An Evaluation of Distributed Concurrency Control

Harding, Aken, Pavlo and Stonebraker

Presented by: Thamir Qadah
For CS590-BDS
Outline

● Motivation
● System Architecture
● Implemented Distributed CC protocols
  ○ 2PL
  ○ TO
  ○ OCC
  ○ Deterministic
● Commitment Protocol
  ○ 2PC
  ○ Why CALVIN does not need 2PC
    ■ What is the tradeoff
● Evaluation environment
  ○ Workload Specs
  ○ Hardware Specs
● Discussion
  ○ Bottlenecks
  ○ Potential solutions
Motivation

● Concerned with:
  ○ When does distributing concurrency control benefit performance?
  ○ When is distribution strictly worse for a given workload?

● Costs of distributed transaction processing are well known [Bernstein et. al ‘87, Oszu and Valduriez ‘11]
  ○ But, in cloud environments providing high scalability and elasticity, trade-offs are less understood.

● With new proposals of distributed concurrency control protocols, there is no comprehensive performance evaluation.
<table>
<thead>
<tr>
<th>Publication</th>
<th>Lock</th>
<th>TS</th>
<th>MV</th>
<th>OCC</th>
<th>Det</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tango [7]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spanner [20]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Granola [21]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centiman [25]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>FaRM [26]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Warp [27]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>MaaT [39]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rococo [41]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ren et al. [45]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1 [47]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Calvin [54]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Wei et al. [58]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>TaPiR [61]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lynx [62]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Deneva (this study)</td>
<td>✓×2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Publication</td>
<td>Lock</td>
<td>TS</td>
<td>MV</td>
<td>OCC</td>
<td>Det</td>
<td>None</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td>Tango [7]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Spanner [20]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Granola [21]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Centiman [25]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>FaRM [26]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Warp [27]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>MaaT [39]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Rococo [41]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ren et al. [45]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>F1 [47]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Calvin [54]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Wei et al. [58]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TaPiR [61]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lynx [62]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deneva (this study)</td>
<td>✓ × 2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Note:** Lock-based implementations may be different (e.g. deadlock detection/avoidance)
Transaction Model

- Deneva uses the concept of stored procedures to model transactions.
  - No client stalls in-between transaction logical steps

- Support for protocols (e.g. CALVIN) that require READ-SET and WRITE-SET to be known in-advanced
  - DBMS needs to compute that.
    - Simplest way: run transaction without any CC measures
High Level System Architecture

**Hosted Cloud Infrastructure**
- Client Process 1 (Hosted Instance A)
- Client Process 2 (Hosted Instance B)
- Client Process 3 (Hosted Instance C)
- Server Process 1 (Hosted Instance D)
- Server Process 2 (Hosted Instance E)
- Server Process 3 (Hosted Instance F)

**Server Process**
- Input messages
- Protocol-specific state (e.g., lock table)

**Multi-Core Execution Engine**

**Local, In-Memory Data Storage**
High Level System Architecture

Client and Server processes are deployed on different hosted cloud instance
Communication among processes uses `nanomsg` socket library
- I/O threads responsible for handling marshaling and unmarshaling transactions, operations, and return values.
- Operations of active transactions are prioritized over new transactions from clients.

**Server Process**
- I/O Threads
- Priority Work Queue
- Execution Engine
- In-memory storage (Hashtable)
- Timestamp Generation
- Local Clock
- MV Record Store
- Write-set Tracker
- Protocol specific components

**Other server processes**
Synch via NTP

**Client Process**
- Non-blocking execution of transactions
  - When a transaction blocks, the thread does not block.
  - The thread “saves the state of the active transaction” and accepts more work from the work queue.
Cloud Hosted Instance

Client Process

Server Process

I/O Threads

Data structures that are specific to each protocol

Lock-table
Scheduler
Sequencer
MV Record Store

Protocol specific components

Waiting Queue

In-memory storage (Hashtable)

Record metadata
Timetable
Write-set Tracker

Timestamp Generation

Local Clock

Sync via NTP

Other server processes
Transaction Protocols

● Concurrency Control
  ○ Two-phase Locking (2PL)
    - NO_WAIT
    - WAIT_DIE
  ○ Timestamp Ordering (TIMESTAMP)
  ○ Multi-version concurrency control (MVCC)
  ○ Optimistic concurrency control (OCC)
  ○ Deterministic (CALVIN)

● Commitment Protocols
  ○ Two-phase Commit (2PC)
Two-phase Locking (2PL)

- **Two phase:**
  - Growing phase: lock acquisition (no lock release)
  - Shrink phase: lock release (no more acquisition)

- **NO_WAIT**
  - Aborts and restarts the transaction if lock is not available
  - No deadlocks (suffers from excessive aborts)

- **WAIT_DIE**
  - Utilizes timestamp
  - Older transactions wait, younger transactions abort
  - Locking in shared mode bypasses lock queue (which contains waiting writers)
Timestamp Ordering (TIMESTAMP)

- Executes transactions based on the assigned timestamp order
- No bypassing of wait queue
- Avoids deadlocks by aborting older transactions when they conflict with transactions holding records exclusively
Multi-version Concurrency Control (MVCC)

