Modern Transactions

6.830 Lecture 18

Tianyu Li

Today -- Whirlwind Tour through Modern Transactions

- Looking Back

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

- Looking Ahead

Today -- Whirlwind Tour through Modern Transactions

- Looking Back
- Multi-node
  - Bottleneck: 2-Phase Commit
  - Single-Site Execution
  - Deterministic Transactions
  - Epoch-based Coordination
- Looking Ahead

Recap - Transactions

Single-node

- Disk-based

- 2PL

- Write-ahead Logging + Fuzzy Checkpoints

Recap - Transactions

Multi-node

- 2 PC for multi-node transactions

- 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

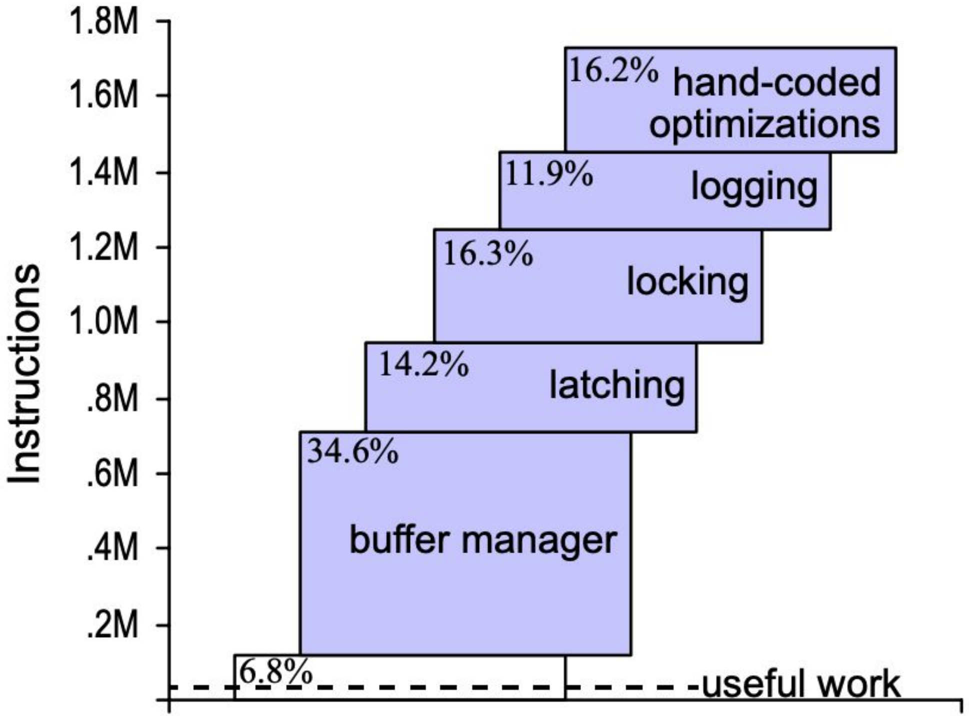
Now: Said 96 core machines has 384 GiB (!) of RAM

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

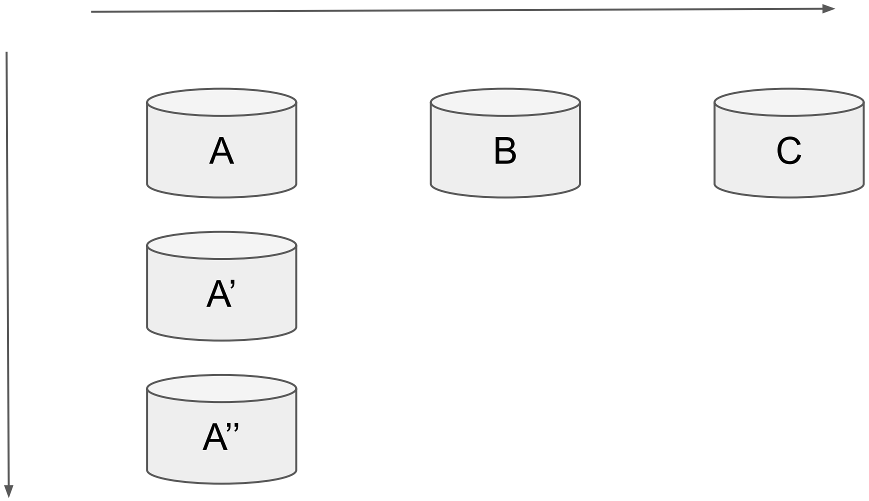
Today -- Whirlwind Tour through Modern Transactions

- Looking Back
- Multi-node
  - Bottleneck: 2-Phase Commit
  - Single-Site Execution
  - Deterministic Transactions
  - Epoch-based Coordination
- Looking Ahead

Recap: Scaling a Database

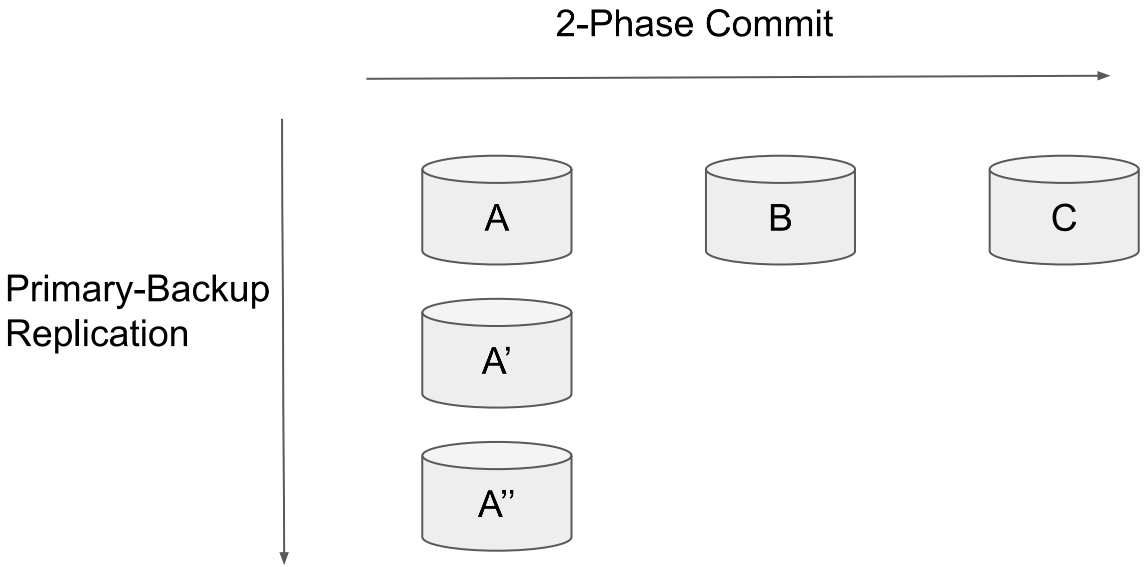
More shard/partitions --- more parallelism and throughput

More replicas --- higher availability



* Replicas usually also serve read requests

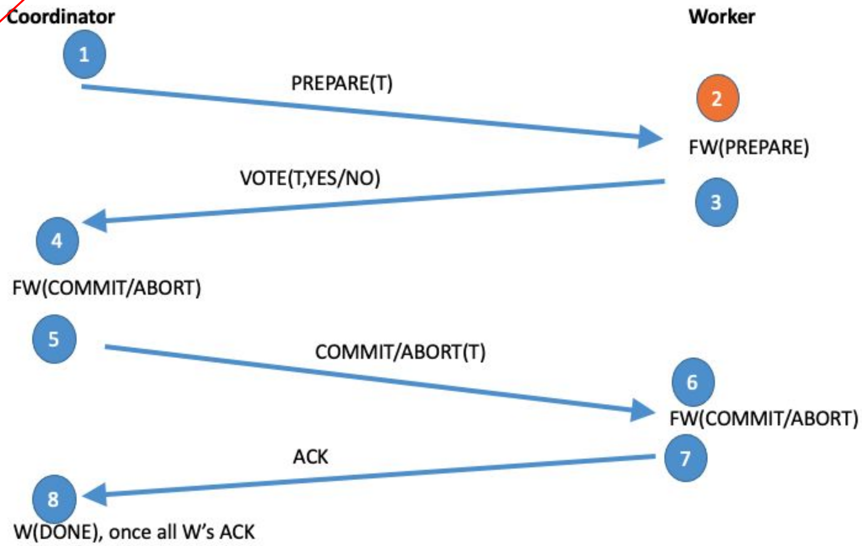
Recap: Scaling a Database



Recap: 2-Phase Commit

Commit Point

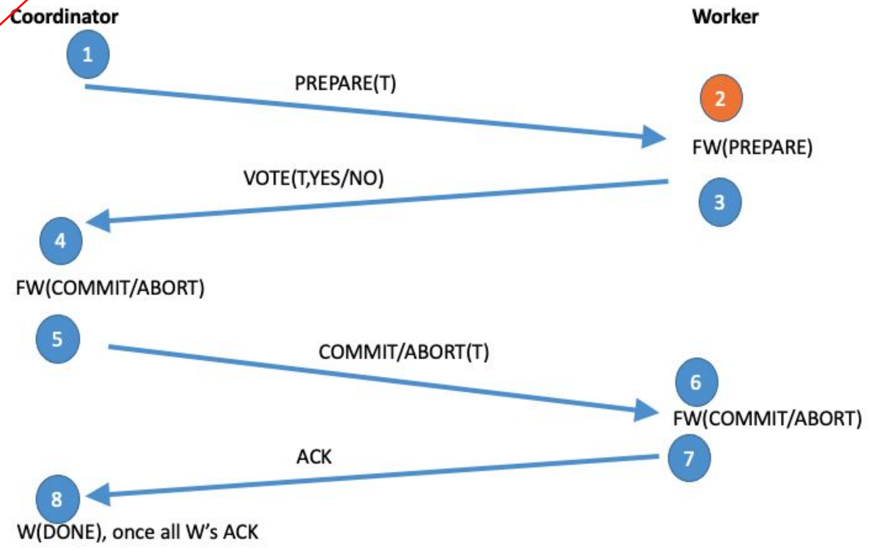
1. Log start of transaction
2. Execute transaction on worker nodes
3. PREPARE each worker
4. Log transaction commit if all OK
5. Commit each worker
6. Log Done



Recap: 2-Phase Commit

Commit Point

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



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

replicas	latency (ms)			throughput (Kops/sec)		
	write	read-only transaction	snapshot read	write	read-only transaction	snapshot read
1D	9.4±.6	—	—	4.0±.3	—	—
1	14.4±1.0	1.4±.1	1.3±.1	4.1±.05	10.9±.4	13.5±.1
3	13.9±.6	1.3±.1	1.2±.1	2.2±.5	13.8±3.2	38.5±.3
5	14.4±.4	1.4±.05	1.3±.04	2.8±.3	25.3±5.2	50.0±1.1

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

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 simply too expensive.

