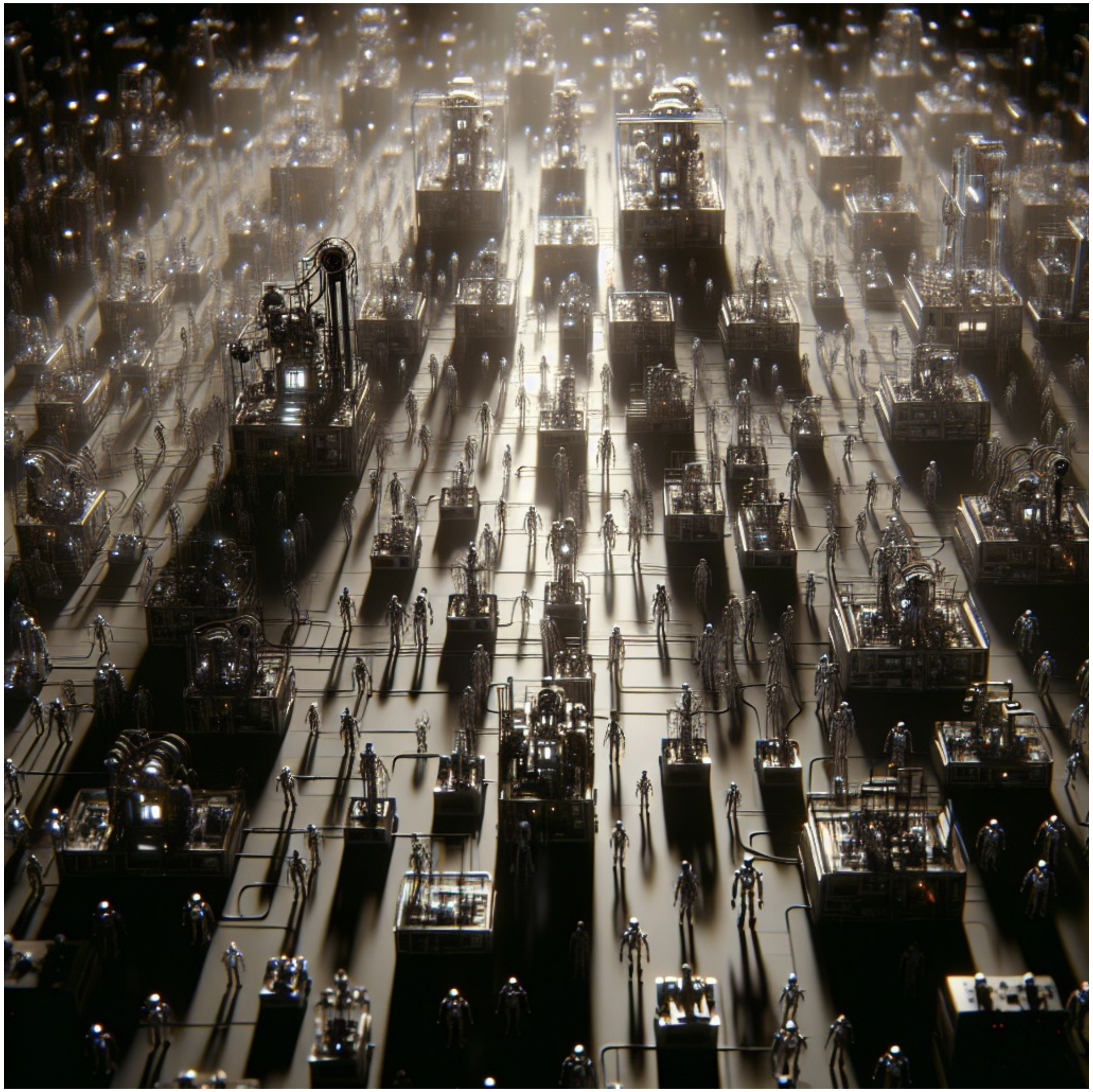


High-Performance Transactions

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

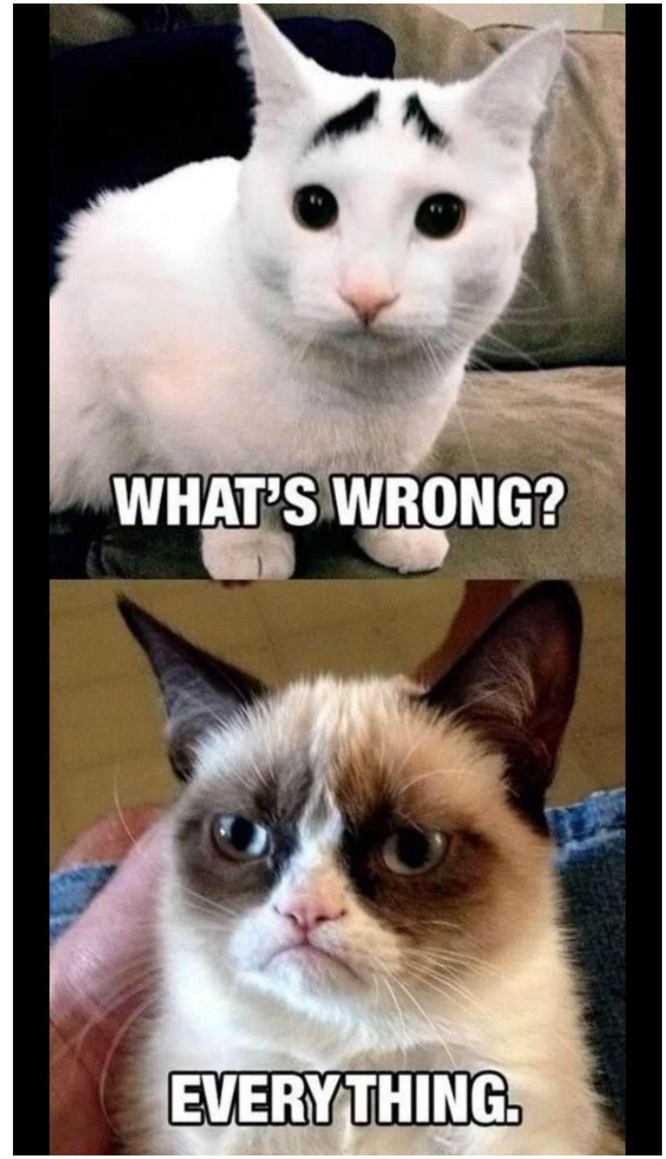
Recap - Transactions

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?



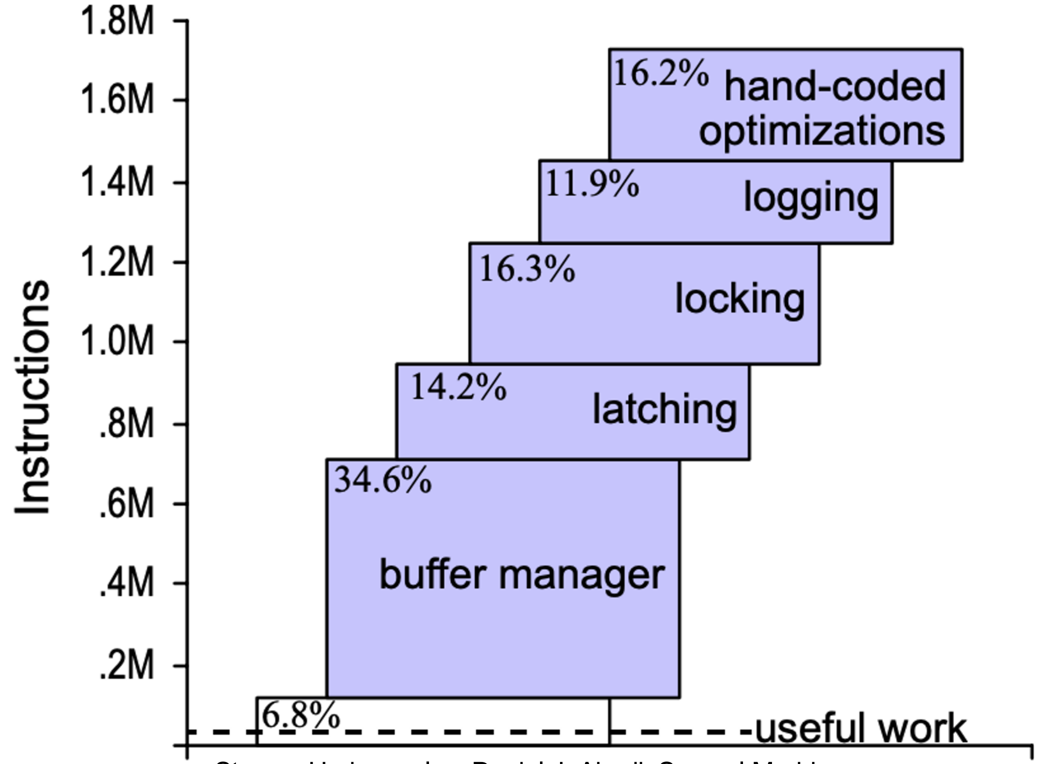
Times Are Different

1980s	Now
Slow Networks (< 10 Mb/sec)	40+ Gb/sec
Small number of on-prem machines	Global-scale, cloud
Single or few-core	100+cores
Few MB of memory	100+GB RAM / machine

