Calvin: Fast Distributed Transactions for Partitioned Database Systems

A. Thomson et al., SIGMOD’12

Presenter: Lianghong Xu
High-level Overview

• A transaction processing and replication layer
  – Based on generic, non-transactional data store
  – full ACID for distributed transactions
  – Active, consistent replication
  – Horizontal scalability
The Problem

- Distributed transactions are \textit{expensive}
  - Agreement protocol
    - Multiple roundtrips in 2-phase commit
    - Much longer than transaction logic itself
    - Limits scalability
  - Locking
    - Lock held during the entire transaction
      - Including network latency
    - Possible deadlock
Consistent replication

• Many systems allow inconsistent replication
  – Replicas can diverge, but are eventually consistent
  – Dynamo, SimpleDB, Cassandra...

• Consistent replication: emerging trend
  – Instant failover
  – Increased latency, especially for geo-replication

Cost is only in latency, not throughput or contention
Goals of Calvin

• Eliminate 2-phase commit (scalability)
• Reduce lock duration (throughput)
• Provide consistent replication (consistency)

Approach:
• Decoupling “transaction logic” from “heavy-lifting tasks”
  • replication, locking, disk access…

• Deterministic concurrency control
Non-deterministic Database Systems

• Aborts on non-deterministic events
  – E.g., node failure

• Serial ordering cannot be pre-determined given certain transaction inputs
  – Determined in the runtime
  – Ordering can diverge for different executions
  – Example: 2-phase locking
Serial Ordering in 2-phase Locking

![Diagram showing the number of locks over time with a note about ordering of locked points determining the serialization order.]

[Slide from Yoongu Kim]
Deterministic database systems

• Given transactional inputs, serial ordering is *pre-determined*.

• Benefits
  – No agreement protocol
    • No need to check node failures
    • Recovery from other replicas
    • *No aborts* due to non-deterministic events
  – Consistent replication made easier
    • Only need to replicate *transactional inputs*

• Disadvantage
  – Reduced concurrency (potentially)
Architecture

Client

Requests

Replica A

Sequencer

Scheduler

Storage

Replica B

Sequencer

Scheduler

Storage

Replicated requests
Architecture

Client

Requests

Replicated requests

Replica A
- Sequencer
- Scheduler
- Storage

Replica B
- Sequencer
- Scheduler
- Storage
Architecture

Client

Replica A
- Sequencer
- Scheduler
- Storage

Replica B
- Sequencer
- Scheduler
- Storage

Requests

Replicated requests

Partial request batch

Local access

Remote access
**Architecture**

- **Client**
  - Requests
  - Replicated requests

- **Sequencer**
- **Scheduler**
- **Storage**

**Consistent replication**
- Partial request batch

**Deterministic locking**
- Remote access
- Local access

**“CRUD” storage interface**

- Replica A
- Replica B
Sequencer and replication

• Only transactional inputs need to be replicated
  – No need to worry about serial ordering (determinism)

• Asynchronous replication
  – Master replica handles all requests
  – Propagate to slave replicas afterwards
  – Low latency, but complex failure recovery

• Synchronous replication
  – Based on Paxos (ZooKeeper)
  – Larger latency
  – Throughput not affected
    • Contention footprint remains the same
Scheduler and deterministic locking

- Logical view of records
- Global transaction order for its own share
- Only responsible for locking locally stored data
- **Single-threaded** locking manager
  - Resemble 2-phase locking
  - **Enforce transaction ordering from sequencers**
    - For conflicted updates
  - No deadlock
Deterministic Locking

Pre-determined serial ordering

Memory

Disk

T1
Read A
Read B
Write A

T2
Read B
Write B

T3
Read C

A
B
C
Deterministic Locking

Pre-determined serial ordering

T1: Lw(A)

T1: Lr(B)
  T2: Lw(B)

T3: Lr(C)

Memory

Disk

A

B

C

T1:
  Read A
  Write A

T2:
  Read B
  Write B

T3:
  Read C
Deterministic Locking

Pre-determined serial ordering

T1: Lw(A)
T2: Lw(B)
T3: Lr(C)

T1 must precede T2

T1: Lr(B)
T2: Lw(B)

T1: Read A
Write A
Read B
Write B
Read C
Worker execution

Pre-determined serial ordering

T1
- Read A
- Write A

T2
- Read B
- Write B

T3
- Read C

Memory

Disk

A

B

C
Worker execution

Pre-determined serial ordering

T1
- Read A
- Read B
- Write A

T2
- Read B
- Write B

T3
- Read C

Memory

Disk

Active participant

Passive participant

Passive participant
Worker execution

Pre-determined serial ordering

Memory

Disk

Active participant

Passive participant

Passive participant

A

B

C

Read A
Read B
Write A

Read B
Write B

Read C

T1

T2

T3
Worker execution

Pre-determined serial ordering

Memory

Disk

Active participant

Passive participant

Passive participant
Worker execution

Pre-determined serial ordering

Active participant

Passive participant

Passive participant
Worker execution

Pre-determined serial ordering

T1
- Read A
- Read B
- Write A

T2
- Read B
- Write B

T3
- Read C

Active participant
Problem of deterministic locking

- **T2 cannot proceed** because T1 is block waiting on disk access to A
- Even if T1 does NOT acquire any lock for B yet
- T3 can still proceed

Deterministic locking

- T1
  - Read A
  - Read B
  - Write A
- T2
  - Read B
  - Write B
- T3
  - Read C

Need to fetch from disk
Problem of deterministic locking

- **T2 cannot proceed** because T1 is block waiting on disk access to A
- Even if T1 does NOT acquire any lock for B yet
- T3 can still proceed
Calvin’s Solution

• **Delay** forwarding transaction requests
  – Prefetech data from disk
  – Hope: most (99%) transactions execute without need to access disk

• Problem
  – How long to delay?
    • Need to estimate disk and network latency
  – Tracking **all** in-memory records
    • Need a scalable approach
  – Future work
Checkpointing

• Fault-tolerance made easier in Calvin
  – Instant failover
  – Only log transactional inputs, no REDO log

• Checkpointing for *fast recovery*
  – Failure recovery from a recent state
  – Rather than from scratch
Checkpointing (cont.)

- **Naïve synchronous checkpointing**
  - Freeze one replica only for checkpointing
  - Difficult to bring the frozen replica up-to-date

- **Asynchronous variation of Zigzag**
  - Zigzag
    - 2 copies per replica, 2 times memory footprint
    - 1 copy is up-to-date, the other is the latest checkpoint
    - Need to stop completely to ensure checkpoint consistency
  - Calvin’s optimizations
    - Pre-specified virtual consistency point, no need to stop the system
    - Copy-on-write only during checkpointing, reduce memory footprint

- **Asynchronous snapshot mode**
  - Storage layer needs to support multiversioning
Scalability

- Calvin scales (near-)linearly
- Throughput comparable to world-record
  - With much cheaper hardware

Figure 4: Total and per-node TPC-C (100% New Order) throughput, varying deployment size.
Scalability (cont.)

• Scales linearly for low contention workloads
• Scales sub-linearly when contention is high
  • Stragglers (slow machines, execution process skew)
  • Exacerbated with higher contention
Scalability under high contention

Calvin scales better than 2PC in face of high contention