Why does this limit throughput?

- R/W transactions in Spanner use locking
- 100 ms commit latency ≈ 10 ops per second per row
- Spanner is fine if you perform many reads.
- Not so much if you perform many writes.

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]
- More on this next lecture

Why bother with distributed transactions then?

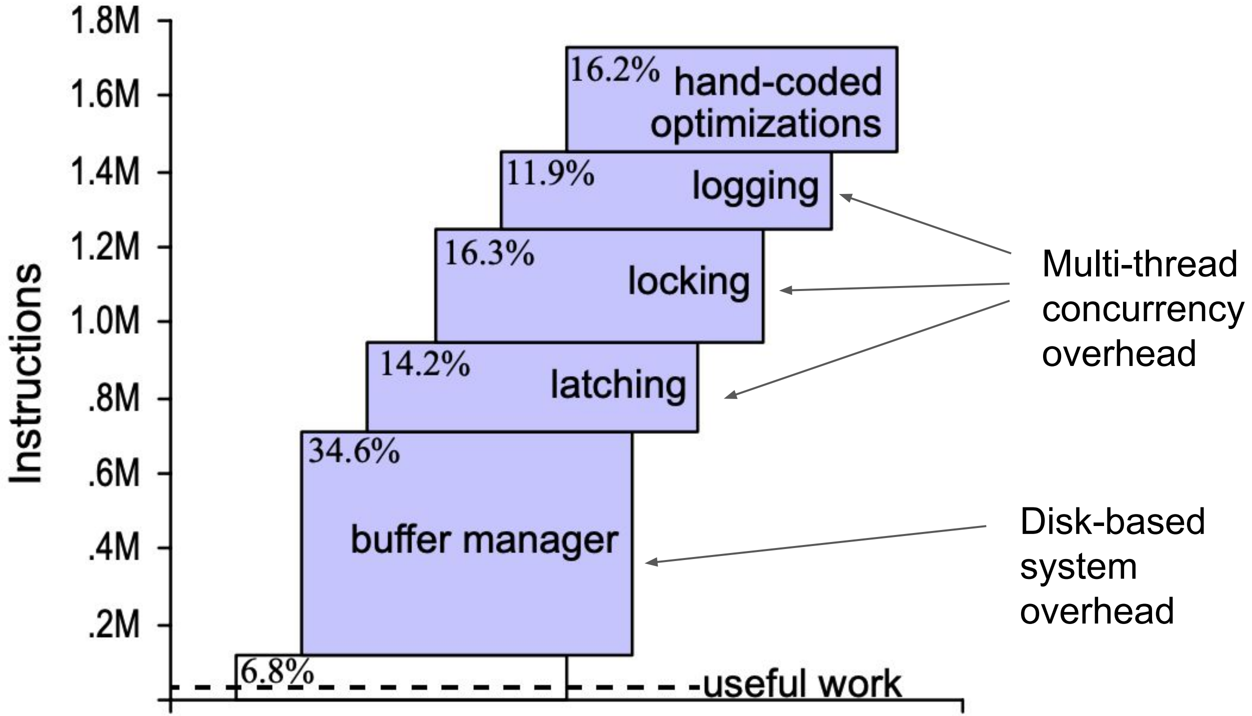
- Really powerful abstraction
- Extremely useful
- Impossibilities are mathematical. We are here to build systems.

Attempt 1: H-Store

- Large collaborative project from Brown, CMU, MIT, Yale, and Intel.
- Distributed & main-memory
- Commercialized as VoltDB

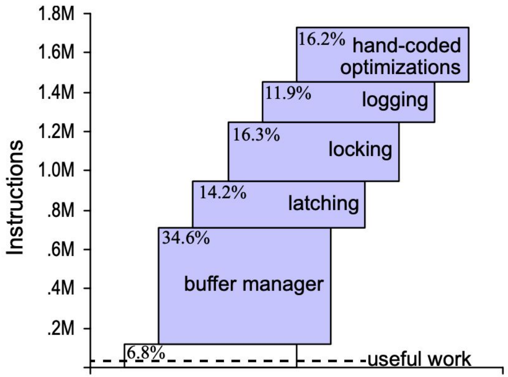
Key Idea

Recall:

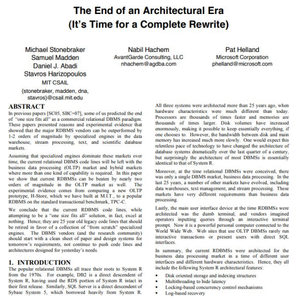


Key Idea

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



M. Stonebraker et. al.
The End of an Architectural Era (It’s Time for a Complete Rewrite). VLDB 2007.



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.

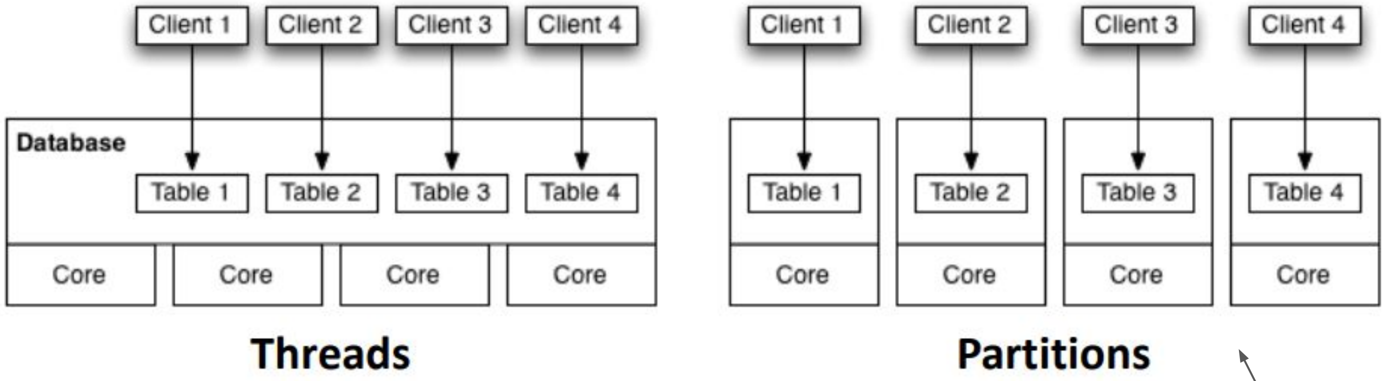
Moreover, at the time relational DBMSs were conceived, there was only a single DBMS market, business data processing. In the last 25 years, a number of other markets have evolved, including data warehouses, text management, and stream processing. These markets have very different requirements than business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

Partition vs. Threads



Synchronization
Overhead

No Concurrency
Control Required

Slide courtesy of Prof. Andy Pavlo

Is this reasonable?

Specialized for OLTP (Online Transaction Processing) workload

- Transactions finish quickly
- Transactions almost always point queries
- Transactions known beforehand

Example: TPC-C

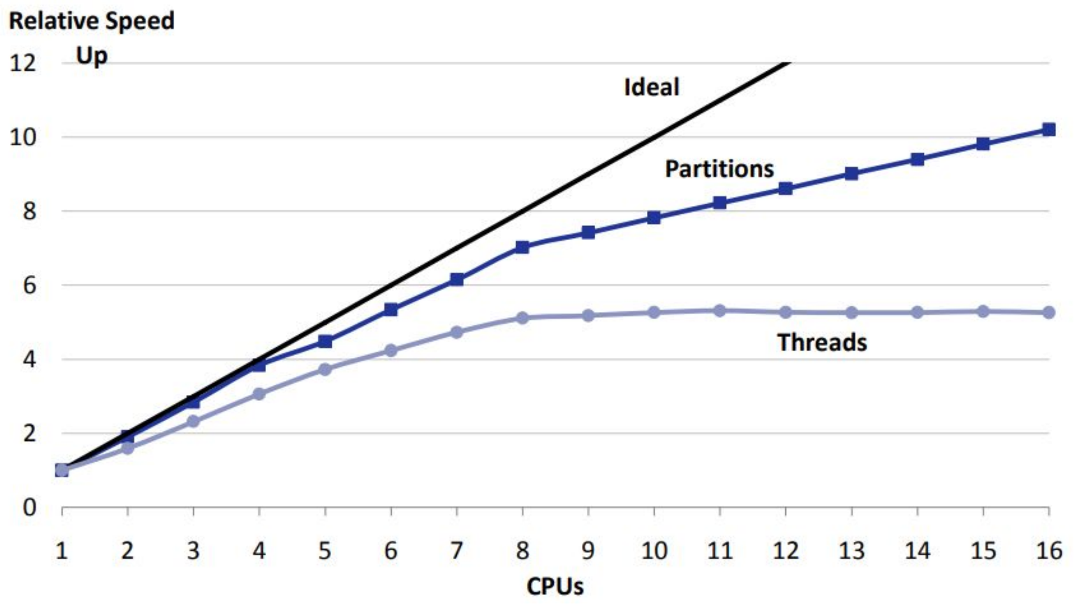
- Standardized benchmark used by everyone
- Models a warehouse order processing system
- Several types of transaction issued at random
- E.g. NewOrder Transaction:
  - Check item stock level
  - Create a new order
  - Update item stock level