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

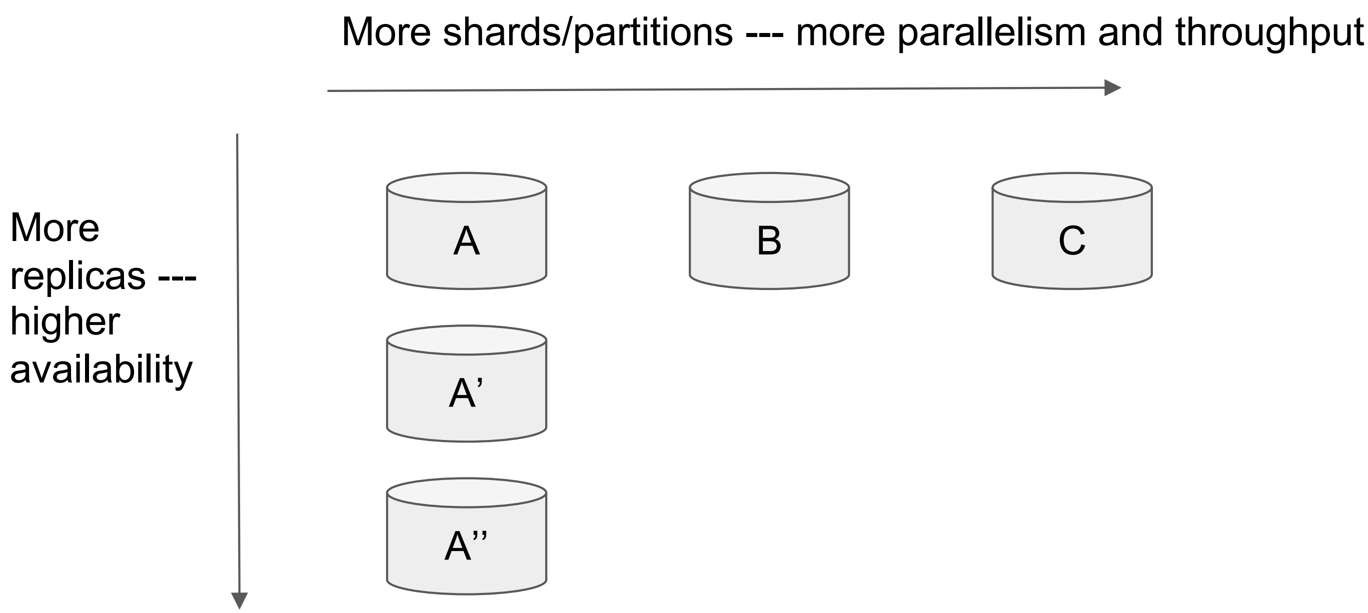


Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. SIGMOD 2008

Today -- High Performance Transactions

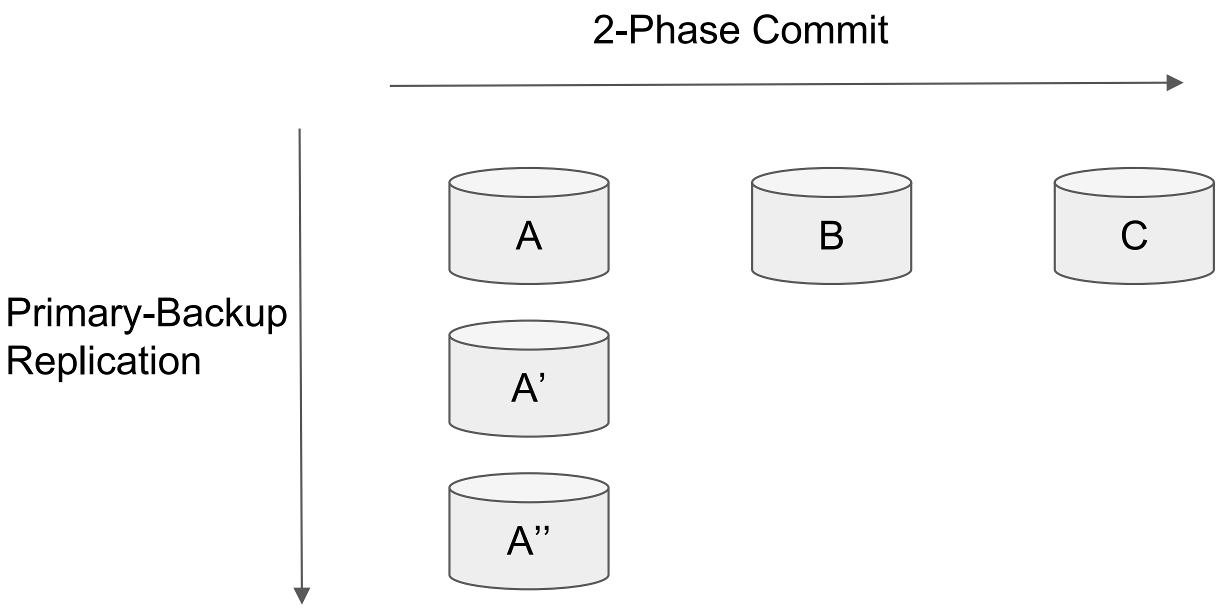
- Looking Back
- Multi-node
  - Bottleneck: 2-Phase Commit
  - Single-Site Execution
  - Deterministic Transactions
- Cloud Transactions

Recap: Scaling a Database



* Replicas usually also serve read requests

Recap: Scaling a Database

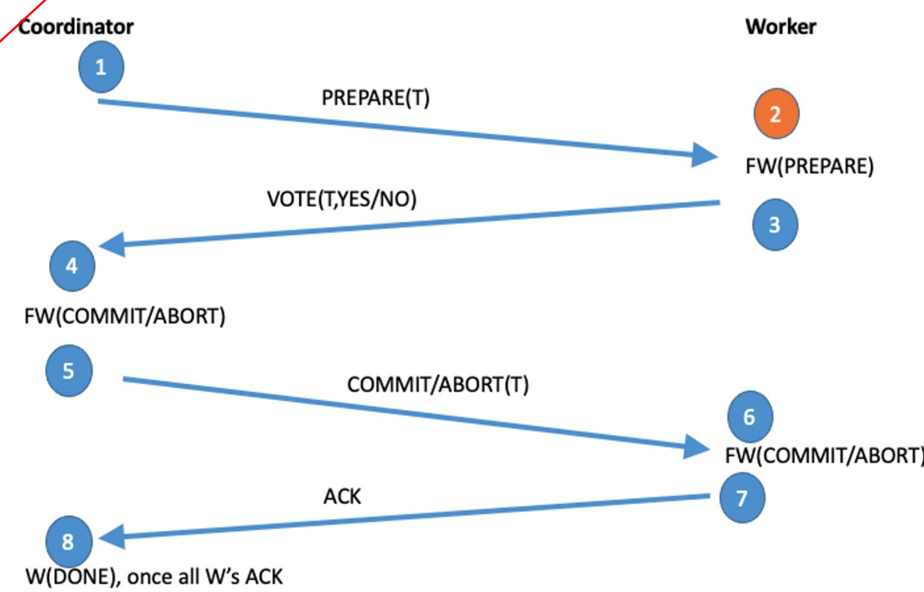


Unless we are careful, replication hurts write performance, but increases availability

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

Commit Point



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)

Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

1 Introduction

Spanner is a scalable, globally-distributed database designed, built, and deployed at Google. At the highest level of abstraction, it is a database that shards data across many sets of Paxos [21] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows.

Applications can use Spanner for high availability, even in the face of wide-area natural disasters, by replicating their data within or even across continents. Our initial customer was F1 [35], a rewrite of Google's advertising backend. F1 uses five replicas spread across

tency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in schematized semi-relational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps. Spanner supports general-purpose transactions, and provides a SQL-based query language.

As a globally-distributed database, Spanner provides several interesting features. First, the replication configurations for data can be dynamically controlled at a fine grain by applications. Applications can specify constraints to control which datacenters contain which data, how far data is from its users (to control read latency), how far replicas are from each other (to control write latency), and how many replicas are maintained (to control durability, availability, and read performance). Data

Corbett et. al. Spanner: Google's Globally-Distributed Database. OSDI 2012

Problem

2PC Scalability

participants	latency (ms)	
	mean	99th percentile
1	17.0 ±1.4	75.0 ±34.9
2	24.5 ±2.5	87.6 ±35.9
5	31.5 ±6.2	104.5 ±52.2
10	30.0 ±3.7	95.6 ±25.4
25	35.5 ±5.6	100.4 ±42.7
50	42.7 ±4.1	93.7 ±22.9
100	71.4 ±7.6	131.2 ±17.6
200	150.5 ±11.0	320.3 ±35.1

2PC end-to-end Latency

operation	latency (ms)		count
	mean	std dev	
all reads	8.7	376.4	21.5B
single-site commit	72.3	112.8	31.2M
multi-site commit	103.0	52.2	32.1M

- 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

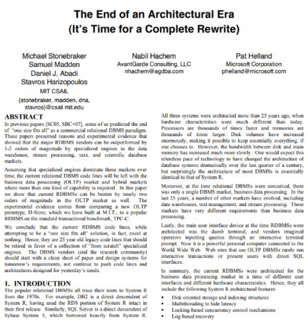
Why bother with distributed transactions then?

