High-Performance Transactions

6.5830/6.5831
Lecture 18
Sam Madden

Based on slides from Tianyu Li
Recap – Transaction Model So Far

Single-node

- Disk-based
- 2PL
- Write-ahead Logging + Checkpoints
Recap - Transactions

Multi-node

- 2PC for multi-node transactions
- Shared-nothing architecture. Use replication for high-availability.
Critique

- “Classical” DBMSes matured in the 80s and 90s
- Hardware & workloads were very different back then
- Why are we still using the same model for processing transactions?
**Times Are Different**

<table>
<thead>
<tr>
<th>1980s</th>
<th>Now</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow Networks (&lt; 10 Mb/sec)</td>
<td>40+ Gb/sec</td>
</tr>
<tr>
<td>Small number of on-prem machines</td>
<td>Global-scale, cloud</td>
</tr>
<tr>
<td>Single or few-core</td>
<td>100+cores</td>
</tr>
<tr>
<td>Few MB of memory</td>
<td>100+GB RAM / machine</td>
</tr>
</tbody>
</table>

Is ARIES still the right way to go?
Classic Design Has High Overhead

- Running old code on new hardware != speed-up
- New performance bottlenecks
- New architecture required to make use of faster hardware

Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. SIGMOD 2008
Today -- High Performance Transactions

- Looking Back

- Multi-node
  - Bottleneck: 2-Phase Commit
  - Single-Site Execution
  - Deterministic Transactions

- Cloud Transactions
Recap: Scaling a Database

- More shards/partitions --- more parallelism and throughput
- More replicas --- higher availability

* Replicas usually also serve read requests
Recap: Scaling a Database

Unless we are careful, replication hurts write performance, but increases availability.
Recap: 2-Phase Commit

1. Log start of transaction
2. Execute transaction on worker nodes
3. PREPARE each worker
4. Log transaction commit if all OK
5. Commit each worker
6. Log Done
Critique: 2-Phase Commit

- 2 network round trips + synchronous logging
  - Worse still — likely need to hold locks throughout process

- 2PC blocks when coordinator fails

- 2PC sacrifices performance for strong guarantees
Example: Google Spanner

- A rare example of geo-distributed strongly consistent transactional system
  - You get the same guarantee as single-node

- Optimized for read-only transactions with TrueTime

- Optimized 2PC (on Paxos)
Problem

2PC Scalability

<table>
<thead>
<tr>
<th>participants</th>
<th>latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
</tr>
<tr>
<td>1</td>
<td>17.0 ± 1.4</td>
</tr>
<tr>
<td>2</td>
<td>24.5 ± 2.5</td>
</tr>
<tr>
<td>5</td>
<td>31.5 ± 6.2</td>
</tr>
<tr>
<td>10</td>
<td>30.0 ± 3.7</td>
</tr>
<tr>
<td>25</td>
<td>35.5 ± 5.6</td>
</tr>
<tr>
<td>50</td>
<td>42.7 ± 4.1</td>
</tr>
<tr>
<td>100</td>
<td>71.4 ± 7.6</td>
</tr>
<tr>
<td>200</td>
<td>150.5 ± 11.0</td>
</tr>
</tbody>
</table>

2PC end-to-end Latency

<table>
<thead>
<tr>
<th>operation</th>
<th>latency (ms)</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>all reads</td>
<td>8.7 ± 376.4</td>
<td>21.5B</td>
</tr>
<tr>
<td>single-site commit</td>
<td>72.3 ± 112.8</td>
<td>31.2M</td>
</tr>
<tr>
<td>multi-site commit</td>
<td>103.0 ± 52.2</td>
<td>32.1M</td>
</tr>
</tbody>
</table>

- 2PC is very expensive
Question: Can we do better?
Aside: Why is this difficult?

Well-known theoretical limitations

- In short, you CANNOT have a “fast and reliable” distributed ACID system.
  - Two Generals Problem [Gray ’78]
  - CAP Theorem [Brewer ’00, Gilbert ’02]
  - Coordination Avoidance in Database Systems [Bailis ’15]

- We covered this last lecture
  - Many use cases regress to using “NoSQL” systems with more scalability but less guarantees
Why bother with distributed transactions then?

- Really powerful abstraction
- Extremely useful
- Impossibilities are mathematical. We are here to build systems*.

* Often called “NewSQL” systems
Attempt 1: H-Store

- Large collaborative project @ MIT (among other places).
- Distributed & main-memory
- Commercialized as VoltDB
How to make transaction processing databases 10x faster?

