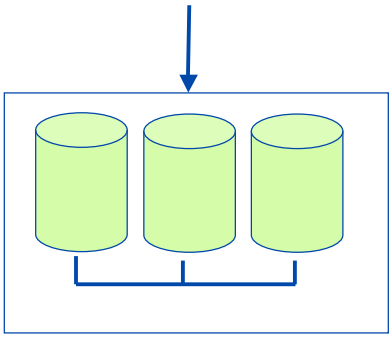
Postgres-R(SI) Data Replication and Snapshot Isolation

Shuqing Wu

McGill University

Montreal, Canada

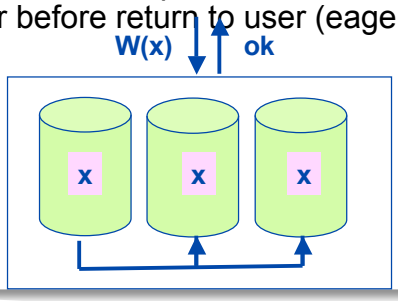
Example: Cluster Replication



- Cluster of DB replicas
- Read-one-Write-All-Available
- Performance
  - Distribute Load
  - Scale
- Fault-Tolerance

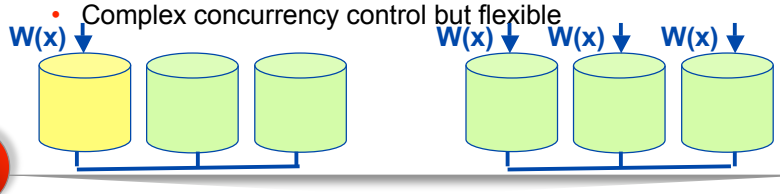
Typical Replica Control

- Keep copies consistent
  - Execute each update first at one replica
  - Return to user once one update has succeeded
  - Propagate updates to other replicas or execute updates at other replicas either after return to user (lazy) or before return to user (eager)



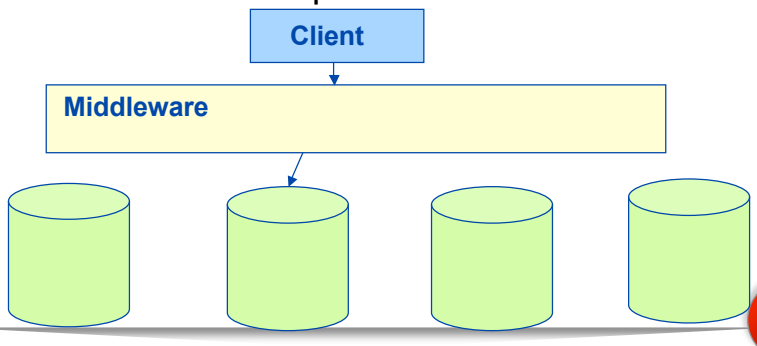
Replica Control

- Challenge
  - Guarantee that updates succeed at all replicas
  - Combine with concurrency control
- Primary Copy
  - Updates executed at primary and then sent to secondaries
  - Secondary only accept queries
  - Easry concurrency control
- Update Everywhere
  - Each replica accepts update requests and propagates changes to others
  - Complex concurrency control but flexible



Middleware based

- Client submits operations (e.g., SQL statements) to middleware
- Middleware coordinates database replicas and where which operations are executed

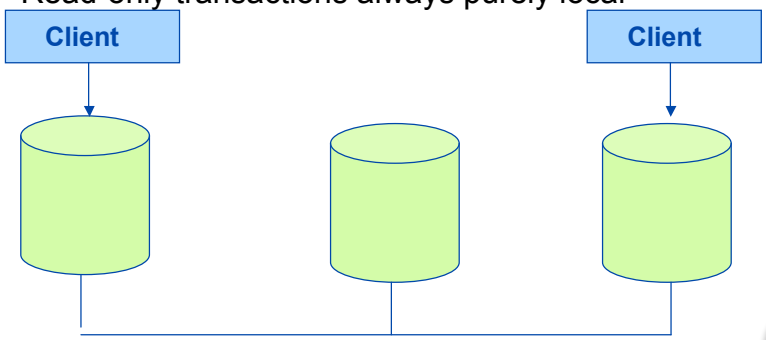


Middleware based

- Many research prototypes, some commercial systems
- Pro
  - No changes to DBS
  - Heterogeneous setup possible
- Cons
  - Each solutions has particular restriction. For example
    - Transactions must be declared read-only/update in advance (primary copy approaches)
    - Transaction must declare all tables to be accessed in advance
    - Operations cannot be submitted one by one but in a bunch
    - Update must be executed on all replicas before confirmation is returned to user
  - Each operation filtered through the middleware
    - Inappropriate in any WAN communication (e.g., client/middleware)

Kernel Based

- Each client connected to one replica
- Transaction executed locally
- Local replica propagates changes to other replicas
- Read-only transactions always purely local