- 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



The End of an Architectural Era
(It's Time for a Complete Rewrite)

Michael Stonebraker
Samuel Madden
Daniel J. Abadi
Stavros Harizopoulos
MIT CSAIL
{stonebraker, madden, dna, stavros}@csail.mit.edu

Nabil Hachem
AvantGarde Consulting, LLC
nhachem@agdba.com

Pat Helland
Microsoft Corporation
phelland@microsoft.com

ABSTRACT

In previous papers [SC05, SBC+07], some of us predicted the end of "one size fits all" as a commercial relational DBMS paradigm. These papers presented reasons and experimental evidence that showed that the major RDBMS vendors can be outperformed by 1-2 orders of magnitude by specialized engines in the data warehouse, stream processing, text, and scientific database markets.

Assuming that specialized engines dominate these markets over time, the current relational DBMS code lines will be left with the business data processing (OLTP) market and hybrid markets where more than one kind of capability is required. In this paper we show that current RDBMSs can be beaten by nearly two orders of magnitude in the OLTP market as well. The experimental evidence comes from comparing a new OLTP prototype, H-Store, which we have built at M.I.T., to a popular RDBMS on the standard transactional benchmark, TPC-C.

We conclude that the current RDBMS code lines, while attempting to be a "one size fits all" solution, in fact, excel at nothing. Hence, they are 25 year old legacy code lines that should be retired in favor of a collection of "from scratch" specialized engines. The DBMS vendors (and the research community) should start with a clean sheet of paper and design systems for tomorrow's requirements, not continue to push code lines and architectures designed for yesterday's needs.

1. INTRODUCTION

The popular relational DBMSs all trace their roots to System R from the 1970s. For example, DB2 is a direct descendent of System R, having used the RDS portion of System R intact in their first release. Similarly, SQL Server is a direct descendent of Sybase System 5, which borrowed heavily from System R.

All three systems were architected more than 25 years ago, when hardware characteristics were much different than today. Processors are thousands of times faster and memories are thousands of times larger. Disk volumes have increased enormously, making it possible to keep essentially everything, if one chooses to. However, the bandwidth between disk and main memory has increased much more slowly. One would expect this relentless pace of technology to have changed the architecture of database systems dramatically over the last quarter of a century, but surprisingly the architecture of most DBMSs is essentially identical to that of System R.

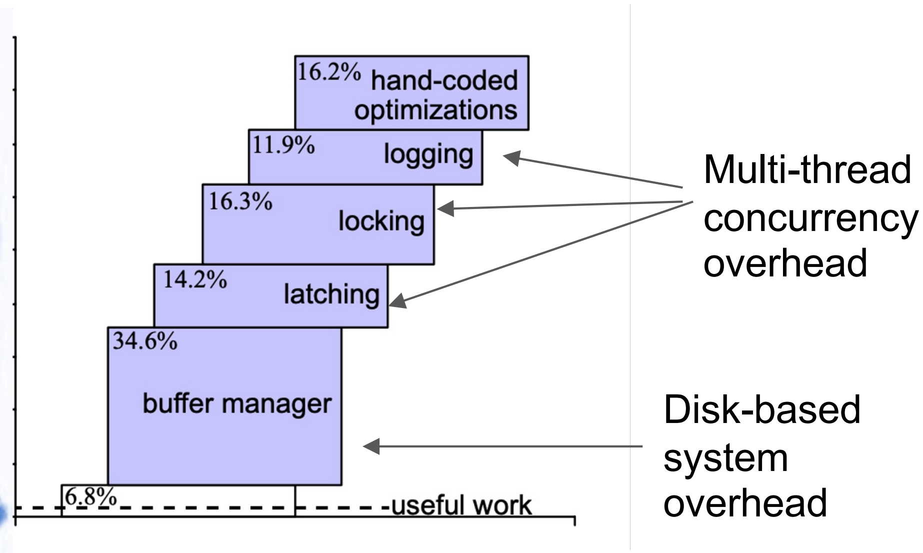
Moreover, at the time relational DBMSs were conceived, there was only a single DBMS market, business data processing. In the last 25 years, a number of other markets have evolved, including data warehouses, text management, and stream processing. These markets have very different requirements than business data processing.

Lastly, the main user interface device at the time RDBMSs were architected was the dumb terminal, and vendors imagined operators inputting queries through an interactive terminal prompt. Now it is a powerful personal computer connected to the World Wide Web. Web sites that use OLTP DBMSs rarely run interactive transactions or present users with direct SQL interfaces.

In summary, the current RDBMSs were architected for the business data processing market in a time of different user interfaces and different hardware characteristics. Hence, they all include the following System R architectural features:

- Disk oriented storage and indexing structures
- Multithreading to hide latency
- Locking-based concurrency control mechanisms
- Log-based recovery

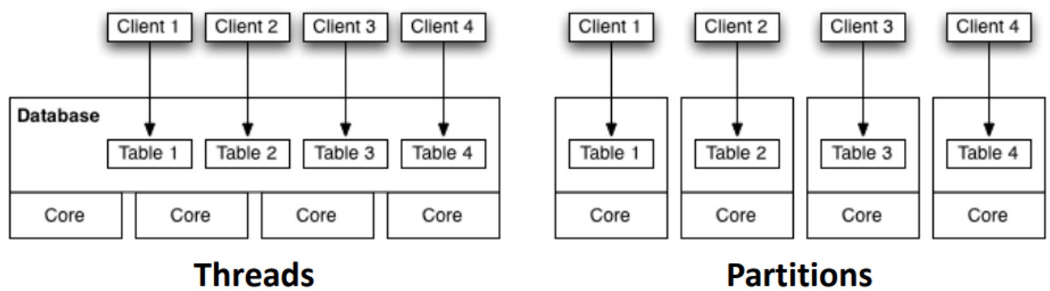
How to make transaction processing databases 10x faster?



- Eliminate
  - Disk I/O
  - Locking
  - Concurrency C
  - Disk based rec
- Sounds nuts, but g e
  - 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

Example stored procedure:

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

- 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

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

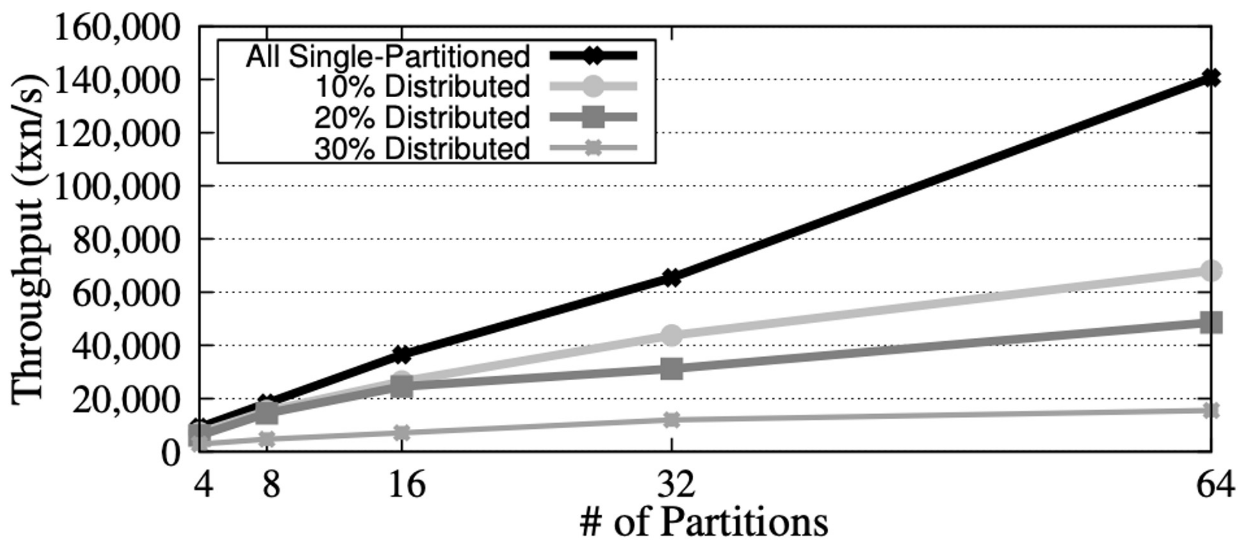
H-Store: Performance

- 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 txn/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



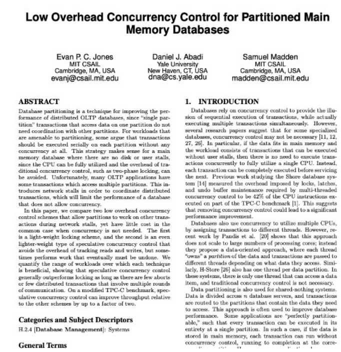
H-Store: Speculative Execution

- 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



Low Overhead Concurrency Control for Partitioned Main Memory Databases

Evan P. C. Jones
MIT CSAIL
Cambridge, MA, USA
evanj@csail.mit.edu

Daniel J. Abadi
Yale University
New Haven, CT, USA
dna@cs.yale.edu

Samuel Madden
MIT CSAIL
Cambridge, MA, USA
madden@csail.mit.edu

ABSTRACT

Database partitioning is a technique for improving the performance of distributed OLTP databases, since "single partition" transactions that access data on one partition do not need coordination with other partitions. For workloads that are amenable to partitioning, some argue that transactions should be executed serially on each partition without any concurrency at all. This strategy makes sense for a main memory database where there are no disk or user stalls, since the CPU can be fully utilized and the overhead of traditional concurrency control, such as two-phase locking, can be avoided. Unfortunately, many OLTP applications have some transactions which access multiple partitions. This introduces network stalls in order to coordinate distributed transactions, which will limit the performance of a database that does not allow concurrency.