- Eliminate
  - Disk I/O
  - Locking
  - Concurrency Control
  - Disk based recovery
- Sounds nuts, but go on to show how we:
  - Do this while preserving transactional guarantees
  - Get massive scalability, even on multicores
No Disk

- Horizontally partition into RAM-sized chunks
  - Most OLTP workloads partition nearly perfectly
    - E.g., in Amazon, almost all transactions begin with a customer
    - Also true of TPC-C, Ebay, travel sites, banking, etc.
  - Most OLTP databases easily fit into the aggregate RAM of a cluster

- Replicate for durability
  - If one site crashes, another has data

Image courtesy of Prof. Andy Pavlo
No Concurrency Control

- **Single-threaded execution**
  - Only execute one transaction at a time
  - All transactions “one shot” stored procedures
    - No user stalls / “think time”
  - Concurrent transactions needed to mask I/O latency
    - Unneeded if every transaction takes 100 us and there are no disk, network, or user stalls
- **Fall back on 2PC for multi-site transactions**

Example stored procedure:

```sql
Debit(A, B, amt):
  UPDATE accts SET bal = bal - amt
  WHERE acct_no = $A$
  UPDATE accts SET bal = bal + amt
  WHERE acct_no = $B$
```

*Remember, database is in memory and most transactions can be answered at a single partition!*
No Disk-based Logging

- Recover from replicas
  - By copying state on crash
  - Possible to asynchronously checkpoint to disk

- May need in-memory logs for transaction undo
Is this reasonable?

Specialized for OLTP (Online Transaction Processing) workload

- Transactions access a few records
- Transaction templates are known beforehand
- Working set fits in memory
- Data is (mostly) partitioned
Example: TPC-C

- Standardized benchmark used by everyone
- Models a warehouse order processing system
- Several types of transaction issued at random
- E.g. NewOrder Transaction:
  - Check item stock level
  - Create a new order
  - Update item stock level

Small set of pre-declared transactions
H-Store: Performance

- Vanilla H-Store ran 70K TPC-C txns
- At the time:
  - MySQL ~1K on similar hardware

- At the time, TPC-C record was about 133 K txns/s on a 128 core server.
  - H-Store achieved half of that on low-end desktops.
H-Store: Partitioning

- H-Store performance hinges on percentage of one-site txns
- Huge win if we can maximize one-site probability
- Intelligent partitioning required
H-Store: Speculative Execution

- Recall: H-Store single-threaded
- Problem: 2PC takes > 10 ms to complete
  - If lots of 2PC, performance suffers
- In vanilla H-Store partition simply waits out the 10ms instead of doing work
H-Store: Speculative Execution

- Observation: Most transactions succeed
- Idea: Assume transaction succeeds. Forge ahead but don’t release speculative results.
- Problem: introduces concurrency, but must not add overhead

Evan P.C. Jones, Daniel J Abadi, Samuel Madden. Low Overhead Concurrency Control for Partitioned Main Memory Databases. SIGMOD 2010
H-Store: Speculative Execution

- Idea: Speculate when waiting for 2PC outcome (e.g., after the transaction has completed)
  - No locks required
  - Local transactions execute assuming 2PC will complete
  - Results held back until 2PC finishes
    - Record undo information in-memory
    - If 2PC fails, all speculated transactions fail
  - Paper explores several other models
H-Store: Speculative Execution

- Synthetic benchmark --- single operation transactions
- Baseline no conflict
Attempt 2: Calvin / Aria

- Why is H-Store faster without concurrency?

- No non-determinism from threading
  - Limits cross thread/node coordination need
  - Coordination often a bottleneck

- Can the same idea be applied to truly distributed transactions?
Key Idea: Calvin

- Have a global *deterministic* ordering of transaction execution.

- Take the input and execute anywhere. Get the same result.

Deterministic Transactions

External Sequencer

Sequencer only issues transactions that don’t conflict
Deterministic Transactions

- Observe: this is not so different from 2PL, where execution is equivalent to a serial schedule

- However: Calvin fixes the schedule before execution, so no locking required
  - Assuming we only issue concurrent transactions that don’t conflict

- Therefore: coordination also largely done before execution

- Avoids 2PC because no deadlocks; if a node fails it can simply re-run transactions in pre-determined order
Practical Considerations

- Sequencer needs to know which items a transaction will access
  - Hard! What about
    
    ```sql
    UPDATE sal = sal * 1.05 WHERE sal < 50k
    ```
  - Locks that are needed are data dependent

- Sequencer is a bottleneck of the system and single-point of failure
- We still want concurrency for performance on a single node
- Need to be recoverable / durable
Practical Considerations

- Sequencer is a bottleneck of the system and single-point of failure
- We still want concurrency for performance on a single node
- Need to be recoverable / durable
Sequencer: Initial Attempt