* Technically, TPC-C is strictly specified. 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%

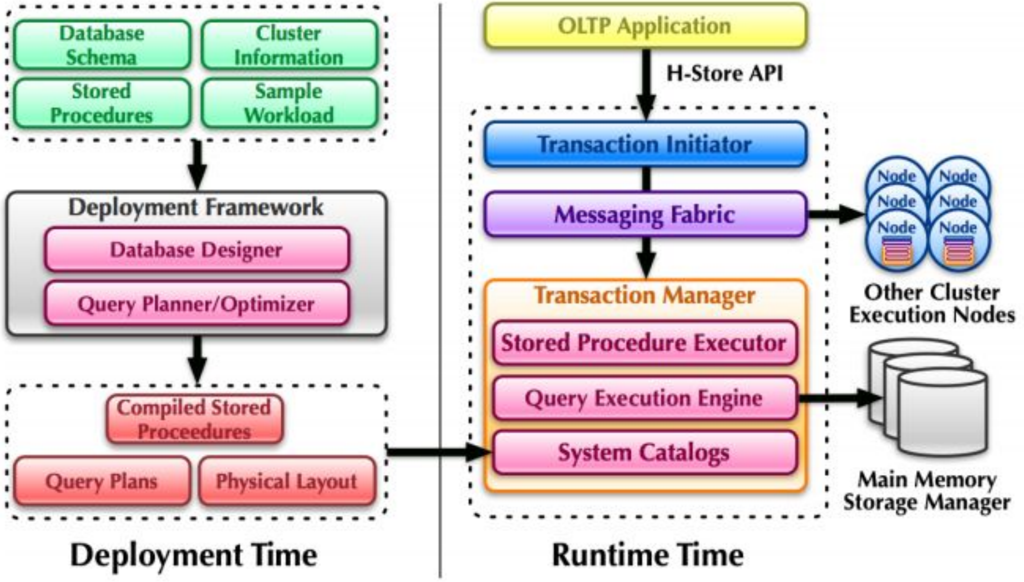
Partition vs. Threads: Performance



Slide courtesy of Prof. Andy Pavlo

H-Store Architecture

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



Kallman et. al., H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. VLDB 2008.

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 K txn/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

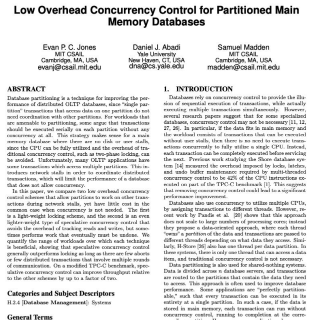
H-Store: Speculative Execution

- 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



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

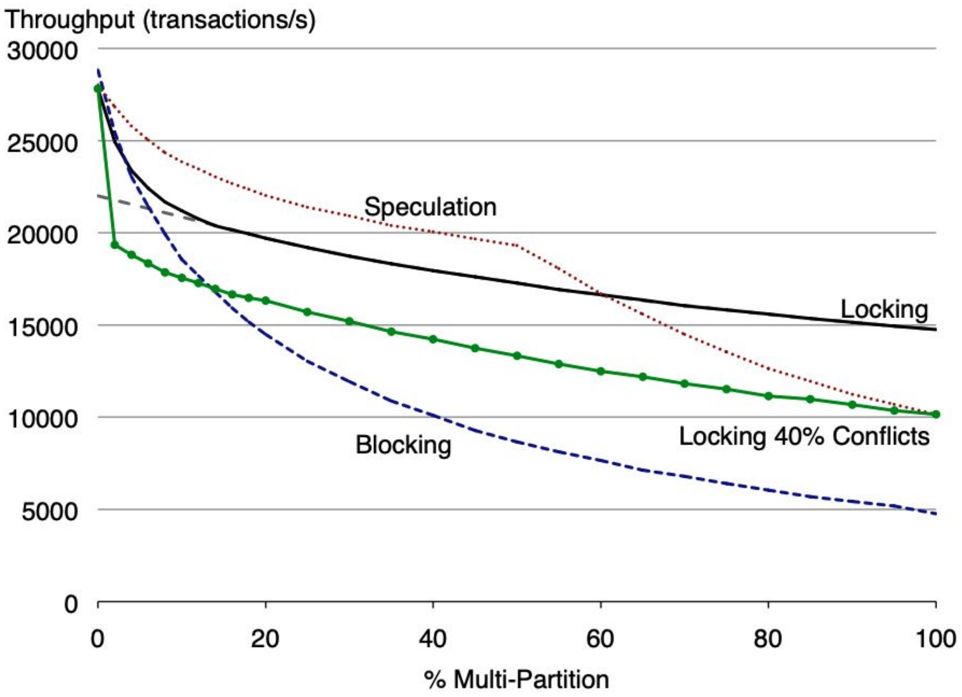
- Idea 2: Speculate whenever stalled
  - E.g., when a multi-partition transaction is reading a remote value
  - Locking required
  - Overhead lower than 2PL, since no latching
  - Still expensive

H-Store: Speculative Execution

- Synthetic benchmark --- single operation transactions

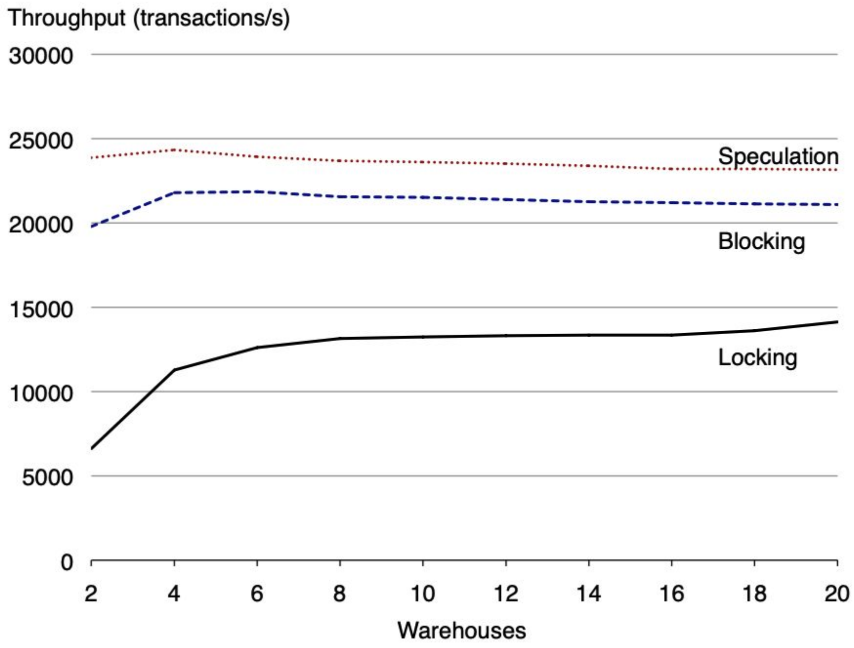
- Access half-keys on distributed

- Baseline no conflict



H-Store: Speculative Execution

- TPC-C
- Locking overhead increases with complex workload
- Speculation better

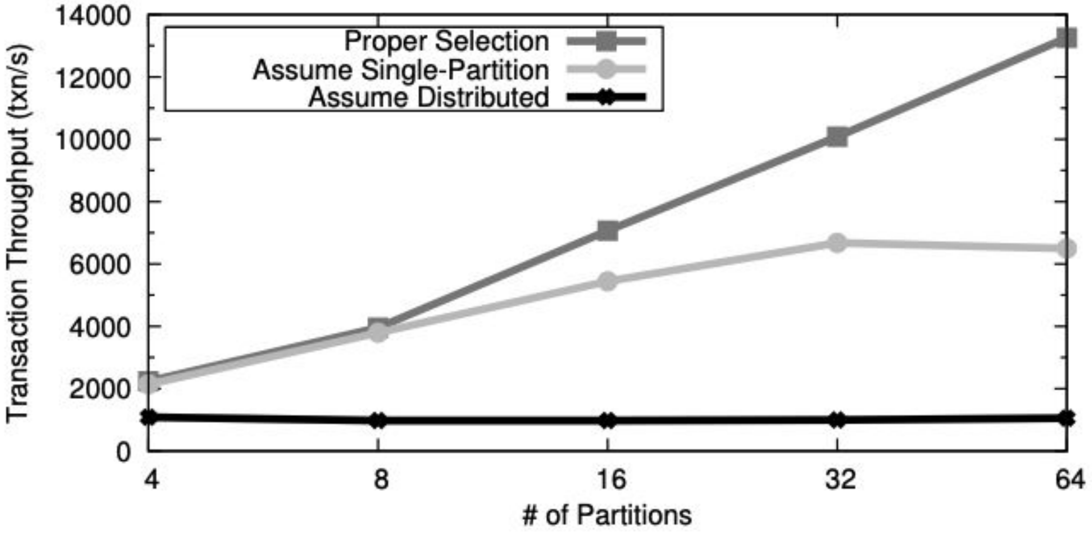


H-Store: Further Optimizations

- 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?

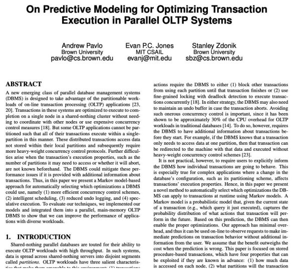
H-Store: Predictive Modeling

- 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.

Andrew Pavlo, Evan P.C. Jones, Stanley Zdonik. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. VLDB 2012.