In this paper, we compare two low overhead concurrency control schemes that allow partitions to work on other transactions during network stalls, yet have little cost in the common case when concurrency is not needed. The first is a light-weight locking scheme, and the second is an even lighter-weight type of speculative concurrency control that avoids the overhead of tracking reads and writes, but sometimes performs work that eventually must be undone. We quantify the range of workloads over which each technique is beneficial, showing that speculative concurrency control generally outperforms locking as long as there are few aborts or few distributed transactions that involve multiple rounds of communication. On a modified TPC-C benchmark, speculative concurrency control can improve throughput relative to the other schemes by up to a factor of two.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

1. INTRODUCTION

Databases rely on concurrency control to provide the illusion of sequential execution of transactions, while actually executing multiple transactions simultaneously. However, several research papers suggest that for some specialized databases, concurrency control may not be necessary [11, 12, 27, 26]. In particular, if the data fits in main memory and the workload consists of transactions that can be executed without user stalls, then there is no need to execute transactions concurrently to fully utilize a single CPU. Instead, each transaction can be completely executed before servicing the next. Previous work studying the Shore database system [14] measured the overhead imposed by locks, latches, and undo buffer maintenance required by multi-threaded concurrency control to be 42% of the CPU instructions executed on part of the TPC-C benchmark [1]. This suggests that removing concurrency control could lead to a significant performance improvement.

Databases also use concurrency to utilize multiple CPUs, by assigning transactions to different threads. However, recent work by Pandis et al. [20] shows that this approach does not scale to large numbers of processing cores; instead they propose a data-oriented approach, where each thread "owns" a *partition* of the data and transactions are passed to different threads depending on what data they access. Similarly, H-Store [26] also has one thread per data partition. In these systems, there is only one thread that can access a data item, and traditional concurrency control is not necessary.

Data partitioning is also used for shared-nothing systems. Data is divided across n database servers, and transactions are routed to the partitions that contain the data they need to access. This approach is often used to improve database performance. Some applications are "perfectly partitionable," such that every transaction can be executed in its entirety at a single partition. In such a case, if the data is stored in main memory, each transaction can run without concurrency control, running to completion at the corre-

H-Store: Speculative Execution

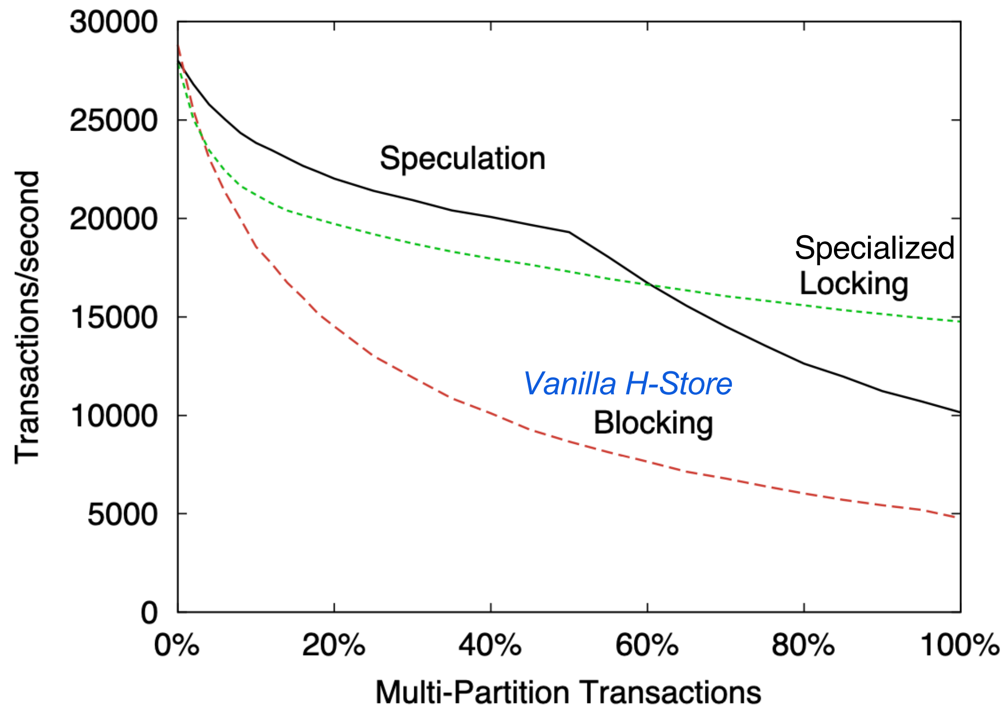
- Idea: Speculate when waiting for 2PC outcome (e.g., after the transaction has completed)
    - No locks required
    - Local transactions execute assuming 2PC will complete
    - Results held back until 2PC finishes
        - Record undo information in-memory
    - If 2PC fails, all speculated transactions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

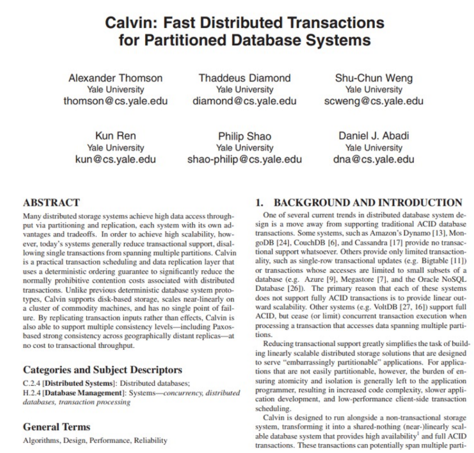
- Can the same idea be applied to truly distributed transactions?

Key Idea: Calvin

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

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

Alexander Thomson et. al. Calvin: Fast Distributed Transactions for Partitioned Database Systems. SIGMOD 2012.



Calvin: Fast Distributed Transactions for Partitioned Database Systems

Alexander Thomson, Yale University, thomson@cs.yale.edu
Thaddeus Diamond, Yale University, diamond@cs.yale.edu
Shu-Chun Weng, Yale University, scweng@cs.yale.edu
Kun Ren, Yale University, kun@cs.yale.edu
Philip Shao, Yale University, shao-philip@cs.yale.edu
Daniel J. Abadi, Yale University, dna@cs.yale.edu

ABSTRACT

Many distributed storage systems achieve high data access throughput via partitioning and replication, each system with its own advantages and tradeoffs. In order to achieve high scalability, however, today's systems generally reduce transactional support, disallowing single transactions from spanning multiple partitions. Calvin is a practical transaction scheduling and data replication layer that uses a deterministic ordering guarantee to significantly reduce the normally prohibitive contention costs associated with distributed transactions. Unlike previous deterministic database system prototypes, Calvin supports disk-based storage, scales near-linearly on a cluster of commodity machines, and has no single point of failure. By replicating transaction inputs rather than effects, Calvin is also able to support multiple consistency levels—including Paxos-based strong consistency across geographically distant replicas—at no cost to transactional throughput.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed databases;
H.2.4 [**Database Management**]: Systems—*concurrency, distributed databases, transaction processing*

General Terms

Algorithms, Design, Performance, Reliability

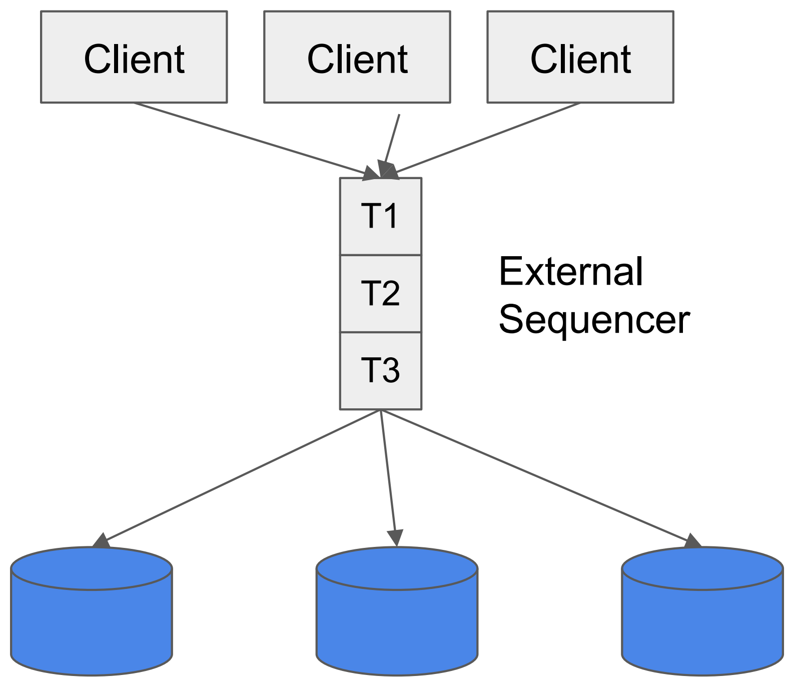
1. BACKGROUND AND INTRODUCTION

One of several current trends in distributed database system design is a move away from supporting traditional ACID database transactions. Some systems, such as Amazon's Dynamo [13], MongoDB [24], CouchDB [6], and Cassandra [17] provide no transactional support whatsoever. Others provide only limited transactionality, such as single-row transactional updates (e.g. Bigtable [11]) or transactions whose accesses are limited to small subsets of a database (e.g. Azure [9], Megastore [7], and the Oracle NoSQL Database [26]). The primary reason that each of these systems does not support fully ACID transactions is to provide linear outward scalability. Other systems (e.g. VoltDB [27, 16]) support full ACID, but cease (or limit) concurrent transaction execution when processing a transaction that accesses data spanning multiple partitions.

