High-Performance Transactions

6.5830/6.5831 Lecture 19
Tianyu Li
Recap - Transactions

Single-node

- Disk-based

- 2PL

- Write-ahead Logging + Fuzzy Checkpoints
Recap - Transactions

Multi-node

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

- “Classical” DBMS matured in the 80s and 90s
- Hardware & workload was much different back then
- DBMS is about dealing with limitation of hardware
- Do the original assumptions still hold?
Critique

Back then: Network is slow

   Now: Fiber optics can easily hit 100s of Gbps

Back then: The database machine exists in a basement

   Now: The public cloud, global scale
Critique

Back then: Single (or a couple of) processor(s)

Now: AWS offers 96 core machines

Back then: RAM is limited

Now: Said 96 core machines has 384 GiB (!) of RAM
What happens now?

Can’t we just take the DBMS, run it on faster hardware, and get better performance?
What happens now?

- 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
  ○ Epoch-based Coordination

● Looking Ahead
Recap: Scaling a Database

More shard/partitions --- more parallelism and throughput

More replicas --- higher availability

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

Primary-Backup Replication

2-Phase Commit

A

A'

A''

B

C
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 in coordinator fails, until the coordinator can be replaced or recovered

- 2PC basically 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)

Corbett et al. Spanner: Google’s Globally-Distributed Database. OSDI 2012
### Problem

- Read-only transactions scale and perform well
- Read-write transactions not so much

<table>
<thead>
<tr>
<th>replicas</th>
<th>latency (ms)</th>
<th>throughput (Kops/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>write</td>
<td>read-only transaction</td>
</tr>
<tr>
<td>1D</td>
<td>9.4±.6</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>14.4±1.0</td>
<td>1.4±.1</td>
</tr>
<tr>
<td>3</td>
<td>13.9±.6</td>
<td>1.3±.1</td>
</tr>
<tr>
<td>5</td>
<td>14.4±.4</td>
<td>1.4±.05</td>
</tr>
</tbody>
</table>
Problem

2PC Scalability

<table>
<thead>
<tr>
<th>participants</th>
<th>latency (ms)</th>
<th>99th percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>17.0 ± 1.4</td>
<td>75.0 ± 34.9</td>
</tr>
<tr>
<td>2</td>
<td>24.5 ± 2.5</td>
<td>87.6 ± 35.9</td>
</tr>
<tr>
<td>5</td>
<td>31.5 ± 6.2</td>
<td>104.5 ± 52.2</td>
</tr>
<tr>
<td>10</td>
<td>30.0 ± 3.7</td>
<td>95.6 ± 25.4</td>
</tr>
<tr>
<td>25</td>
<td>35.5 ± 5.6</td>
<td>100.4 ± 42.7</td>
</tr>
<tr>
<td>50</td>
<td>42.7 ± 4.1</td>
<td>93.7 ± 22.9</td>
</tr>
<tr>
<td>100</td>
<td>71.4 ± 7.6</td>
<td>131.2 ± 17.6</td>
</tr>
<tr>
<td>200</td>
<td>150.5 ± 11.0</td>
<td>320.3 ± 35.1</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></td>
<td>mean</td>
<td>std dev</td>
</tr>
<tr>
<td>all reads</td>
<td>8.7</td>
<td>376.4</td>
</tr>
<tr>
<td>single-site commit</td>
<td>72.3</td>
<td>112.8</td>
</tr>
<tr>
<td>multi-site commit</td>
<td>103.0</td>
<td>52.2</td>
</tr>
</tbody>
</table>

- 2PC is simply too 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
Key Idea

Recall:

- **Disk-based system overhead**
- **Multi-thread concurrency overhead**
Key Idea

- H-Store is a main-memory system
- H-Store partitions data, and executes single-threaded within each partition (site)

M. Stonebraker et. al. 
Partition vs. Threads

- **Threads**
  - Synchronization Overhead

- **Partitions**
  - No Concurrency Control Required

Slide courtesy of Prof. Andy Pavlo
Is this reasonable?

Specialized for OLTP (Online Transaction Processing) workload

- Transactions finish quickly
- Transactions are almost always point queries
- Transactions are known beforehand
- Working set fits in memory
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

* Technically, TPC-C is strictly specified, but most benchmarks out there only loosely follow the specification.
Partitions

- Turns out, most OLTP workloads mostly partitionable
- TPC-C is about 90% partitionable
- Perform 2PC only for the remaining 10%
H-Store Architecture

- Stored Procedures Only
- Partitioned + Replicated
- Differentiates between:
  - Single-site Transactions
  - Others

H-Store: Performance

- Vanilla H-Store can do 70K TPC-C txns compared to a couple thousand from before

- At the time, TPC-C record was about 133 Ktxn/s on a 128 core server.
  - H-Store can do half of that on low-end desktops.
H-Store: Further Optimizations

- Speculatively execute transactions when blocked
- Predict transaction behavior
- Partition database and schedule work intelligently to minimize percentage of distributed transactions
H-Store: Further Optimizations

- Speculatively execute transactions when blocked

- Predict transaction behavior

- Partition database and schedule work intelligently to minimize percentage of distributed transactions
H-Store: Speculative Execution

- Recall: H-Store single-threaded
- Also recall: 2PC takes > 10 ms to complete
- A partition simply waits out the 10ms instead of doing work
H-Store: Speculative Execution

● Observation: Most transactions succeed

● Idea: Assume transaction succeeds. Do useful work.