On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems

Andrew Pavlo, Brown University, pavlo@cs.brown.edu
Evan P.C. Jones, MIT CSAIL, evanj@mit.edu
Stanley Zdonik, Brown University, sbz@cs.brown.edu

ABSTRACT

A new emerging class of parallel database management systems (DBMS) is designed to take advantage of the partitionable workloads of on-line transaction processing (OLTP) applications [23, 20]. Transactions in these systems are optimized to execute to completion on a single node in a shared-nothing cluster without needing to coordinate with other nodes or use expensive concurrency control measures [18]. But some OLTP applications cannot be partitioned such that all of their transactions execute within a single-partition in this manner. These distributed transactions access data not stored within their local partitions and subsequently require more heavy-weight concurrency control protocols. Further difficulties arise when the transaction's execution properties, such as the number of partitions it may need to access or whether it will abort, are not known beforehand. The DBMS could mitigate these performance issues if it is provided with additional information about transactions. Thus, in this paper we present a Markov model-based approach for automatically selecting which optimizations a DBMS could use, namely (1) more efficient concurrency control schemes, (2) intelligent scheduling, (3) reduced undo logging, and (4) speculative execution. To evaluate our techniques, we implemented our models and integrated them into a parallel, main-memory OLTP DBMS to show that we can improve the performance of applications with diverse workloads.

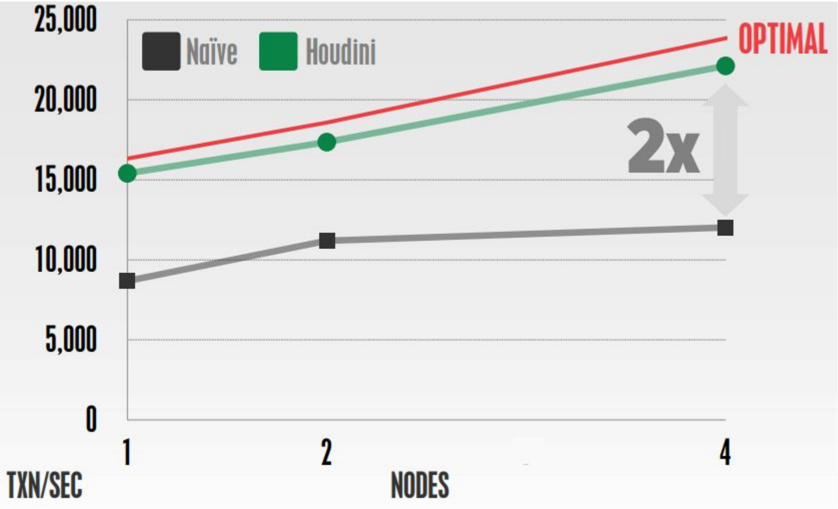
1. INTRODUCTION

Shared-nothing parallel databases are touted for their ability to execute OLTP workloads with high throughput. In such systems, data is spread across shared-nothing servers into disjoint segments called *partitions*. OLTP workloads have three salient characteris-

actions require the DBMS to either (1) block other transactions from using each partition until that transaction finishes or (2) use fine-grained locking with deadlock detection to execute transactions concurrently [18]. In either strategy, the DBMS may also need to maintain an undo buffer in case the transaction aborts. Avoiding such onerous concurrency control is important, since it has been shown to be approximately 30% of the CPU overhead for OLTP workloads in traditional databases [14]. To do so, however, requires the DBMS to have additional information about transactions before they start. For example, if the DBMS knows that a transaction only needs to access data at one partition, then that transaction can be redirected to the machine with that data and executed without heavy-weight concurrency control schemes [23].

It is not practical, however, to require users to explicitly inform the DBMS how individual transactions are going to behave. This is especially true for complex applications where a change in the database's configuration, such as its partitioning scheme, affects transactions' execution properties. Hence, in this paper we present a novel method to automatically select which optimizations the DBMS can apply to transactions at runtime using Markov models. A Markov model is a probabilistic model that, given the current state of a transaction (e.g., which query it just executed), captures the probability distribution of what actions that transaction will perform in the future. Based on this prediction, the DBMS can then enable the proper optimizations. Our approach has minimal overhead, and thus it can be used on-line to observe requests to make immediate predictions on transaction behavior without additional information from the user. We assume that the benefit outweighs the cost when the prediction is wrong. This paper is focused on stored procedure-based transactions, which have four properties that can be exploited if they are known in advance: (1) how much data is accessed on each node, (2) what partitions will the transaction

H-Store: Predictive Modeling



Transaction Behavior Prediction

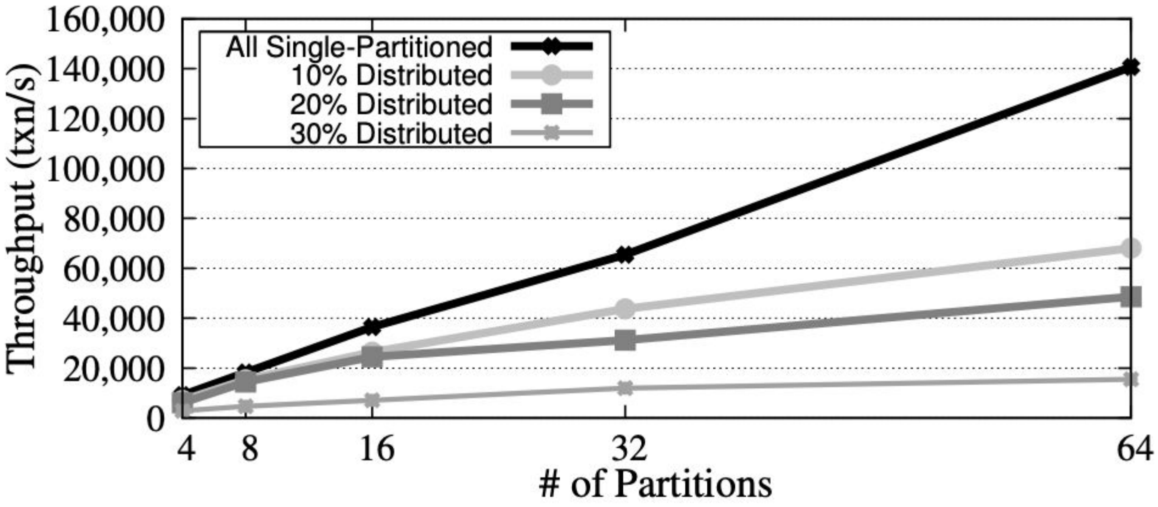
Slide courtesy of Prof. Andy Pavlo

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

Andrew Pavlo, Carlo Curino, Stan Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. SIGMOD 2012.

Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems

Andrew Pavlo
Brown University
pavlo@cs.brown.edu

Carlo Curino
Yahoo! Research
krl@yahoo-inc.com

Stan Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

The advent of affordable, shared-nothing computing systems portends a new class of parallel database management systems (DBMS) for on-line transaction processing (OLTP) applications that scale without sacrificing ACID guarantees [7, 9]. The performance of these DBMSs is predicated on the existence of an optimal database design that is tailored for the unique characteristics of OLTP workloads [43]. Deriving such designs for modern DBMSs is difficult, especially for *enterprise-class* OLTP systems, since they impose extra challenges: the use of stored procedures, the need for load balancing in the presence of time-varying skew, complex schemas, and deployments with larger number of partitions.

To this purpose, we present a novel approach to automatically partitioning databases for enterprise-class OLTP systems that significantly extends the state of the art by: (1) minimizing the number distributed transactions, while concurrently mitigating the effects of temporal skew in both the data distribution and accesses, (2) extending the design space to include replicated secondary indexes, (4) organically handling stored procedure routing, and (3) scaling of schema complexity, data size, and number of partitions. This effort builds on two key technical contributions: an analytical cost model that can be used to quickly estimate the relative coordination cost and skew for a given workload and a candidate database design, and an informed exploration of the huge solution space based on large neighborhood search. To evaluate our methods, we integrated our database design tool with a high-performance parallel, main memory DBMS and compared our methods against both popular heuristics and a state-of-the-art research prototype [17]. Using a diverse set of benchmarks, we show that our approach improves throughput by up to a factor of 16× over these other approaches.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design

Keywords

OLTP, Parallel, Shared-Nothing, H-Store, KB, Stored Procedures

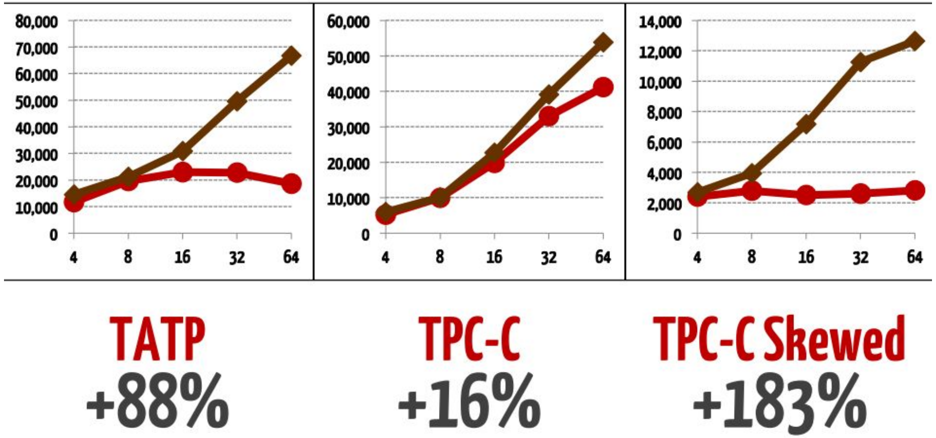
this problem employed by many Web-based companies is to *partition* the data and workload across a large number of commodity, shared-nothing servers using a cost-effective, parallel DBMS. Many of these companies have adopted various new DBMSs, colloquially referred to as *NoSQL* systems, that give up transactional ACID guarantees in favor of availability and scalability [9]. This approach is desirable if the consistency requirements of the data are "soft" (e.g., status updates on a social networking site that do not need to be immediately propagated throughout the application).

OLTP systems, especially enterprise OLTP systems that handle high-profile data (e.g., financial and order processing systems), also need to be scalable but cannot give up strong transactional and consistency requirements [27]. The only option previously available for these organizations was to purchase more powerful single-node machines or develop custom middleware that distributes queries over traditional DBMS nodes [41]. Both approaches are prohibitively expensive and thus are not an option for many.