Reducing transactional support greatly simplifies the task of building linearly scalable distributed storage solutions that are designed to serve "embarrassingly partitionable" applications. For applications that are not easily partitionable, however, the burden of ensuring atomicity and isolation is generally left to the application programmer, resulting in increased code complexity, slower application development, and low-performance client-side transaction scheduling.

Calvin is designed to run alongside a non-transactional storage system, transforming it into a shared-nothing (near-)linearly scalable database system that provides high availability[1] and full ACID transactions. These transactions can potentially span multiple parti-

Deterministic Transactions



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

    ```
    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

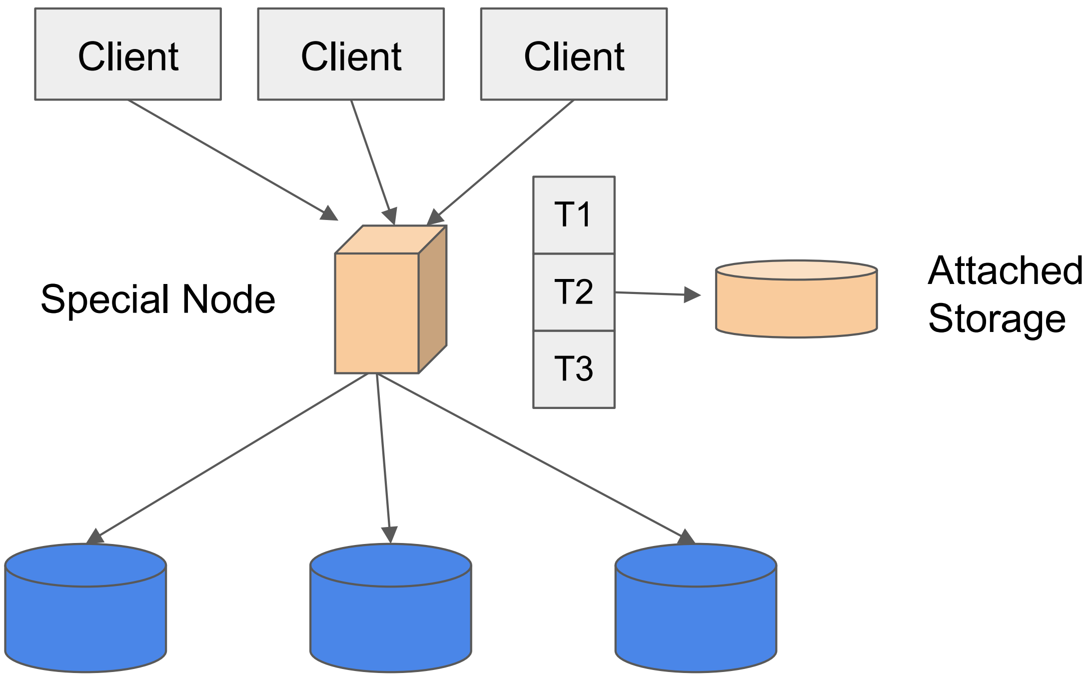
Practical Considerations

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

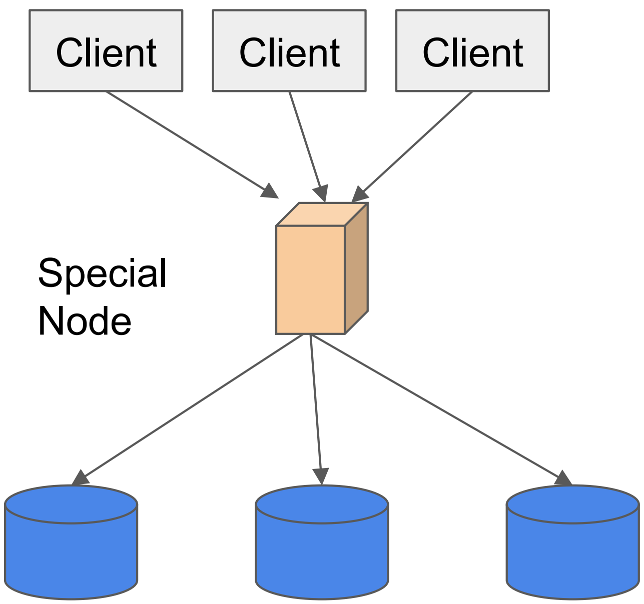
- We still want concurrency for performance on a single node

- Need to be recoverable / durable

Sequencer: Initial Attempt

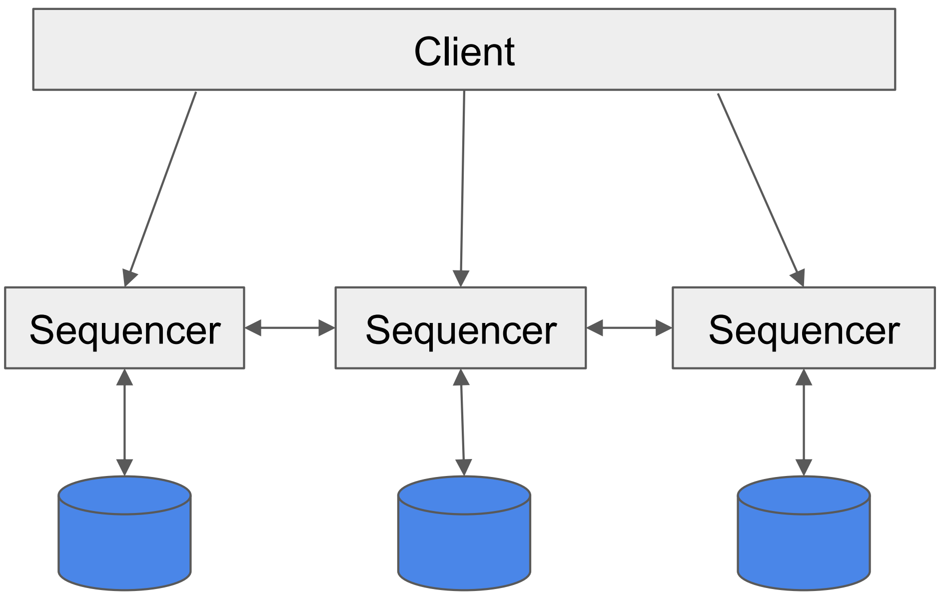


Sequencer: Initial Attempt



- 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

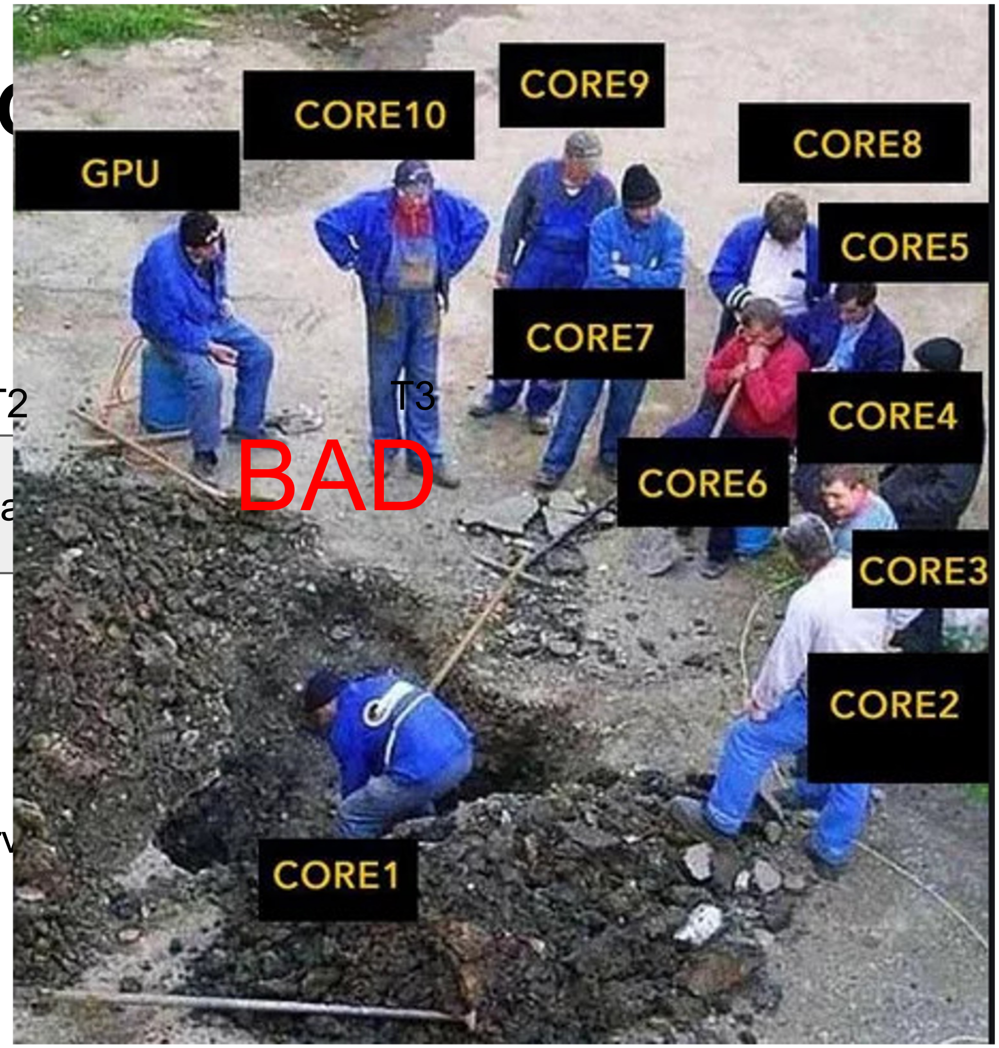
Practical Considerations

- 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 (

Consider Schedule:

T1	T2
Read A, Write B	Rea



- No actual conflict
- No reason to execute in-order
- Challenge: concurrent execution that preserv

Scheduler: Deterministic Concurrency Control

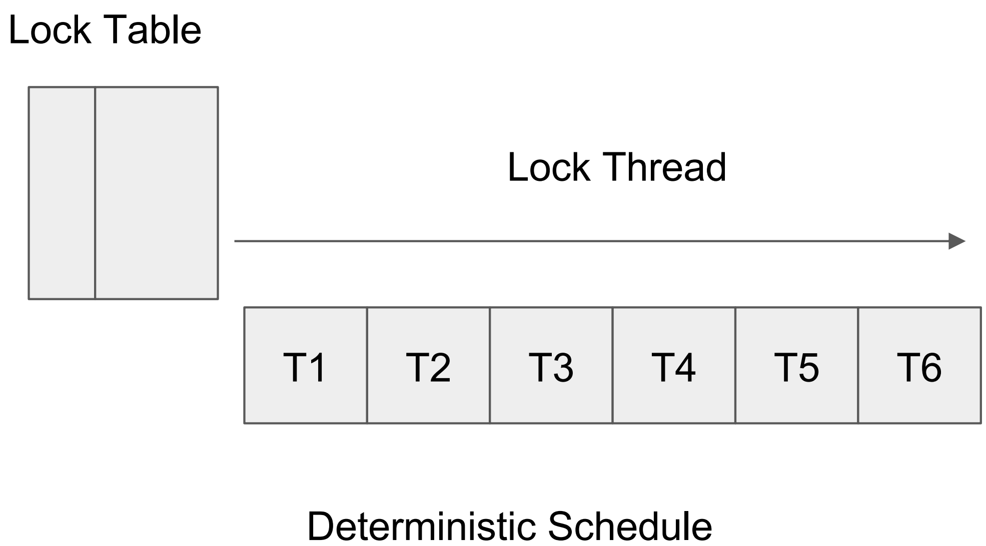
- Need to allow for concurrent execution

- 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

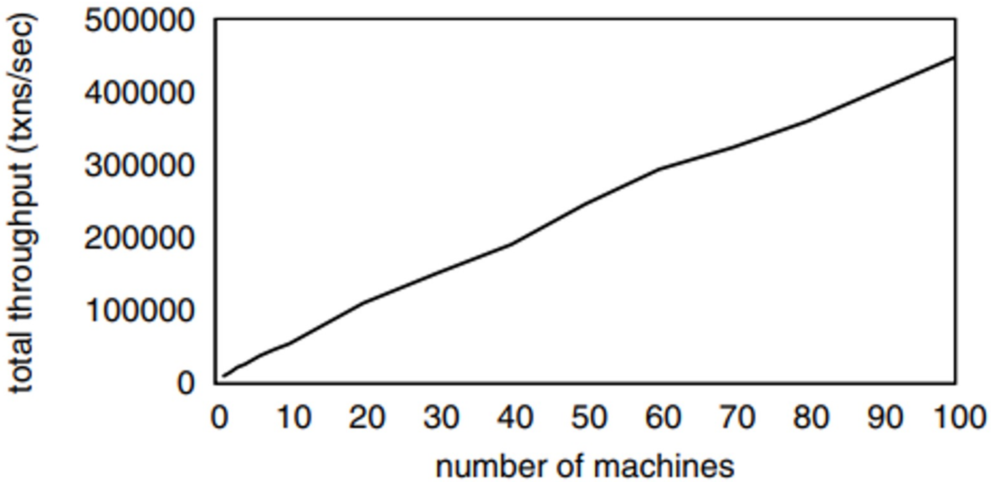
Logging and Checkpoints

- 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

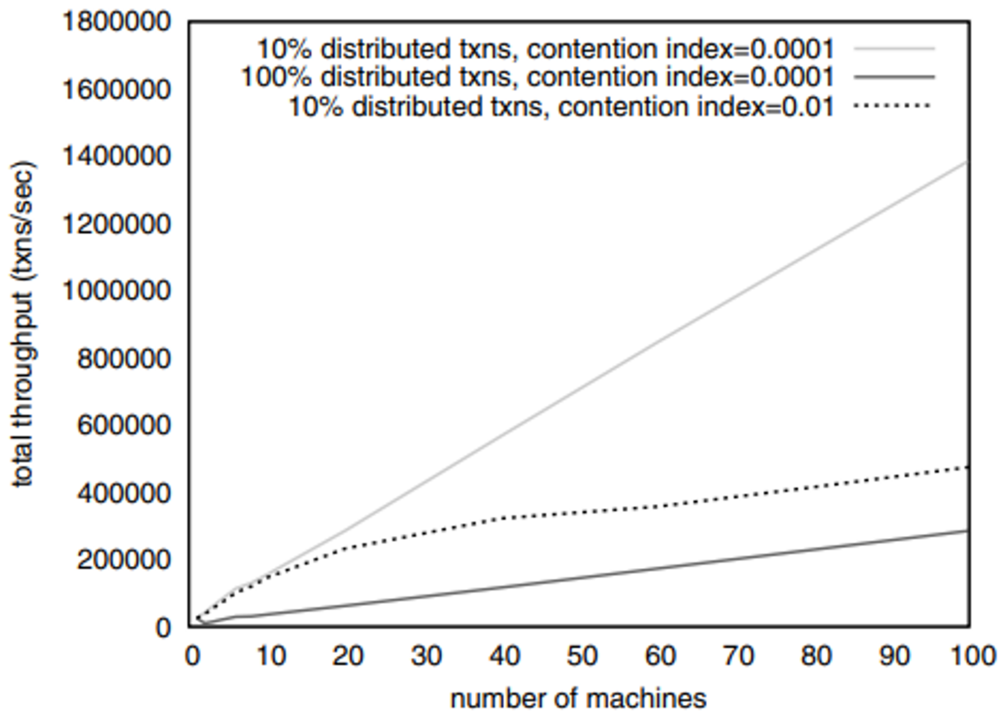
- Checkpointing needed
  - Otherwise, on failure, have to replay from the beginning of time
  - Need a way to take transaction-consistent snapshots

Calvin: Results

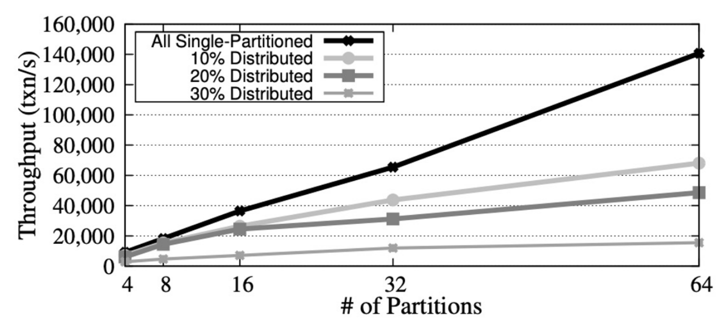


- TPC-C (100% New Order)

Calvin: Results



- Synthetic Microbenchmark



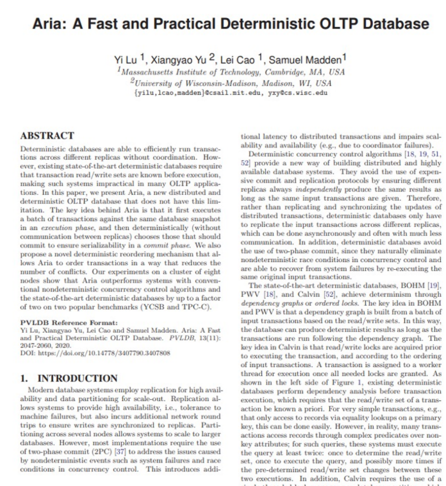
Calvin: Criticism

- 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

Yi Lu, Xiangyao Yu, Lei Cao, Samuel Madden. Aria: A Fast and Practical Deterministic OLTP Database. VLDB 2020.



Aria: A Fast and Practical Deterministic OLTP Database

Yi Lu [1], Xiangyao Yu [2], Lei Cao [1], Samuel Madden[1]
[1]Massachusetts Institute of Technology, Cambridge, MA, USA
[2]University of Wisconsin-Madison, Madison, WI, USA
{yilu,lcao,madden}@csail.mit.edu, yxy@cs.wisc.edu

ABSTRACT

Deterministic databases are able to efficiently run transactions across different replicas without coordination. However, existing state-of-the-art deterministic databases require that transaction read/write sets are known before execution, making such systems impractical in many OLTP applications. In this paper, we present Aria, a new distributed and deterministic OLTP database that does not have this limitation. The key idea behind Aria is that it first executes a batch of transactions against the same database snapshot in an *execution phase*, and then deterministically (without communication between replicas) chooses those that should commit to ensure serializability in a *commit phase*. We also propose a novel deterministic reordering mechanism that allows Aria to order transactions in a way that reduces the number of conflicts. Our experiments on a cluster of eight nodes show that Aria outperforms systems with conventional nondeterministic concurrency control algorithms and the state-of-the-art deterministic databases by up to a factor of two on two popular benchmarks (YCSB and TPC-C).