● 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 1: Only speculatively execute when waiting for 2PC
  ○ No locks required
  ○ Speculative results held back until 2PC finishes
  ○ Record undo information in-memory

● Idea 2: Speculate whenever stalled
  ○ E.g., when a multi-partition transaction is reading a remote value
  ○ Locking required
  ○ Overhead lower than 2PL, since no latching
  ○ Still expensive
H-Store: Speculative Execution

- Synthetic benchmark --- single operation transactions
- Baseline no conflict

![Graph showing throughput vs. multi-partition]
H-Store: Speculative Execution

- TPC-C
- Locking overhead increases with complex workload
- Speculation better
H-Store: Further Optimizations

- Speculatively execute transactions when blocked
- Predict transaction behavior
- Partition database and schedule work intelligently to minimize percentage of distributed transactions
H-Store: Predictive Modeling

- Observe: many transaction optimizations available. None applies to all situations.
H-Store: Predictive Modeling

- Guessing game to apply the right optimization

- Question: Can we make educated guesses instead?
H-Store: Predictive Modeling

- Answer: Yes.

- Use Markov model to observe behavior, predict probabilities and act accordingly*

* One might call this “Machine Learning” in 2021, but 2012 was a simpler time.

H-Store: Predictive Modeling

Transaction Behavior Prediction

Slide courtesy of Prof. Andy Pavlo
H-Store: Further Optimizations

- Speculatively execute transactions when blocked
- Predict transaction behavior
- Partition database and schedule work intelligently to minimize percentage of distributed transactions
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: Partitioning

- Hiring someone to do partitioning is expensive and unreliable
- Can automate in H-Store with Large Neighborhood Search
- Details omitted

H-Store: Partitioning

Slide courtesy of Prof. Andy Pavlo
Takeaway

● The Good
  ○ Specialization is good
  ○ Partitioning is really powerful
  ○ Seemingly simple optimizations can lead to huge speed-ups

● The Not-so-good
  ○ If you really need distributed transactions, you are out of luck*

* People are still working on it!
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
Deterministic Transactions

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

- However: Calvin fixes the schedule before execution

- Therefore: coordination also largely done 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
- We still need to track progress of replicas to give guarantees
Practical Considerations

- Sequencer is a bottleneck of the system and single-point of failure
- We still want concurrency for performance on a single node
- We still need to track progress of replicas to give guarantees
Sequencer: Initial Attempt
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
Key Idea: Epochs

- Recall: Coordination off critical path = win

- Idea: Synchronize loosely at set intervals (i.e., epochs)

- Wait for next epoch if uncertain

- Common idea in concurrent programming. Also used in single-node systems (e.g., Silo, Bw-Tree)
Example: Epoch

- Concurrently updating a buffer in-place
- Periodically “freeze” a log chunk to flush to disk

![Diagram showing concurrent updates and write process]

Shared Data Structure
Example: Epoch-based Sequencer

Global Txn Ordering

Sequencer 1

T1  T5
  T6

Seq 1, Epoch 11
  1, 5, 6

Sequencer 2

T2  T4

Seq 2, Epoch 11
  2, 4

Sequencer 3

T3  T7

Seq 3, Epoch 11
  3, 7
Example: Epoch-based Sequencer

Global Txn Ordering

- Sequencer 1: Seq 1, Epoch 11 (1, 5, 6)
- Sequencer 2: Seq 2, Epoch 11 (2, 4)
- Sequencer 3: Seq 3, Epoch 11 (3, 7)

- Now: Global ordering can be obtained through round-robin
What’s the price?