[Diagram showing clients connected to a special node with T1, T2, T3, and attached storage.]
Sequencer: Initial Attempt

- Special node failure difficult to handle
- Txn throughput bottlenecked by special node throughput
Distributed Sequencer

- Don’t synchronize for every request
- Each sequencer collects a batch of requests
- Periodically replicate / persist and exchange batches
Practical Considerations

- Sequencer is a bottleneck of the system and single-point of failure

- We still want concurrency for performance on a single node

- Need to be recoverable / durable
Scheduler: Deterministic Concurrency Control

Consider Schedule:

- No actual conflict
- No reason to execute in-order
- Challenge: concurrent execution that preserves deterministic schedule
Scheduler: Deterministic Concurrency Control

- Need to allow for concurrent execution

- However, concurrent execution has to follow predetermined schedule
Scheduler: Deterministic Concurrency Control

- Similar to 2 PL
- Allow arbitrary concurrent execution permitted by lock manager
- However, control how locks are granted
Scheduler: Deterministic Concurrency Control

- Don’t request locks, grant locks.
- Dedicated lock thread assigns locks strictly in predetermined order.
- Transaction executes when all locks are granted.
- **Assumption**: read/write set known / can be determined before execution.
Practical Considerations

- Sequencer is a bottleneck of the system and single-point of failure
- We still want concurrency for performance on a single node
- Need to be recoverable / durable
Logging and Checkpoints

- Transactions still need to be durable; since we don’t want to FORCE after every command, need to have a way to redo work

- Because deterministic:
  - Can just log commands and the order they execute in
  - No undo logging required
    - No deadlocks / node-generated aborts

- Checkpointing needed
  - Otherwise, on failure, have to replay from the beginning of time
  - Need a way to take transaction-consistent snapshots
Calvin: Results

- TPC-C (100% New Order)
Calvin: Results

- Synthetic Microbenchmark
Calvin: Criticism

- Transaction read/write sets must be known beforehand
- Not always practical
Aria: Practical Deterministic OLTP

- Relaxes the requirement to know R/W sets beforehand
- Speculatively execute first, repair later
- Details omitted
Takeaways

- Determinism can be a good thing
- Distributed coordination off the critical path = win
What have we achieved?

- A class of new transactional systems (aka. NewSQL) that retains the strong guarantees of traditional relational DBMS, while being much more scalable and performant like NoSQL systems
  - These systems are largely main-memory systems
  - These system optimize around partitioning and sharding for performance
  - These system feature new, interesting concurrency control / distributed commit schemes

  - Txn throughput went from a couple of thousands to millions per second
We Are Boring

Sam Madden
madden@csail.mit.edu

AI is enjoying a renaissance, with popular press and major corporations investing in a variety of smart, AI-based applications, from self-driving cars to household robots to household gadgets that learn our behaviors and preferences. Despite all of these applications revolving around data, the database community appears to cede these domains to our AI colleagues. This is absurdly shortsighted, and foolish. After inventing the world-wide web, and (nearly) big data, we risk being an also-ran in computer science in the coming decade. These smart systems will work, and play, and the database community ought to be thinking about what we can add value to these applications.

What Are We Doing With Our Lives?
Nobody Cares About Our Concurrency Control Research

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT
Most of the academic papers on concurrency control published in the last five years have assumed the following two design decisions:
(1) applications execute transactions with serializable isolation and
(2) applications execute most (if not all) of their transactions using
stored procedures. I know this because I am guilty of writing these
papers too. But results from a recent survey of database administra-
tors indicates that these assumptions are not realistic. This survey
includes both legacy deployments where the cost of changing the
application to use either serializable isolation or stored procedures

1. ACKNOWLEDGEMENTS
This work was supported (in part) by the Intel Science and Tech-
nology Center for Big Data and the U.S. National Science Founda-
tion (CCF-1438955).

2. BIOGRAPHIES
Andrew Pavlo is an Assistant Professor of Database in the
Computer Science Department at Carnegie Mellon University. At
CMU, he is a member of the Database Group and the Parallel Data
Criticism

- Do we really need many more transactions per second?
  - In most enterprises, general purpose OLTP (e.g., Postgres / MySQL) are fine
  - Some extremes: 750 M req/s on China’s 11/11 Single’s Day
    - Most of this workload embarrassingly parallel

- Are these new algorithms practical?
  - Assumptions, e.g., all data in RAM, replicas for recoverability, write sets known requires specialized use cases and assumptions
  - Often easier to just use a general-purpose system
Transactions in the Cloud