- Maintain multiple timestamped copies of each record
- Minimizes conflict between reads and writes
- Limit the number of copies stored
- Abort transactions that try to access records that have been garbage collected
MVCC

Cloud Hosted Instance

Server Process

I/O Threads

Priority Work Queue

Execution Engine

Protocol specific components

I/O Threads

Priority Work Queue

Execution Engine

In-memory storage (Hashtable)

Timestamp Generation

Lock-table

Waiting Queue

Scheduler

Record metadata

Sequencer

Timetable

MV Record Store

Write-set Tracker

Other servers

Processes

Sync via NTP

Local Clock
Optimistic Concurrency Control (OCC)

- Based on MaaT [Mahmoud et. al, MaaT protocol, VLDB’14]
- Strong-coupling with 2PC:
  - CC’s Validation == 2PC’s Prepare phase
- Maintains time ranges for each transaction
- Validation by constraining the time range of the transaction
  - If time range is valid => COMMIT
  - Otherwise => ABORT
Deterministic (CALVIN)

- Discussed in previous class
- Key idea: impose a deterministic order on a batch of transactions
- Avoids 2PC
- Unlike others, requires READ_SET and WRITE_SET of transactions to be known a priori, otherwise needs to be computed before starting the execution of the transaction
- In Deneva, a dedicated thread is used for each of sequencer and scheduler.
CALVIN

Cloud Hosted Instance

I/O Threads -> Priority Work Queue -> Execution Engine

Protocol specific components:
- Lock-table
- Scheduler
- Sequencer
- MV Record Store

Server Process:
- Waiting Queue
- Record metadata
- Timetable
- Write-set Tracker
- Timestamp Generation
- Local Clock

Sync via NTP

Other server processes
Evaluation “Hardware”

- Amazon EC2 instances (m4.2xlarge)

M4 instances are the latest generation of General Purpose Instances. This family provides a balance of compute, memory, and network resources, and it is a good choice for many applications.

Features:
- 2.3 GHz Intel Xeon® E5-2686 v4 (Broadwell) processors or 2.4 GHz Intel Xeon® E5-2676 v3 (Haswell) processors
- EBS-optimized by default at no additional cost
- Support for Enhanced Networking
- Balance of compute, memory, and network resources

<table>
<thead>
<tr>
<th>Model</th>
<th>vCPU</th>
<th>Mem (GiB)</th>
<th>SSD Storage (GB)</th>
<th>Dedicated EBS Bandwidth (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m4.large</td>
<td>2</td>
<td>8</td>
<td>EBS-only</td>
<td>450</td>
</tr>
<tr>
<td>m4.xlarge</td>
<td>4</td>
<td>16</td>
<td>EBS-only</td>
<td>750</td>
</tr>
<tr>
<td>m4.2xlarge</td>
<td>8</td>
<td>32</td>
<td>EBS-only</td>
<td>1,000</td>
</tr>
<tr>
<td>m4.4xlarge</td>
<td>16</td>
<td>64</td>
<td>EBS-only</td>
<td>2,000</td>
</tr>
</tbody>
</table>
Evaluation Methodology

- Table partitions are loaded on each server before each experiment
- Number of open client connections: 10K
- 60 seconds warmup
- 60 seconds measurements
- Throughput measure as the number of successfully completed
- Restart an aborted transaction (due to CC) after a penalization period
Evaluation Workload

- YCSB
- TPC-C: warehouse order processing system
- Product-Part-Supplier
Evaluation Workload

- **YCSB**
  - Single table with 1 primary key and 10 columns of 100B each
    - ~16 million records per partition => 16GB per node
  - Each transaction accesses 10 records with independent read and write operation in random order
  - Zipfian distribution of access with theta in [0,0.9]

- **TPC-C**: warehouse order processing system

- **Product-Part-Supplier**
Evaluation Workload

- YCSB
- **TPC-C: warehouse order processing system**
  - 9 tables partitioned by warehouse_id
  - Item table is read-only and replicated at every server
  - Implemented two transaction of TPCC specs (88% of workload)
    - Payment: 15% chance to access a different partition
    - NewOrder: ~10% are multi-partition transactions

- Product-Part-Supplier
Evaluation Workload

- YCSB
- TPC-C: warehouse order processing system
- **Product-Part-Supplier**
  - 5 tables: 1 for each products, parts and suppliers. 1 table maps products to parts. 1 table maps parts to suppliers
  - Transactions:
    - Order-Product (MPT): reads parts of a product and decrement the stock quantity of the parts
    - LookupProduct (MPT): (read-only) retrieve parts and their stock quantities
    - UpdateProductPart (SPT): updates product-to-parts mapping
Scheduling is the bottleneck in CALVIN. Fully parallelized operation because they are independent operations. But it should degrade under high contention few data items are accessed which are serialized unless replication is used.

Figure 2: Contention – The measured throughput of the protocols on 16 servers when varying the skew factor in the YCSB workload.
**Figure 2: Contention** – The measured throughput of the protocols on 16 servers when varying the skew factor in the YCSB workload.
Can this threshold be extended by adding more servers?