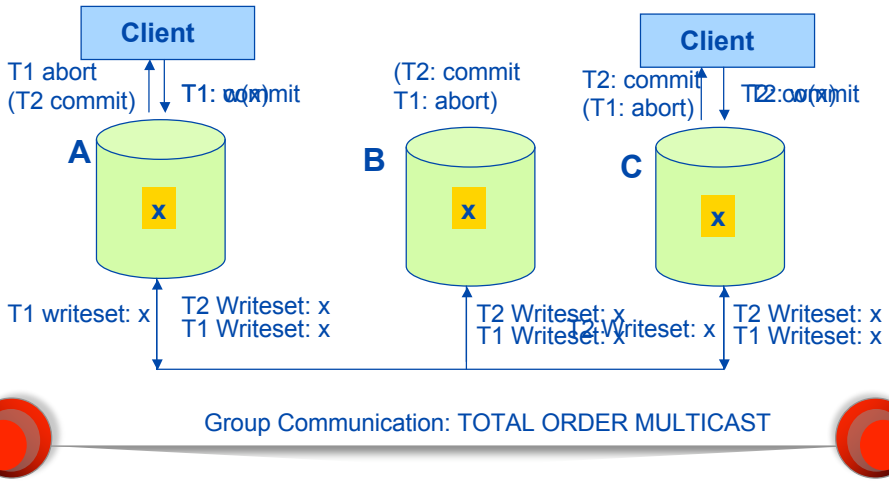


Kernel Based

- Many lazy solutions (commercial / research):
  - Data of different replicas might be stale/inconsistent for considerable time
- Few eager solutions
- Pro
  - No restrictions on transactions
  - No indirection through middleware
  - Everything comes as one package
- Cons
  - Challenge of integration of replica control and DBS based concurrency control
- The solution here is kernel based

Postgres-R (VLDB 2000): update everywhere

- Use total order multicast to resolve conflicts



Execution Overview

- Local Phase:
  - Transaction executes locally at one replica
- Send Phase:
  - At commit time, writeset is multicast to all replicas in total order
- Decision Phase:
  - Decide whether conflicts and abort/commit accordingly
  - VLDB-2000: Based on total order combined with strict 2-phase-locking
  - Requires additional messages

Snapshot Isolation

- New Challenge with PostgreSQL version 7/8
  - Version 6: strict 2-phase locking and serializability
  - Since version 7: multi-version concurrency control providing snapshot isolation
- Very interesting since snapshot isolation provided by more and more database systems (Oracle, new SQL server)
- Principles
  - Reads always read a committed snapshot as of time of transaction starts
  - If two concurrent transactions want to update the same data object, only one may succeed, the other aborts
- Implemented via combination of multi-version (for read) and locking+optimistic concurrency control (for write)

PostgreSQL: record versions

- For simplicity: only look at SQL update
- Each update on record
  - Invalidates valid version of record
  - creates new record version
- Each version tagged with
  - tmin: Transaction identifier (tid) that created version
  - tmax: tid that invalidated version
- Valid version at given timepoint
  - tmin is from committed transaction
  - tmax is NIL or from aborted or active transaction

X1	t3	t5	values	X2	t5	Nil	values

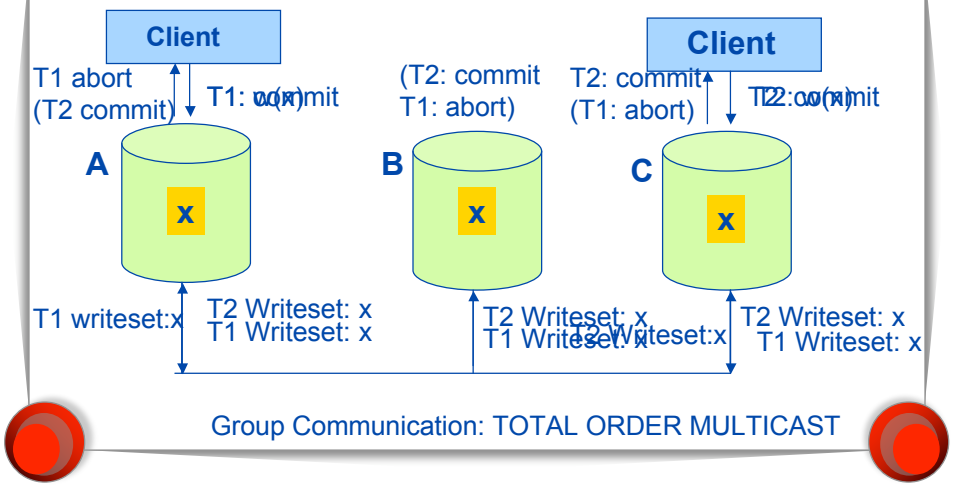
PostgreSQL: reads

- Upon read on x by T
  - Read committed vesion as of start time
  - tmin committed before T started
  - tmax NIL or aborted or committed after T started
- Example
  - At start of T: T3 committed, T5 still running
  - T reads version x1 created by T3