PVLDB Reference Format:
Yi Lu, Xiangyao Yu, Lei Cao and Samuel Madden. Aria: A Fast and Practical Deterministic OLTP Database. *PVLDB*, 13(11): 2047-2060, 2020.
DOI: https://doi.org/10.14778/3407790.3407808

1. INTRODUCTION

Modern database systems employ replication for high availability and data partitioning for scale-out. Replication allows systems to provide high availability, i.e., tolerance to machine failures, but also incurs additional network round trips to ensure writes are synchronized to replicas. Partitioning across several nodes allows systems to scale to larger databases. However, most implementations require the use of two-phase commit (2PC) [37] to address the issues caused by nondeterministic events such as system failures and race conditions in concurrency control. This introduces additional latency to distributed transactions and impairs scalability and availability (e.g., due to coordinator failures).

Deterministic concurrency control algorithms [18, 19, 51, 52] provide a new way of building distributed and highly available database systems. They avoid the use of expensive commit and replication protocols by ensuring different replicas always *independently* produce the same results as long as the same input transactions are given. Therefore, rather than replicating and synchronizing the updates of distributed transactions, deterministic databases only have to replicate the input transactions across different replicas, which can be done asynchronously and often with much less communication. In addition, deterministic databases avoid the use of two-phase commit, since they naturally eliminate nondeterministic race conditions in concurrency control and are able to recover from system failures by re-executing the same original input transactions.

The state-of-the-art deterministic databases, BOHM [19], PWV [18], and Calvin [52], achieve determinism through *dependency graphs* or *ordered locks*. The key idea in BOHM and PWV is that a dependency graph is built from a batch of input transactions based on the read/write sets. In this way, the database can produce deterministic results as long as the transactions are run following the dependency graph. The key idea in Calvin is that read/write locks are acquired prior to executing the transaction, and according to the ordering of input transactions. A transaction is assigned to a worker thread for execution once all needed locks are granted. As shown in the left side of Figure 1, existing deterministic databases perform dependency analysis before transaction execution, which requires that the read/write set of a transaction be known a priori. For very simple transactions, e.g., that only access to records via equality lookups on a primary key, this can be done easily. However, in reality, many transactions access records through complex predicates over non-key attributes; for such queries, these systems must execute the query at least twice: once to determine the read/write set, once to execute the query, and possibly more times if the pre-determined read/write set changes between these two executions. In addition, Calvin requires the use of a single-threaded lock manager per database partition, which

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

Criticism

We Are Boring

Sam Madden
madden@csail.mit.edu

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

Despite all of these applications revolving around data, the datab
content to cede these domains to our AI colleagues. This is absurdly
the world-wide web, and (nearly) big data, we risk being an also-ra
in computer science in the coming decade. These smart systems wil
work, and play, and the database community ought to be thinking

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

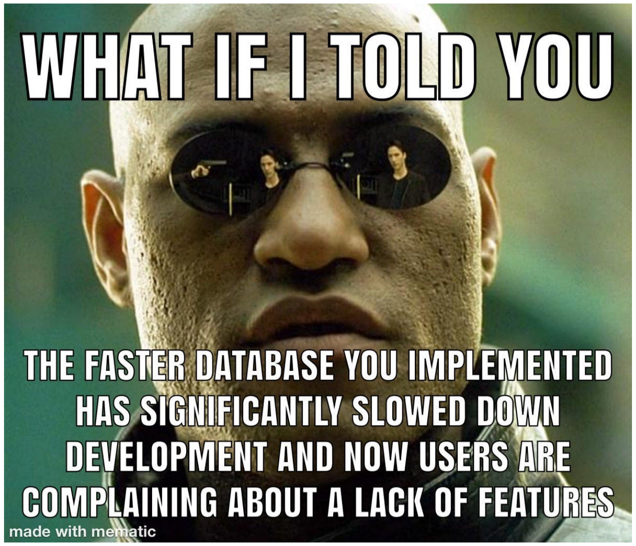
This work was supported (in part) by the Intel Science and Technology Center for Big Data and the U.S. National Science Foundation (CCF-1438955).

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

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

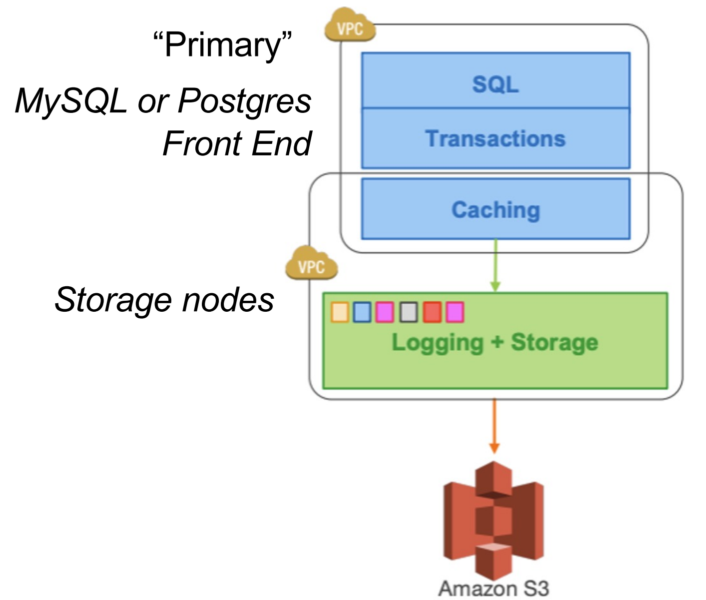
Cloud-native OLTP

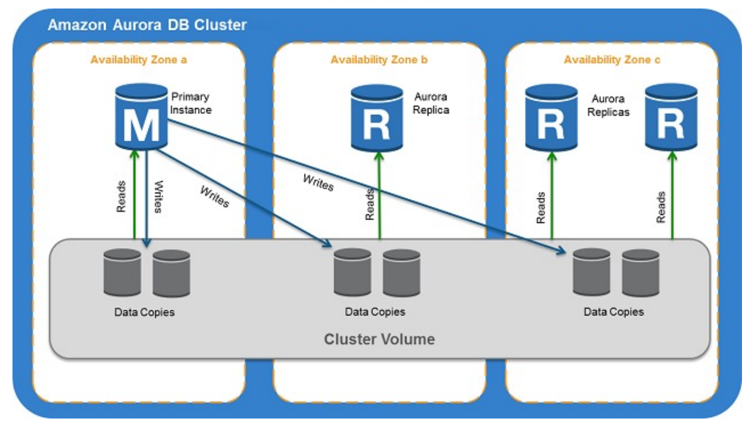
- Key Idea: Storage & Compute Separation
  - Use cloud object storage (e.g., S3) for persistent storage layer
  - Attach ephemeral machines to storage when needed
  - Allows for separate scaling of resources

- 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)

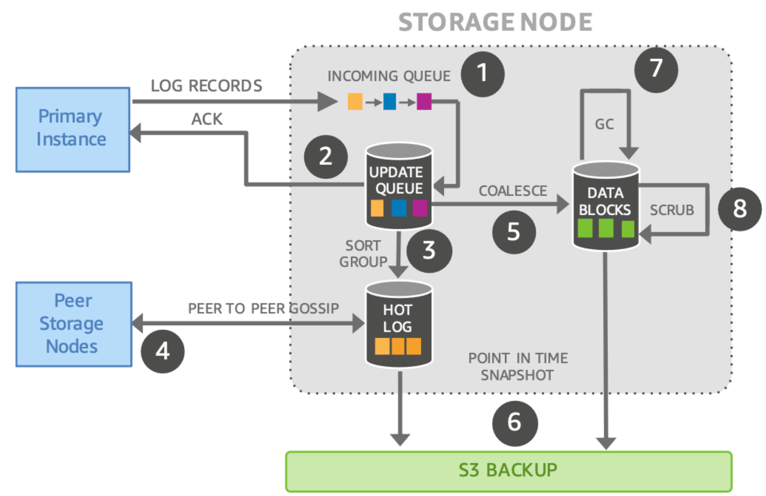
Example: Amazon Aurora

- 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,

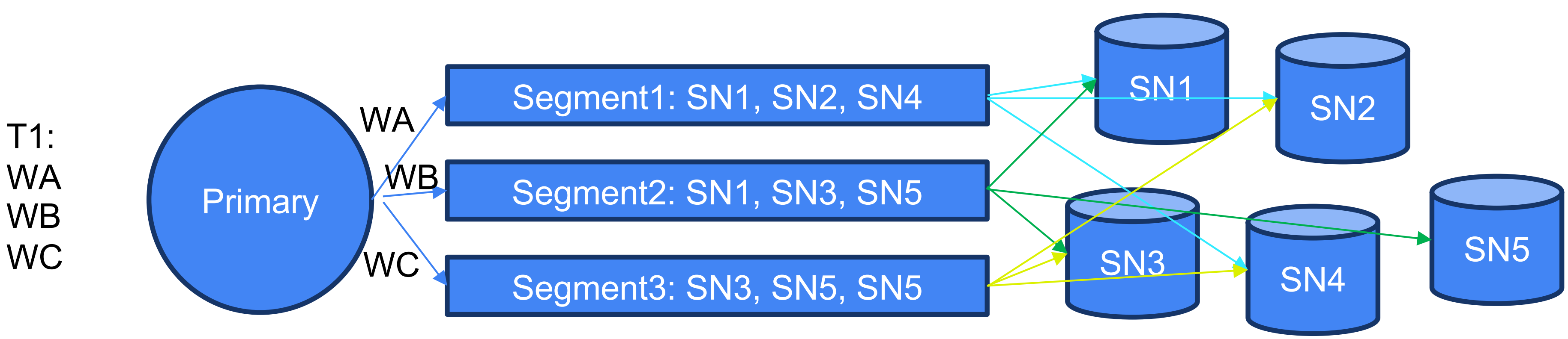
(6) backs them up to S3,

(7) garbage collects backed-up data

(8) periodically verifies checksums continue to match the data on disk.