As an alternative to NoSQL and custom deployments, a new class of parallel DBMSs, called *NewSQL* [7], is emerging. These systems are designed to take advantage of the partitionability of OLTP workloads to achieve scalability without sacrificing ACID guarantees [9, 43]. The OLTP workloads targeted by these NewSQL systems are characterized as having a large number of transactions that (1) are short-lived (i.e., no user stalls), (2) touch a small subset of data using index look-ups (i.e., no full table scans or large distributed joins), and (3) are repetitive (i.e., typically executed as pre-defined transaction templates or stored procedures [43, 42].)

The scalability of OLTP applications on many of these newer DBMSs depends on the existence of an optimal *database design*. Such a design defines how an application's data and workload is partitioned or replicated across nodes in a cluster, and how queries and transactions are routed to nodes. This in turn determines the number of transactions that access data stored on each node and how skewed the load is across the cluster. Optimizing these two factors is critical to scaling complex systems: our experimental evidence shows that a growing fraction of distributed transactions and load skew can degrade performance by over a factor 10× Hence,

H-Store: Partitioning



Optimized Partition

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

Attempt 2: Calvin / Aria

- Why is H-Store faster without concurrency?

- No non-determinism from threading
  - Limits cross thread/node coordination need

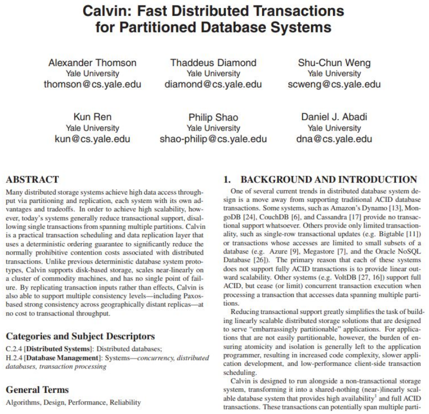
- Can the same idea be applied to truly distributed transactions?

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

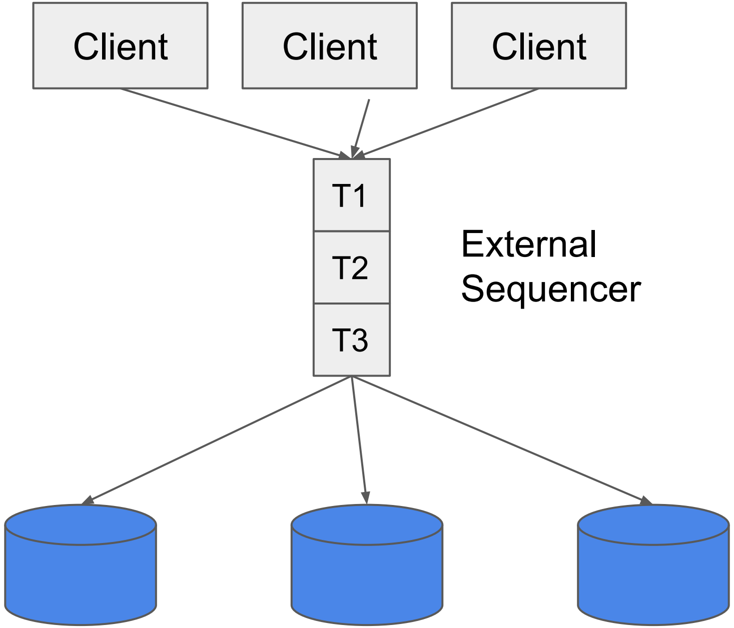
1. BACKGROUND AND INTRODUCTION

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



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

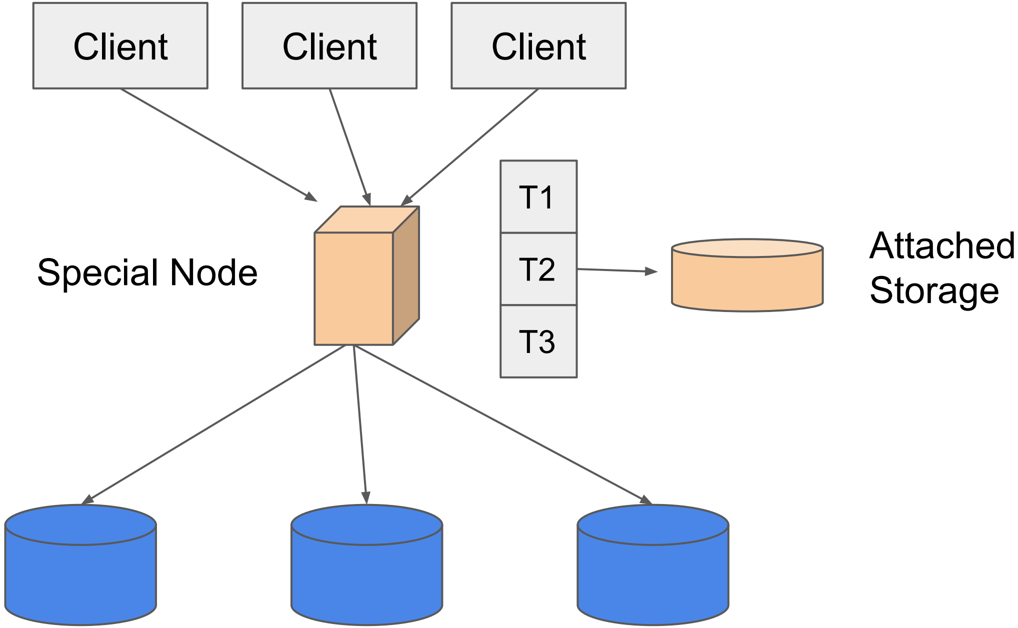
Practical Considerations

- 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

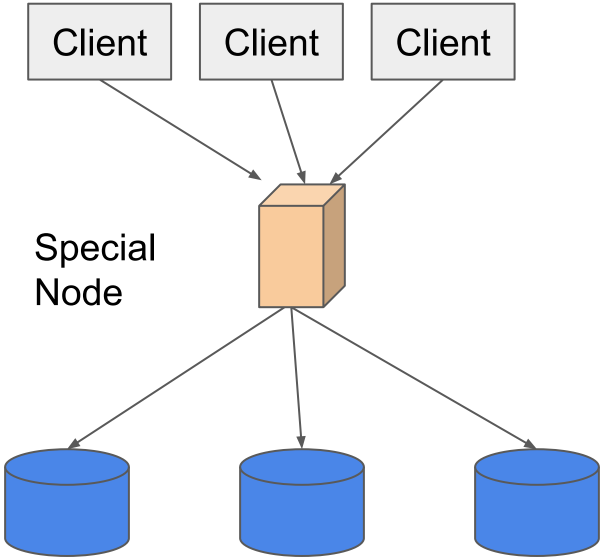
Practical Considerations

- 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

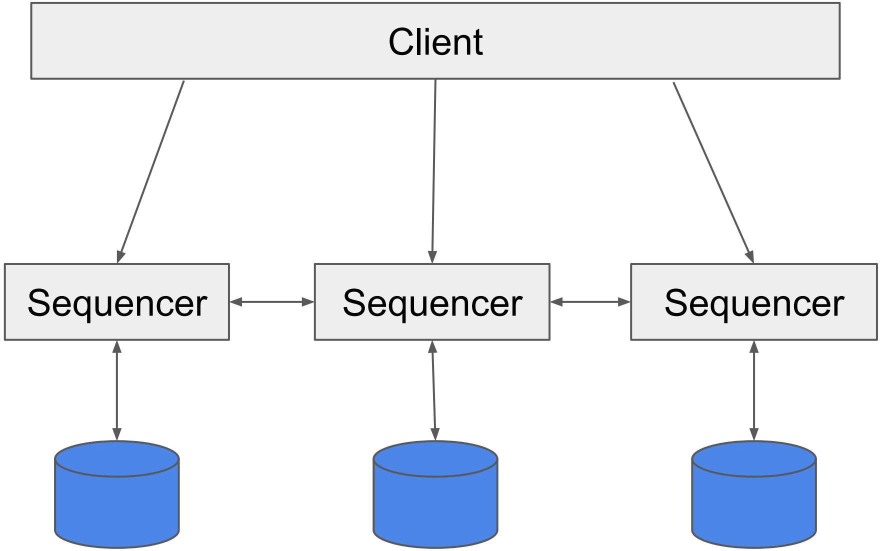


Sequencer: Initial Attempt



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

Distributed Sequencer



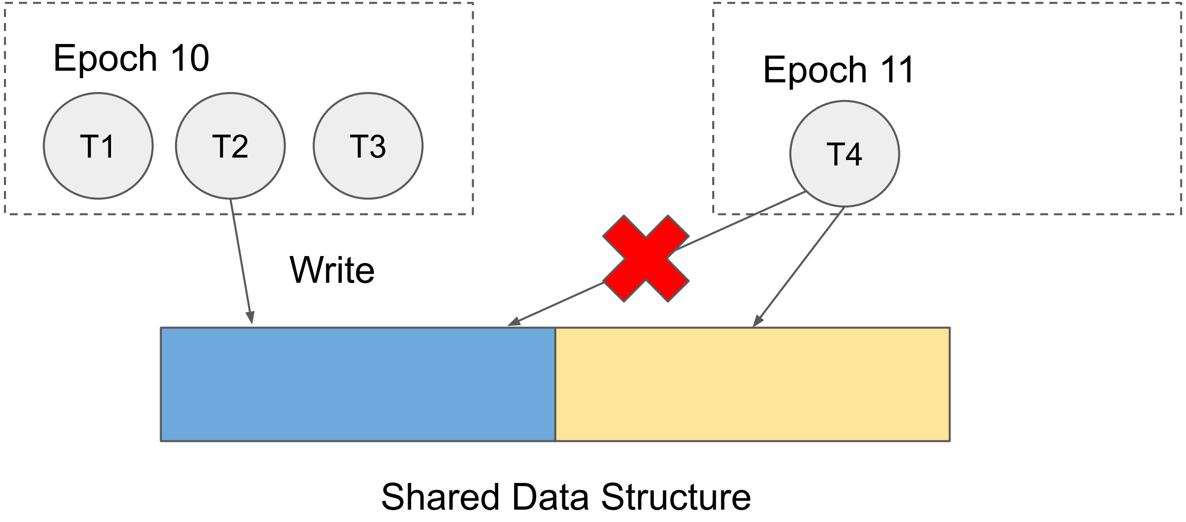
- Don’t synchronize for every request
- Each sequencer collects a batch of requests
- Periodically replicate / persist and exchange batches