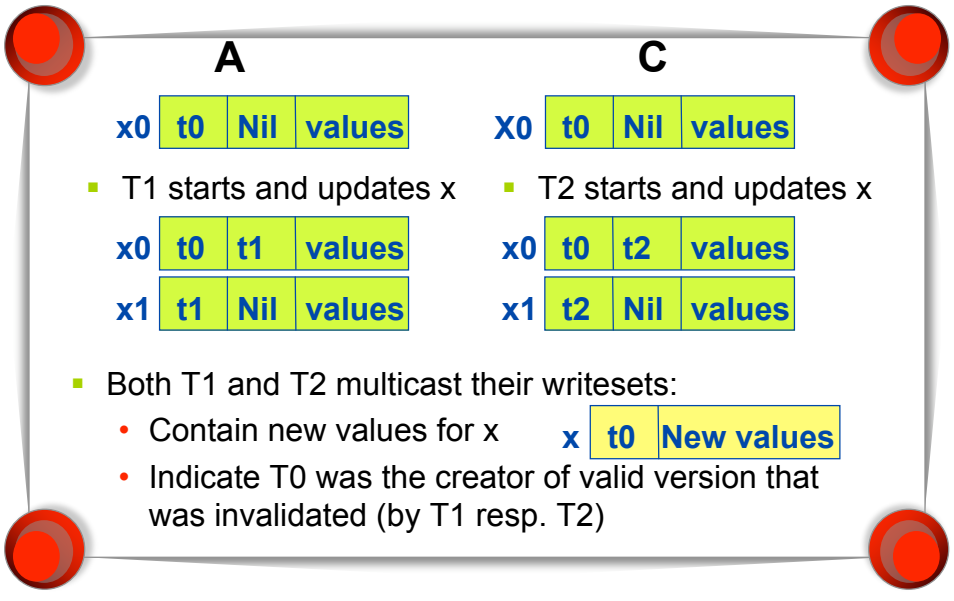
PostgreSQL: writes

- Example 1:
  - At start of T: T3 committed, T5 running
  - T5 commits after T starts
  - T gets lock and valid version x2
  - T aborts since concurrent to T5
- Example 2:
  - At start of T: T3 and T5 committed
  - T gets lock and valid version x2
  - T5 committed before T started, so T succeeds
  - T invalidates X2 and creates new version x3

Postgres-R: Recall Example execution



Local Phases at A/C



Execution at A

- Writeset of remote T2 arrives and T2 starts
  - Conflict Check: Get valid version of x and compare tmin with creator id T0 in writeset

x0	t0	t1	values

x	t0	New values

  - No conflict
  - Request lock on x
  - Local transaction T1 holds lock
  - Force T1 to abort
  - Get lock
  - Apply change
  - Commit

x0	t0	t2	values
x1	t1	Nil	values
x2	t2	Nil	values

- Writeset of local T1 arrives and is discarded

Execution at C

- Writeset of local T2 arrives
  - T2 is still alive => commit T2

x0	t0	t2	values
x1	t2	Nil	values

- Writeset of remote T1 arrives and T1 starts
  - Conflict Check: Get valid version of x and compare creator id with creator id T0 in writeset

x1	t2	Nil	values

x	t0	New values

  - Detect conflict
  - Abort T1
  - Note this conflict is detected although T1 and T2 do not run concurrently at C

Execution at B

- Writeset of remote T2 arrives and T2 starts
  - Conflict check: Get valid version of x and compare creator id with creator id T0 in writeset

x0	t0	Nil	values

x	t0	New values

  - No conflict
  - get lock on x
  - Apply change
  - Commit

x0	t0	t2	values
x1	t2	Nil	values

- Writeset of remote T1 arrives and T1 starts
  - Conflict check: Get valid version of x and compare creator id with creator id T0 in writeset

x1	t2	Nil	values

x	t0	New values

  - Detect conflict
  - Abort T1

Challenges

- Make same abort/commit decision at all replicas
  - although start and commit at different physical times at different replicas
  - although different tids at different replicas
    - Each database generates its own tids
    - They are independent

Solution Outlines

- Serial application of writesets
  - Receiving writesets, version check (if remote), applying writesets (if remote), and committing is serial one writeset after the other

Solution Outlines

- Global vs. Local Transaction identifiers
  - At begin, a transaction receives a local identifier
  - tmin and tmax in versions are local (not change PostgreSQL)
  - At writeset delivery, transaction receives global identifier (according to global multicast order)
  - Piggyback and perform check on global identifiers
  - Efficient matching of tid/gid

Implementation Challenges

- Writesets contain more than one transaction
  - Careful when aborting local transactions
- Aborting local transactions
  - Signal based
  - Has to consider many critical sections in which transactions may not be aborted
- PostgreSQL's locking scheme for updates
- Failures and Recovery

TPC-W benchmark

- Small configuration
- Browsing with 80% reads, 20% writes
- 40 concurrent clients
- The more servers, the better performance for both queries and update transactions

Update intensive

- Few concurrent clients:
  - Centralized system better
- Many concurrent clients
  - Replicated system better since client management distributed

Current Status

- Variation of this protocol currently implemented by developers sponsored by
  - Afilias Inc. (.net/.info domain name registry)
  - RedHat
  - Fujitsu
  - NTT Data corporation
  - And others
- Beta version to be expected in couple of months