Extracting More Concurrency from Distributed Transactions

Authored by Shuai Mu, Yang Chui, Yeng Zhang, Wyatt Lloyd and Jinyang Li

Presented by Darshan Valia
Introduction

- Everyone wants their system to scale while supporting transactions

Transactions require strict serializability
  - Guaranteed by concurrency control

- What if there were no concurrency control in a system, like say shopping at Amazon?
  - Amazon might charge you twice
  - Amazon might deliver the same item twice for the price of one

- Popular protocols providing concurrency control:
  - Two Phase Locking (2PL)
  - Optimistic Concurrency Control (OCC)
Use Case

- Combo offer for “Imitation Game” and “Theory of Everything”
- Stock for Imitation Game in Shard 1, Stock for Theory of Everything in Shard 2
- Two users buying both at same time

```
IG = 5 left
TOE = 3 left
```
Two Phase Locking

SHARD 1
- T1 Locks IG
- T2 wants IG, blocked
- T1 decrements IG stock
- T1 Commits, unlocks IG
- T2 locks IG
- T2 decrements IG stock
- T2 commits, unlocks IG

SHARD 2
- T1 Locks TOE
- T2 wants TOE, blocked
- T1 decrements TOE stock
- T1 Commits, unlocks TOE
- T2 locks TOE
- T2 decrements TOE stock
- T2 commits, unlocks TOE
Optimistic Concurrency Control

SHARD 1

T1 arrives, records timestamp
T2 arrives, records timestamp
T1 modifies value
T2 modifies value
T1 validates, inconsistency found, ABORT
T2 validates, inconsistency found, ABORT

SHARD 2

T1 arrives, records timestamp
T2 arrives, records timestamp
T1 modifies value
T2 modifies value
T1 validates, inconsistency found, ABORT
T2 validates, inconsistency found, ABORT
Introducing ROCOCO

- ROCOCO - Reordering Conflicts for Concurrency

- Aims to extract more concurrency during contention
  - Without aborting (unlike OCC)
  - Without blocking (unlike 2PC)

- Basic Idea:
  - Break transactions into atomic pieces
  - Identify dependencies of various transaction pieces across different servers
  - Reorder the pieces deterministically and then execute
Introduction to ROCOCO
Start Phase

T1
- TOE
- IG
- T1
- Dep

T2
- TOE
- IG
- T1
- Dep

TOE

IG
Start Phase
Start Phase
Start phase
Start phase
Commit phase
Commit phase

T1
- TOE
- IG

T2
- TOE
- IG

T1
- TOE
- T1
- T2

T2
- T2
- T1
- IG

Dep
- T2
- T1
Commit phase
Commit phase

Reorder deterministically
Commit phase

Reordering done
COMMIT!
Introduction to ROCOCO

- Some transactions cannot be reordered
- What if the output of one piece acts as an input to another piece?
- These pieces need to be executed immediately!
- We need to determine which pieces are immediate and which can be deferred
- This is done by a component called the “Offline Checker”
Unreorderable transactions
Offline Checker : S/C Cycles

```
T1
+----------------+----------------+
| Item_table     | Sibling(S)-edge|
|                |                |
+----------------+----------------+
|                | Conflict(C)-edge|
+----------------+----------------+
T2
+----------------+----------------+
| Item_table     |                |
|                |                |
+----------------+----------------+
```

```
T1
+----------------+----------------+
| Item_table     | Sibling(S)-edge|
|                |                |
+----------------+----------------+
|                | Conflict(C)-edge|
+----------------+----------------+
T2
+----------------+----------------+
| Item_table     |                |
|                |                |
+----------------+----------------+
```
Offline Checker: Immediate/Deferrable pieces
Typical ROCOCO workflow

1. The output of \( p_1 \) contains the input of \( p_3 \)
2. Receive replies to start requests of all pieces
3. The servers may exchange dependencies to reach a deterministic serializable order
4. All pieces have finished executing and all outputs are ready
Typical ROCOCO workflow

1. The output of \( p_1 \) contains the input of \( p_3 \).
2. Receive replies to start requests of all pieces.
3. The servers may exchange dependencies to reach a deterministic serializable order.
4. All pieces have finished executing and all outputs are ready.
Typical ROCOCO workflow

1. The output of $p_1$ contains the input of $p_3$.
2. Receive replies to start requests of all pieces.
3. The servers may exchange dependencies to reach a deterministic serializable order.
4. All pieces have finished executing and all outputs are ready.
Protocol: Start phase

- Coordinator sends requests for pieces to appropriate servers
- If piece is immediate, server executes piece and returns output; else buffers for later execution
- Server creates and maintains dependency graph:
  - Vertices: transactions and their status (started, committing or decided)
  - Edges: Conflicting pieces between two transactions. Labelled by {immediate, deferrable} depending on type of piece
- Server returns updated dependency graph and immediate pieces’ execution outputs
Protocol: Commit Phase

- Begins after coordinator sends commit requests containing aggregated dependency graph of all servers

- Updates status of transaction in graph to “committing” if status is “started”. Aggregates coordinators dependency graph to its own

- Waits for all ancestors of transaction in graph to become committed

- Calculates SCC of transaction, sets all transactions within SCC to “decided” state

- Waits for all ancestors of SCC to be decided

- Server sorts transactions in SCC according to the “I”-edges, executes them in the order given by the sort

- Returns results to coordinator
Optimizations and Fault Tolerance

Optimizations
- Track only one-hop dependencies instead of entire-graph dependencies
  - One technique is to only add the most recent conflicts for each piece to server’s dependency graph instead of all previous ones
  - In start phase, instead of entire dependency graph, server provides only subgraph of transaction’s ancestors which are not yet “decided”

Fault tolerance
- Transaction logs persisted to disk; replicated using paxos-like systems
- Coordinator logs every transaction request
- Server logs every start request
Evaluation : Setup and Workload

- Kodiak testbed; each machine having 1-core 2.6Ghz AMD Opteron 252 CPU, 8GB RAM, Gigabit Ethernet
- Each client running 1-30 single-threaded client processes, each server machine running one single-thread server process
- Logging turned off
- Partition strategy: Partition by warehouse, which in turn is partitioned by districts
- Ratio of customer, district and warehouse = 3M:1K:1
Evaluation: Throughput

![Throughput Graph](image)
Evaluation: Commit Rates
Evaluation : Latency

![Graph showing latency across different concurrency levels for OCC, 2PL, and Rococo.]
Evaluation: Scale

(a) 10 clients per server.
(b) 20 clients per server.
(c) 40 clients per server.
Related Work

- 2PL Forms and variations: Gamma, Bubba, R*, Spanner (replicated commit)
- OCC forms and variations: H-store, VoltDB, MDCC, Percolator, Adya
- Concurrency control with limited transactions: Megastore (serializable transactions only within a data partition), Granola, Calvin and Sinfonia (concurrency protocols for known read-write keys)
- Dependency and interference: Paxos variants, COPS/Eiger (tracks dependencies within operations), Warp
- Transaction Decomposition and Offline checking: Transaction Chopping theory by Shasha et al (utilized by ROCOCO offline checker), Lynx
- Geodistributed systems with weaker semantics: Dynamo, Cassandra, Walter, Gemini
Comments, Criticism and Questions

- No allowance for user-initiated aborts
- Any difference in performance for read-only and read-write transactions? Evaluations are combined for both types
- Breaking transactions to pieces: is this trivial for all OLTP systems?