Key Idea: Epochs

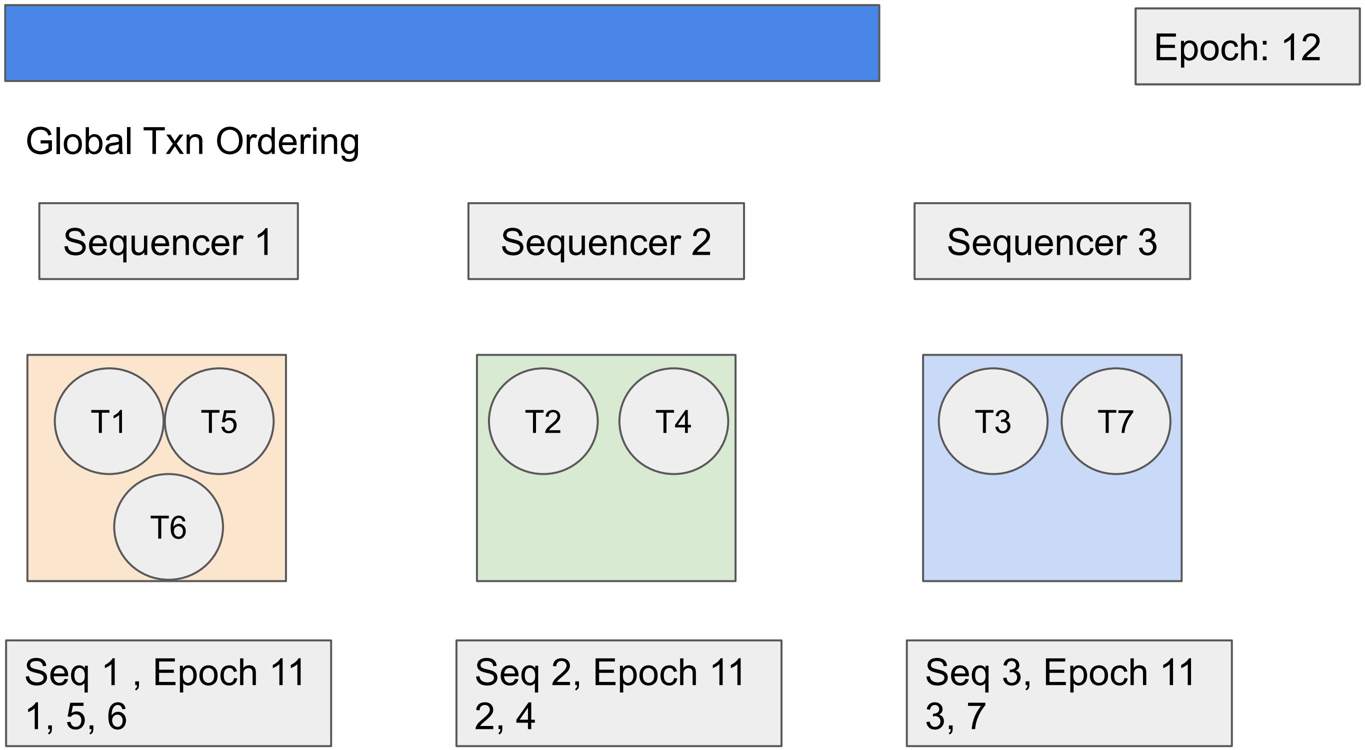
- Recall: Coordination off critical path = win
- Idea: Synchronize loosely at set intervals (i.e., epochs)
- Wait for next epoch if uncertain
- Common idea in concurrent programming. Also used in single-node systems (e.g., Silo, Bw-Tree)

Example: Epoch

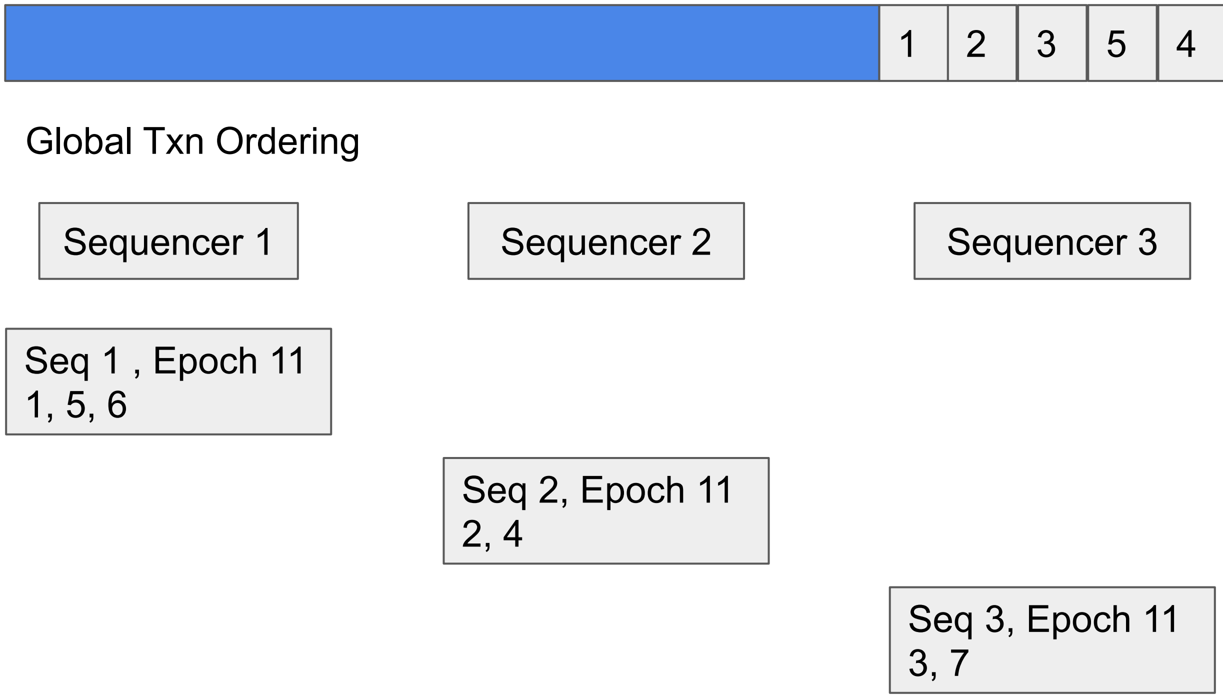
- Concurrently updating a buffer in-place
- Periodically “freeze” a log chunk to flush to disk



Example: Epoch-based Sequencer



Example: Epoch-based Sequencer



- 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

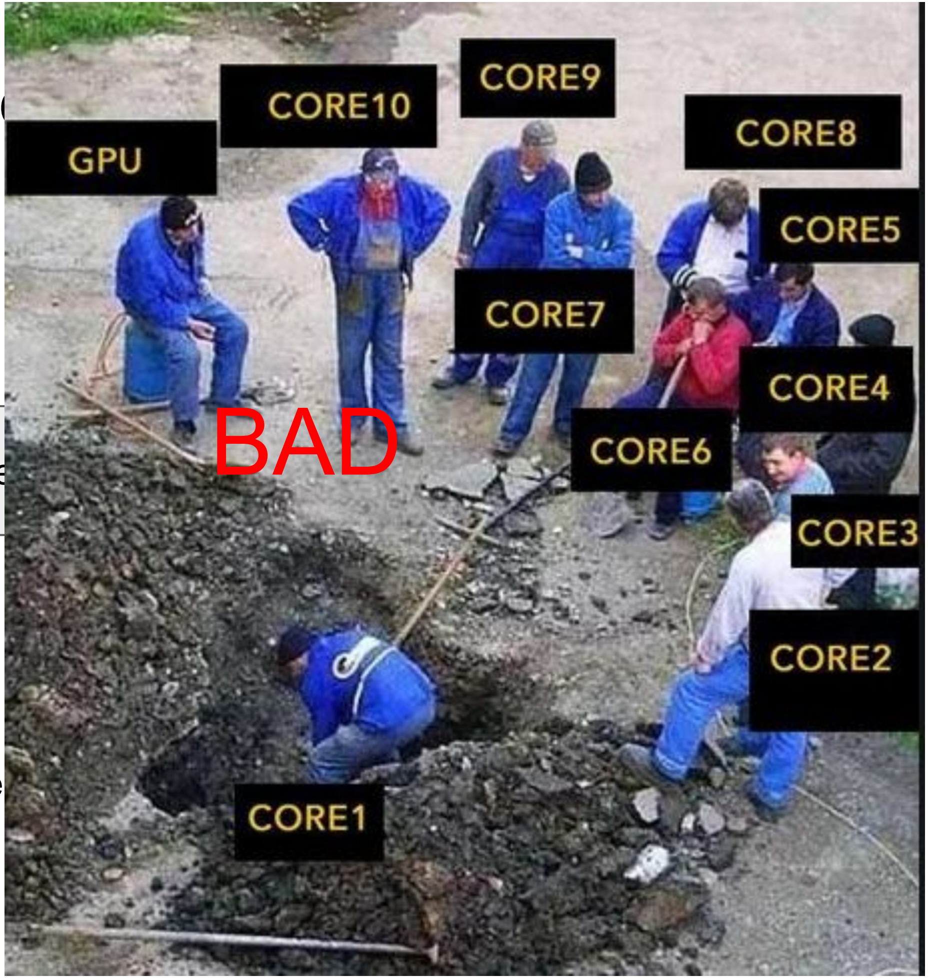
Practical Considerations

- 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 C

Consider Schedule:

Read A, Write B	Re



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

Scheduler: Deterministic Concurrency Control

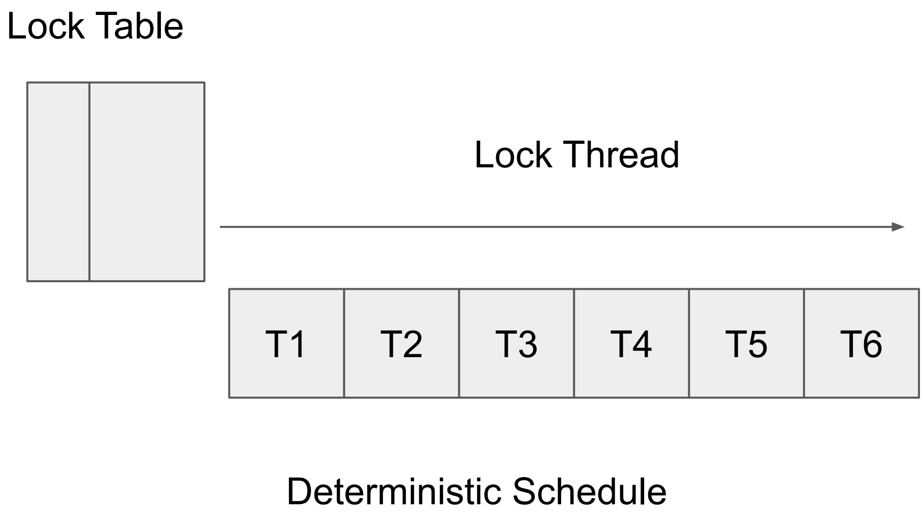
- Need to allow for concurrent execution

- 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

Logging and Checkpoints

- Transactions still need to be durable

- Because deterministic: no redo logging required
  - Log inputs instead

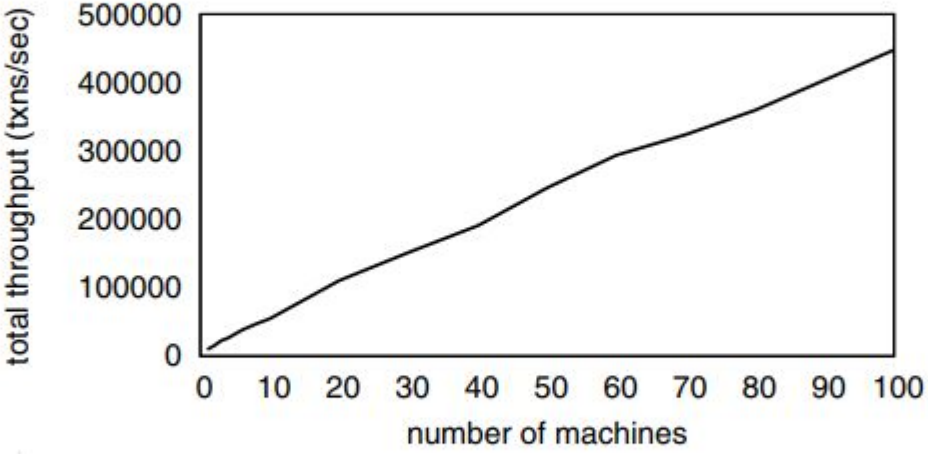
- Checkpointing would be nice though
  - Otherwise, on failure, have to replay from the beginning of time

Logging and Checkpoints

- Thought 1: Zig-Zag algorithm
  - Freeze a replica and write synchronously to disk
  - Requires taking down one replica, not always a good idea

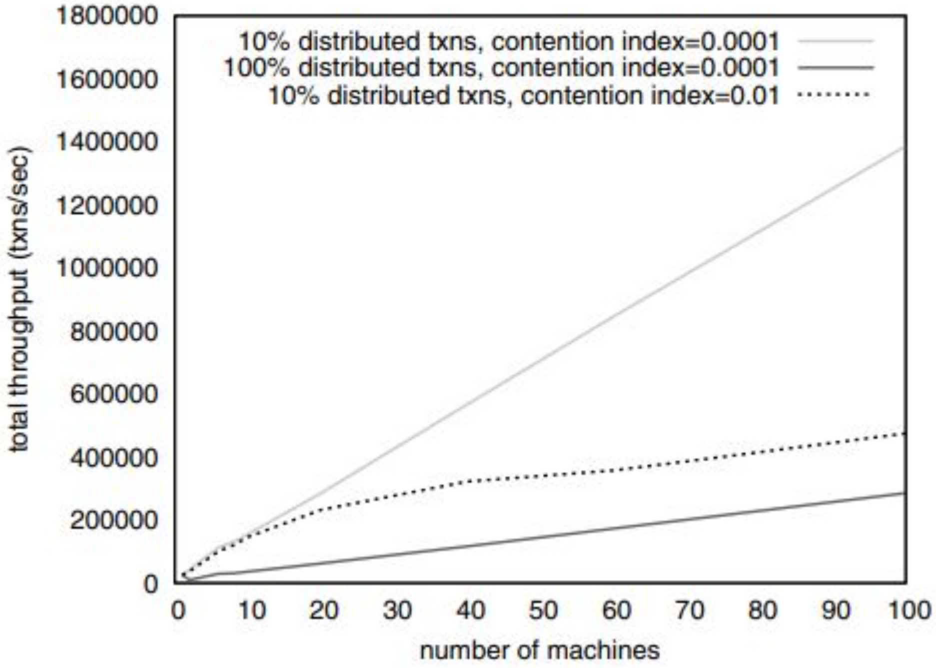
- Thought 2: Multi-versioning
  - Determine a point of consistency in the global schedule
  - Storage implementation multi-versions around that point until checkpoint is complete

Calvin: Results

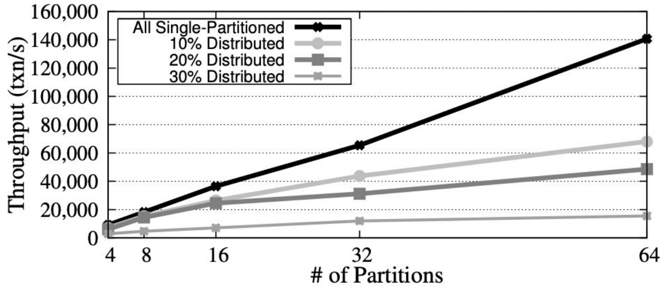


- TPC-C (100% New Order)

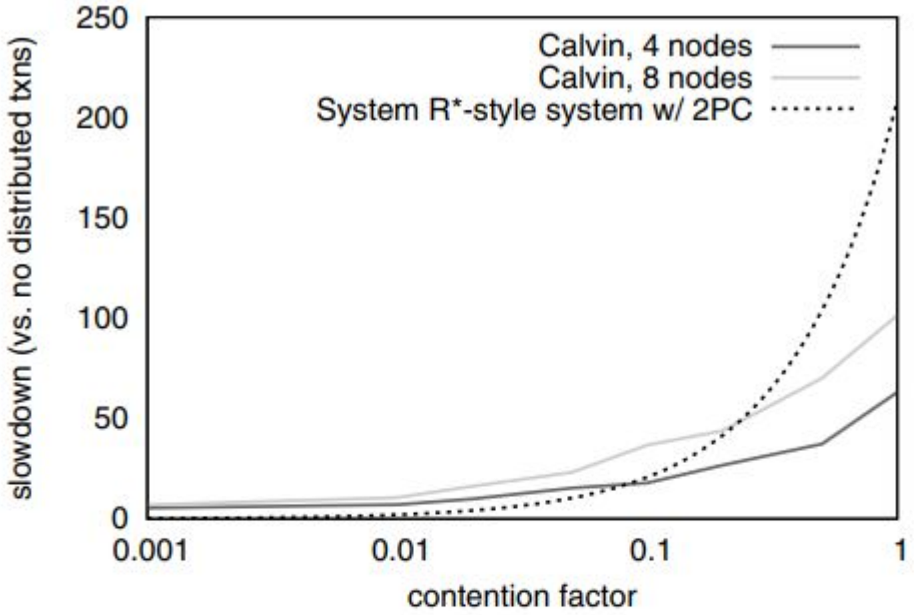
Calvin: Results



- 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

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

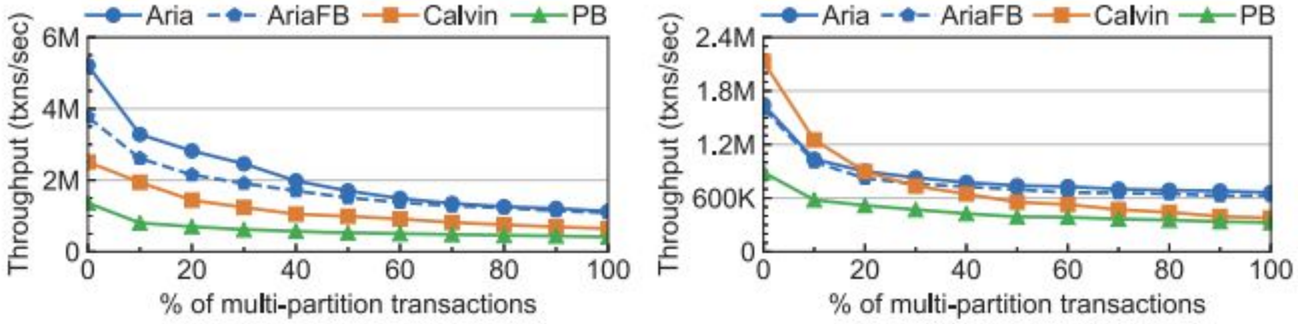
1. INTRODUCTION

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

Aria: Results



(a) YCSB - batch size: 10K * 8 = 80K (b) TPC-C - batch size: 500 * 8 = 4K

Takeaways

- Determinism can be a good thing

- Distributed coordination off the critical path = win

Attempt 3: COCO

- H-Store leveraged workload patterns

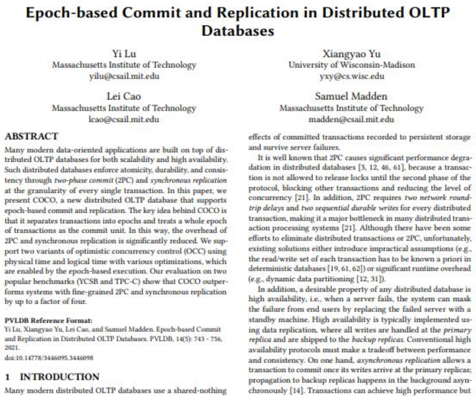
- Calvin/Aria introduces sequencer and increases latency