- Transactions are not sequenced until epoch end
- Recurring theme for epoch-based schemes: throughput vs. latency trade-off
- More on this later
Practical Considerations

- Sequencer is a bottleneck of the system and single-point of failure
- We still want concurrency for performance on a single node
- We still need to track progress of replicas to give guarantees
Scheduler: Deterministic Concurrency Control

Consider Schedule:

| Read A, Write B | Read C | Write D |

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

- We still need to track progress of replicas to give guarantees
Multi-node Transactions: Idea

- Read remotely if needed
  - No overhead

- Write locally

- Multi-node transaction is done when every participant done with local writes
  - Use locks/node progress along the serial order to ensure correct reads
Calvin: Results

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

- Synthetic Microbenchmark
Calvin: Results

- 100 % Multi-partition
- Y-axis is slow down factor
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
Attempt 3: COCO

- H-Store leveraged workload patterns

- Calvin/Aria introduces sequencer and increases latency

- Can we attack the problem of 2PC head-on?
  - Surprisingly, yes. Well, sometimes.
  - Area of (very) recent work
- Multiple transactions perform 2PC together at epoch granularity

- Failures result in all transactions within an epoch to fail together (does not include conflict aborts or user aborts)

- Replicas kept up-to-date at epoch boundary

Yi Lu, Xiangyao Yu, Lei Cao, Samuel Madden. Epoch-based Commit and Replication in Distributed OLTP Databases. VLDB 2021
COCO: Results

TPC-C Performance

Throughput (txns/sec)

- 2PC w/o Replication
- 2PC (Sync)
- Epoch (Async)

~4x improvement

# of messages

- 2PC w/o Replication
- 2PC (Sync)
- Epoch (Async)
COCO: Results over WAN

- Recall: Throughput vs. Latency trade-off
Epochs: Takeaways

- Surprisingly effective in alleviating bottleneck
- Throughput vs. Latency Trade-off
- General pattern in high-performance concurrent/distributed programming

Tianyu Li, Badrish Chandramouli, Samuel Madden. Performant Almost-Latch-Free Data Structures Using Epoch Protection. DaMoN 2022
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

- And we have the performance numbers to show off
  - Exciting.*
  - Txn throughput went from a couple of thousands to millions per second
  - Current record holder for TPC-C does 707 M TpmC
    - OceanBase from Alibaba’s Ant Financial

* Yes, call us nerds.
We Are Boring

Sam Madden
madden@csail.mit.edu

AI is enjoying a renaissance, with popular press and major corporations touting a variety of smart, AI-based applications, from self-driving cars to household robots to household gadgets that learn our behaviors and habits.

Despite all of these applications revolving around data, the database community to cede these domains to our AI colleagues. This is absurdly shortsighted. If not checked, the world-wide web, and (nearly) big data, we risk being an also-ran to an also-ran in computer science in the coming decade. These smart systems will appear in all of our work, and play, and the database community ought to be thinking about how to leverage the tools of the AI revolution to build better systems.

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 administrators 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 Technology Center for Big Data and the U.S. National Science Foundation (CCF-1438955).

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

● Do we really need many more transactions per second?
  ○ Probably not
  ○ The most you will see is around 750 M req/s on China’s 11/11 Single’s Day
  ○ Most of this workload embarrassingly parallel
  ○ Ultimately OLTP demand driven by economic growth, not database speed

● Are these new algorithms practical?
  ○ Maybe. But less than you’d think.
  ○ TPC-C != real-world (many dirty tricks to squeeze numbers out of TPC-C)
  ○ People don’t need transactions all the time
  ○ Most people don’t write highly optimized stored procedures
Should we just call it done?

Yes and No.

- (Almost) nobody cares if you improve TPC-C by 10%
- Guarantees are nice though
  - Many “NoSQL” systems ended up retrofitting to add transactions and/or strong consistency.
- New hardware
  - E.g. NVM and RDMA can change the equation (e.g., Microsoft’s FaRM)
- Cloud-native?
  - Traditional DBMSs fundamentally designed for shared-nothing
  - Cloud services change that
  - Open question on how to build performant, cheap, elastic cloud-native DBMS
Transactions in the Cloud

● Several different assumptions:
  ○ Disks are not persistent – must replicate across availability zones and even data centers
  ○ Strong primitives such as highly-available shared object storage exist
  ○ Economics dominate – people care about paying for just what they need

● Recent years saw a number of “cloud-native” database systems
  ○ E.g., AWS Aurora, Microsoft Socrates, SingleStore, FoundationDB, etc.
  ○ Most of these systems build on top of existing cloud storage services in “shared-disk” fashion
  ○ Most of these systems separate compute and storage for flexibility and service-ify important components for performance (e.g., logging)
  ○ “Serverless” is the hot word where these systems hide away infrastructure details and charge users on a per request basis
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
- Buffer pool loads data from S3, evicts data by erasing it
  - I.e., used exclusively as a cache
- Optimized logging layer to reduce commit latency and materialize pages in S3 by replaying
Cloud-native OLTP

- Area of active research to investigate what is fundamentally new about cloud-native databases
  - Example: can simplify protocol as underlying cloud storage guarantees consensus, fault-tolerance, etc.
  - Other potential directions: leveraging autoscaling serverless functions, converting existing components into multi-tenant services, etc.

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 real is debatable
  - Regardless, many of the innovations run in production today

- Transactions research is alive and well in new settings such as the autoscaling cloud