**Figure 2: Contention** – The measured throughput of the protocols on 16 servers when varying the skew factor in the YCSB workload.
Figure 2: Contention – The measured throughput of the protocols on 16 servers when varying the skew factor in the YCSB workload.
Figure 2: Contention – The measured throughput of the protocols on 16 servers when varying the skew factor in the YCSB workload.
Figure 3: Update Rate – The measured throughput of the protocols on 16 servers when varying the number of update transactions (5 reads / 5 updates) versus read-only transactions (10 reads) in the workload mixture for YCSB with medium contention ($\theta=0.6$).
- Overhead of remote request.
- Overhead 2PC and impact of locking during 2PC

Number of operations per transaction is increased from 10 to 16.

Figure 4: Multi-Partition Transactions – Throughput with a varying number of partitions accessed by each YCSB transaction.

Legend:
- CALVIN
- MVCC
- NO_WAIT
- OCC
- TIMESTAMP
- WAIT_DIE
**Figure 7: 99\%ile Latency** — Latency from a transaction’s first start to its final commit for varying cluster size.
Figure 7: 99%ile Latency – Latency from a transaction’s first start to its final commit for varying cluster size.
Scalability (no contention)

(a) Read-Only (No Contention)
Scalability (medium contention)

(b) Read-Write (Medium Contention)
Scalability (high contention)
Scalability (Breakdown)

- **USEFUL WORK**: All time that the workers spend doing computation on behalf of read or update operations.
- **TXN MANAGER**: The time spent updating transaction metadata and cleaning up committed transactions.
- **CC MANAGER**: The time spent acquiring locks or validating as part of the protocol. For CALVIN, this includes time spent by the sequencer and scheduler to compute execution orders.
- **2PC**: The overhead from two-phase commit.
- **ABORT**: The time spent cleaning up aborted transactions.
- **IDLE**: The time worker threads spend waiting for work.
Scalability (Breakdown - no contention)

MaaT merges 2PC prepare and OCC’s validation

System is not saturated??

(a) Read-Only (No Contention)
Scalability (Breakdown - medium contention)

(b) Read-Write (Medium Contention)
Scalability (Breakdown - high contention)

(c) Read-Write (High Contention)
Latency breakdown

**Figure 8: Latency Breakdown** – Average latency of a transaction’s final execution before commit.
Figure 9: **Network Speed** – The sustained throughput measured for the concurrency protocols for YCSB with artificial network delays.
Table 2: Multi-Region Cluster – Throughput of a 2-node cluster with servers in AWS US East and US West regions.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CALVIN</th>
<th>OCC</th>
<th>MVCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>8,412</td>
<td>11,572</td>
<td>5,486</td>
</tr>
<tr>
<td>Algorithm</td>
<td>NO_WAIT</td>
<td>TIMESTAMP</td>
<td>WAIT_DIE</td>
</tr>
<tr>
<td>Throughput</td>
<td>15,921</td>
<td>4,635</td>
<td>4,736</td>
</tr>
</tbody>
</table>
Scalability - TPCC - Payment transaction

(a) Payment Transaction
Scalability - TPCC - NewOrder transaction

(b) NewOrder Transaction
Data dependant aborts

- YCSB operation are independent
- Modified YCSB transaction to have conditional abort based on a value read.
- 36% decrease in performance compared to 2%-10% decrease on other protocols.
  - $\theta=0.6$, 50% updates
- CALVIN performs worse with higher contention (drops 73K to 19K txn/s)
## Results Summary

<table>
<thead>
<tr>
<th>Class</th>
<th>Algorithm</th>
<th>2PC delay</th>
<th>MPT</th>
<th>Low Contention</th>
<th>High Contention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locking</td>
<td>NO_WAIT, WAIT_DIE</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Timestamp</td>
<td>TIMESTAMP, MVCC</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Optimistic</td>
<td>OCC</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>Deterministic</td>
<td>CALVIN</td>
<td>NA</td>
<td>B</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>
Bottlenecks in DDBMS

- According to the paper, it boils down to the following bottlenecks:
  - 2PC delay
    - CALVIN is designed to eliminate that but in case a transaction will need to abort. It needs to pay the cost of broadcasting the abort decision
  - Data access contention
    - Read-only contenttion can be trivially solved by replication
    - Write contention is difficult
Further research and additional potential solutions

- Authors mentions many aspects for future research and solutions:
  - Impact of recovery mechanisms
  - Leverage better network technologies (e.g. RDMA)
  - Automatic repartitioning [Schism, H-Store]
  - Force a data model adaptation on application developers
    - (e.g. entity group- Helland CIDR’07, G-Store)
  - Semantic based concurrency control methods

- Is there a way to generalize CC protocols into a framework that admits different configurations and yield different CC protocols implementation?
  - e.g. Similar to GiST generalizes search tree for indexes, and SP-GiST generalizes space-partitioning trees.

- Contention-aware adaptive concurrency control
  - 2PL or Timestamp under low contention and switch to OCC or CALVIN under high contention

- Evaluating abort rate