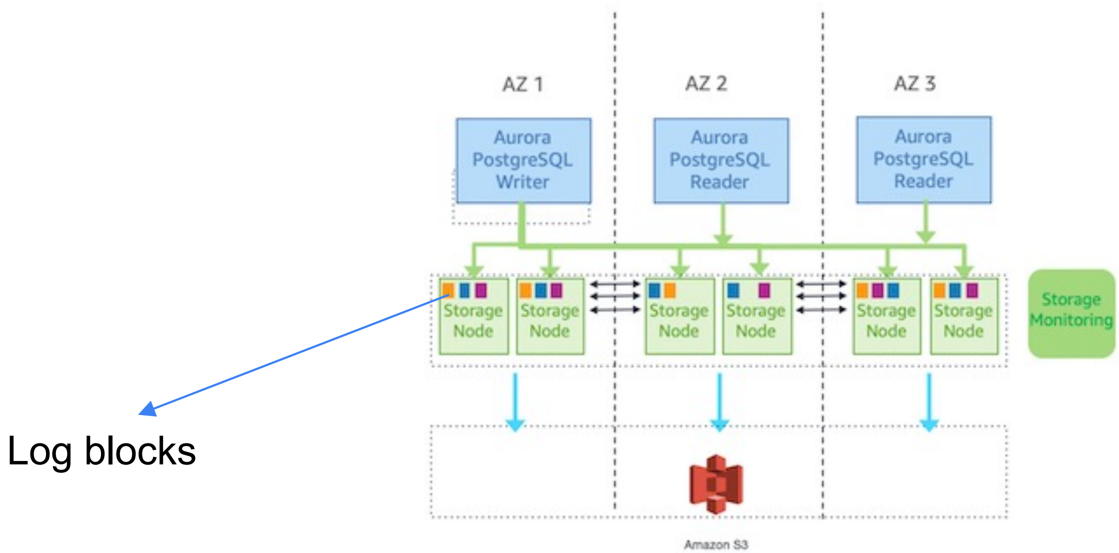
Storage

- 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



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

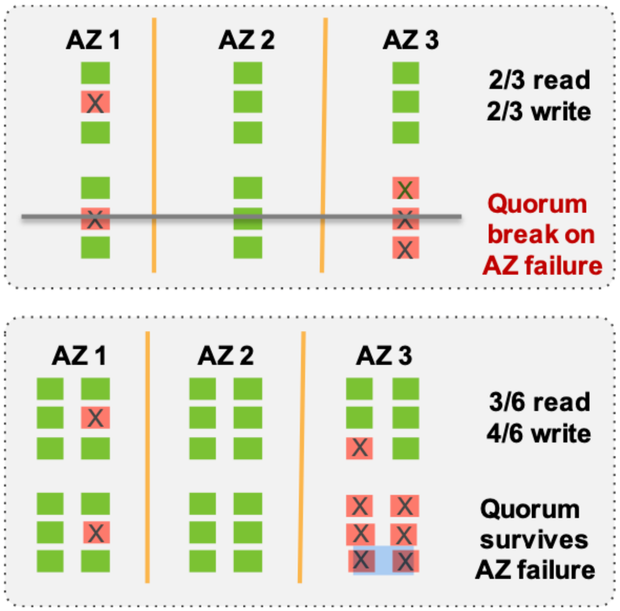


Figure 1: Why are 6 copies necessary ?

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

Industry 3: DB Systems in the Cloud and Open Source — SIGMOD'18, June 10-15, 2018, Houston, TX, USA

Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes

Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta
Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice
Tengiz Kharatishvilli, Xiaofeng Bao
Amazon Web Services

ABSTRACT

Amazon Aurora is a high-throughput cloud-native relational database offered as part of Amazon Web Services (AWS). One of the more novel differences between Aurora and other relational databases is how it pushes redo processing to a multi-tenant scale-out storage service, purpose-built for Aurora. Doing so reduces networking traffic, avoids checkpoints and crash recovery, enables failovers to replicas without loss of data, and enables fault-tolerant storage that heals without database involvement. Traditional implementations that leverage distributed storage would use distributed consensus algorithms for commits, reads, replication, and membership changes and amplify cost of underlying storage. In this paper, we describe how Aurora avoids distributed consensus under most circumstances by establishing invariants and leveraging local transient state. Doing so improves performance, reduces variability, and lowers costs.

KEYWORDS

Databases; Distributed Systems; Log Processing; Quorum Models; Fault tolerance; Quorum Sets; Replication; Recovery; Performance

ACM Reference Format:

Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, and Tengiz Kharatishvilli, Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3183713.3196937

1 INTRODUCTION

IT workloads are increasingly moving to public cloud providers such as AWS. Many of these workloads require a relational database. Amazon Relational Database Service (RDS) provides a managed service that automates database provisioning, operating system and database patching, backup, point-in-time restore, storage and compute scaling, instance health monitoring, failover, and other capabilities. Our experience managing hundreds of thousands of database instances in RDS led to the design requirements for Aurora, a high-throughput cloud-native relational database.

In our earlier paper [12], we provided an overview of the design considerations behind Aurora. A key contribution of that paper is to show that, on a fleet-wide basis, it is insufficient to treat failures as independent. At a minimum, it is necessary to consider the correlated impact of the largest unit of failure in addition to the background noise of on-going independent failures. In AWS, the largest unit of failure a system may need to tolerate is an Availability Zone (AZ). An AZ is a subset of a Region that is connected to other AZs through low-latency networking links, but is isolated for most faults, including power, networking, software deployments, flooding, and other phenomena. Aurora supports "AZ+1" failures, resulting in six copies of data, spread across three AZs, a 4/6 write quorum, and a 3/6 read quorum as illustrated in Figure 1. Aurora implements quorum membership changes to handle unexpected failures, heat management, as well as planned software upgrades.

Figure 1: Why are 6 copies necessary ?

Quorum models, such as the one used by Aurora, are rarely used in high-performance relational databases, despite the benefits they provide for availability, durability, and the reduction of latency jitter. We believe this is because the underlying distributed algorithms typically used in these systems – two-phase commit (2PC), Paxos commit, Paxos membership changes, and their variants – can be expensive and incur additional network overheads. The commercial systems we have seen built on these algorithms may scale well but have order-of-magnitude worse cost, performance, and peak to average latency than a traditional relational database running on a single node against local disk.

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196937

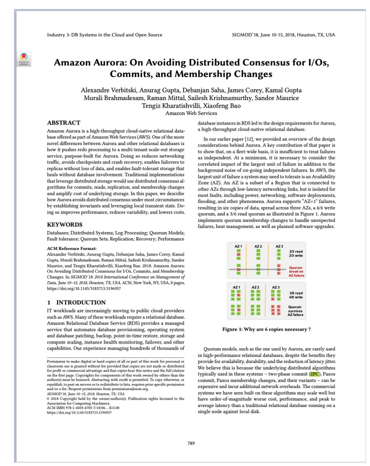
789

Commit

- Transactions may write data stored in different segments
- Since segments are on different replica sets, seems to require 2PC
- However:
    - Primary does concurrency control, properly sequences writes
        - No deadlocks will be generated on storage nodes
    - Quorum writes ensure that replica sets will not fail
        - As long as a quorum of nodes stays up

    ➔ 2PC isn't required
    - 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


Industry 3: DB Systems in the Cloud and Open Source — SIGMOD'18, June 10-15, 2018, Houston, TX, USA

Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes

Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta
Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice
Tengiz Kharatishvilli, Xiaofeng Bao
Amazon Web Services

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 5: Percona TPC-C Variant (tpmC)

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

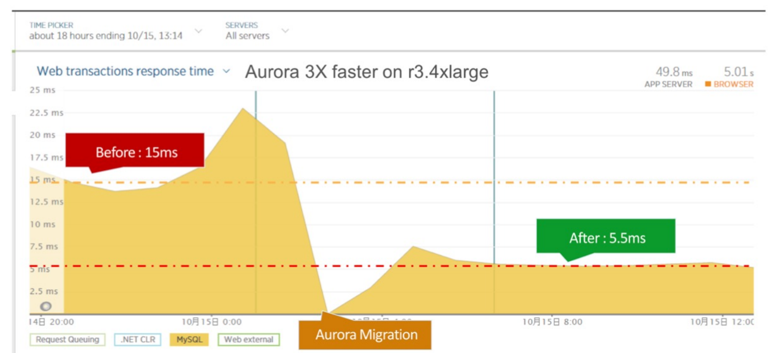


Figure 8: Web application response time

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

A Scaly Cloud