- Several differences
  - Failures common
    - Must replicate across availability zones and even data centers
  - Highly-available shared object storage (e.g., S3) exists
  - Desire for “pay as you go” scaling

- A number of new “cloud-native” database systems have emerged
  - E.g., AWS Aurora, Snowflake, SingleStore, FoundationDB, etc.
  - Build on top of existing cloud storage services in “shared-disk” fashion
  - Most separate compute and storage for flexibility
Cloud-native OLTP

- Key Idea: Storage & Compute Separation
  - Use cloud object storage (e.g., S3) for persistent storage layer
  - Attach ephemeral machines to storage when needed
  - Allows for separate scaling of resources

- Key Challenge: Performance
  - Object storage is often slow & over the network (upwards of 10ms instead of hundreds of microseconds of fast SSDs, and often rate-limited to tens of MBs per second)
Example: Amazon Aurora

- Idea: take existing DBMS (e.g., PostgreSQL), and replace the storage layer
- Optimized storage layer to reduce commit latency and materialize pages in S3 using logging
- Data distributed across multiple storage nodes for read performance and high availability
- Avoids use of 2PC by using quorum writes
Aurora Execution

On write, storage node:

1. receives redo records,
2. appends them to an update queue, acks

In background, the storage node

3. sorts and groups records,
4. gossips with peers to fill in missing records,
5. coalesces them into data blocks,
6. backs them up to S3,
7. garbage collects backed-up data
8. periodically verifies checksums continue to match the data on disk.
Storage

- Every write is structured as a REDO log, with a unique LSN
  - Storage nodes flush blocks to S3 asynchronously
- Data is partitioned into segments, which are replicated
- Different segments may be on different replica sets
- Each segment has a separate log
Log Processing

• Every write (log record) has an LSN, generated by primary
• Storage nodes process log writes in order
  ○ Stall if missing a block
• If a storage node is missing some log, it gossips with other nodes to fill in holes

![Diagram of log processing in Azure](image-url)
Quorum Writes

- Rather than writing all replicas, primary writes to a quorum
- Allows survival of failure of one or more replicas
- Typically, Aurora uses $N=6$, $W=4$, meaning it can tolerate the failure of 2 replicas.
  - Replicas are spread across 3 availability zones
  - Tolerating 2 failures allows one AZ to be down and one other failure

Figure 1: Why are 6 copies necessary?
Reads

- Aurora does not need to do quorum reads, because of the use of a primary
  - Either data is in cache
  - Or it knows which replicas have the most current version of each block
    - Since it coordinates all of the writes

https://dl.acm.org/doi/10.1145/3183713.3196937
Commit

• Transactions may write data stored in different segments
• Since segments are on different replica sets, seems to require 2PC
• However:
  o Primary does concurrency control, properly sequences writes
    ■ No deadlocks will be generated on storage nodes
  o Quorum writes ensure that replica sets will not fail
    ■ As long as a quorum of nodes stays up
→ 2PC isn’t required
  o Commit point is when primary’s log has been replicated
    ■ Log is stored on one replica set
• Recovery of primary is somewhat complicated; see paper

https://dl.acm.org/doi/10.1145/3183713.3196937
Performance

- Despite 4x+ write amplification, performance is good because:
  - Writes append to REDO log; no synchronous block writes
  - Data is spread across many storage nodes, allowing for high concurrency
  - No 2PC required for commit
  - No read amplification

Aurora vs. MySQL on EBS (cloud storage)

<table>
<thead>
<tr>
<th>Connections/Size/Warehouses</th>
<th>Amazon Aurora</th>
<th>MySQL 5.6</th>
<th>MySQL 5.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>500/10GB/100</td>
<td>73,955</td>
<td>6,093</td>
<td>25,289</td>
</tr>
<tr>
<td>5000/10GB/100</td>
<td>42,181</td>
<td>1,671</td>
<td>2,592</td>
</tr>
<tr>
<td>500/100GB/1000</td>
<td>70,663</td>
<td>3,231</td>
<td>11,868</td>
</tr>
<tr>
<td>5000/100GB/1000</td>
<td>30,221</td>
<td>5,575</td>
<td>13,005</td>
</tr>
</tbody>
</table>

Figure 8: Web application response time

Aurora is higher throughput and lower latency, because of use of log shipping and scalable backend
Takeaways

● Transactions have come a long way since the classical 2PL + ARIES + 2PC

● A host of new systems leveraging workload specialization and other clever insights to boost transactional performance by many orders of magnitude
  ○ Whether all of this speed-up is needed is debatable
  ○ Regardless, many of the innovations run in production today

● Transaction research is alive and well in new settings such as the autoscaling cloud