- Can we attack the problem of 2PC head-on?
  - Surprisingly, yes. Well, sometimes.
  - Area of (very) recent work

COCO

- 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. (to appear) VLDB 2021



Epoch-based Commit and Replication in Distributed OLTP Databases

Yi Lu
Massachusetts Institute of Technology
yilu@csail.mit.edu

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

Lei Cao
Massachusetts Institute of Technology
lcao@csail.mit.edu

Samuel Madden
Massachusetts Institute of Technology
madden@csail.mit.edu

ABSTRACT

Many modern data-oriented applications are built on top of distributed OLTP databases for both scalability and high availability. Such distributed databases enforce atomicity, durability, and consistency through *two-phase commit* (2PC) and *synchronous replication* at the granularity of every single transaction. In this paper, we present COCO, a new distributed OLTP database that supports epoch-based commit and replication. The key idea behind COCO is that it separates transactions into epochs and treats a whole epoch of transactions as the commit unit. In this way, the overhead of 2PC and synchronous replication is significantly reduced. We support two variants of optimistic concurrency control (OCC) using physical time and logical time with various optimizations, which are enabled by the epoch-based execution. Our evaluation on two popular benchmarks (YCSB and TPC-C) show that COCO outperforms systems with fine-grained 2PC and synchronous replication by up to a factor of four.

PVLDB Reference Format:
Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Epoch-based Commit and Replication in Distributed OLTP Databases. PVLDB, 14(5): 743 - 756, 2021.
doi:10.14778/3446095.3446098

1 INTRODUCTION

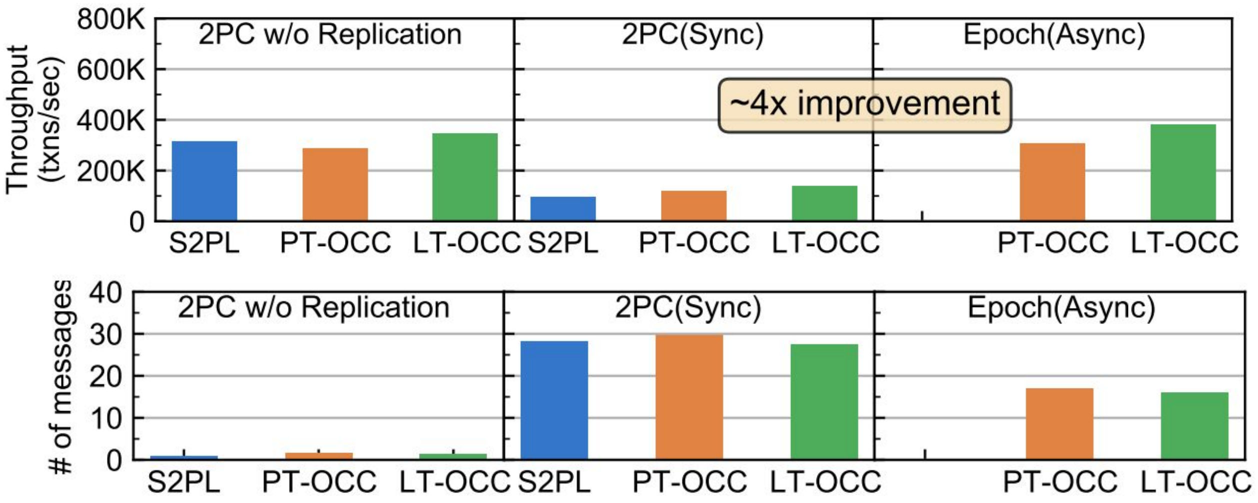
Many modern distributed OLTP databases use a shared-nothing

effects of committed transactions recorded to persistent storage and survive server failures.

It is well known that 2PC causes significant performance degradation in distributed databases [3, 12, 46, 61], because a transaction is not allowed to release locks until the second phase of the protocol, blocking other transactions and reducing the level of concurrency [21]. In addition, 2PC requires *two network round-trip delays* and *two sequential durable writes* for every distributed transaction, making it a major bottleneck in many distributed transaction processing systems [21]. Although there have been some efforts to eliminate distributed transactions or 2PC, unfortunately, existing solutions either introduce impractical assumptions (e.g., the read/write set of each transaction has to be known a priori in deterministic databases [19, 61, 62]) or significant runtime overhead (e.g., dynamic data partitioning [12, 31]).

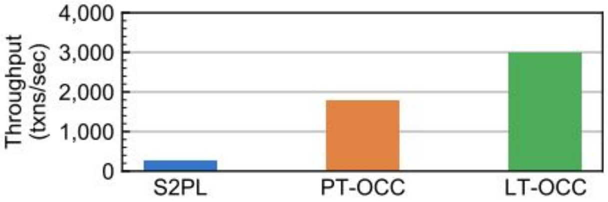
In addition, a desirable property of any distributed database is high availability, i.e., when a server fails, the system can mask the failure from end users by replacing the failed server with a standby machine. High availability is typically implemented using data replication, where all writes are handled at the *primary replica* and are shipped to the *backup replicas*. Conventional high availability protocols must make a tradeoff between performance and consistency. On one hand, *asynchronous replication* allows a transaction to commit once its writes arrive at the primary replicas; propagation to backup replicas happens in the background asynchronously [14]. Transactions can achieve high performance but

COCO: Results

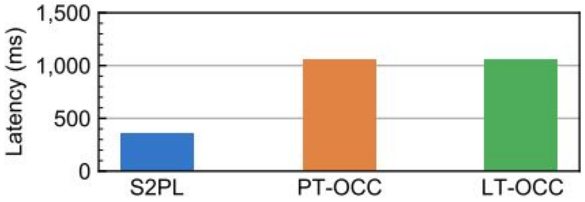


TPC-C Performance

COCO: Results over WAN



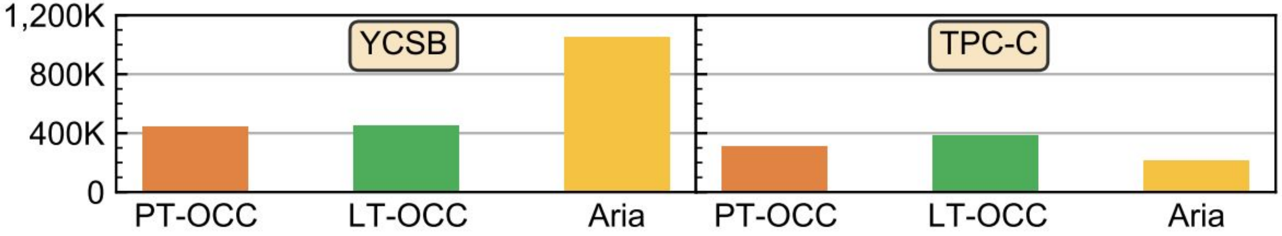
TPC-C Throughput



TPC-C P99 Latency

- Recall: Throughput vs. Latency trade-off

COCO: Comparison with Aria

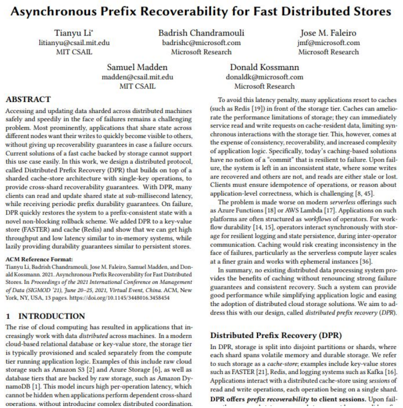


- COCO appears to do better when contention is higher

Can we do better?

- Avoid global epoch counter and track individual shard versions
- Currently only works for non-transactional workload

Tianyu Li, Badrish Chandramouli, Jose M. Faleiro, Samuel Madden, Donald Kossmann. Asynchronous Prefix Recoverability for Fast Distributed Stores. (to appear) SIGMOD 2021



Asynchronous Prefix Recoverability for Fast Distributed Stores

Tianyu Li*
litianyu@csail.mit.edu
MIT CSAIL

Badrish Chandramouli
badrishc@microsoft.com
Microsoft Research

Jose M. Faleiro
jmf@microsoft.com
Microsoft Research

Samuel Madden
madden@csail.mit.edu
MIT CSAIL

Donald Kossmann
donaldk@microsoft.com
Microsoft Research

ABSTRACT

Accessing and updating data sharded across distributed machines safely and speedily in the face of failures remains a challenging problem. Most prominently, applications that share state across different nodes want their writes to quickly become visible to others, without giving up recoverability guarantees in case a failure occurs. Current solutions of a fast cache backed by storage cannot support this use case easily. In this work, we design a distributed protocol, called Distributed Prefix Recovery (DPR) that builds on top of a sharded cache-store architecture with single-key operations, to provide cross-shard recoverability guarantees. With DPR, many clients can read and update shared state at sub-millisecond latency, while receiving periodic prefix durability guarantees. On failure, DPR quickly restores the system to a prefix-consistent state with a novel non-blocking rollback scheme. We added DPR to a key-value store (FASTER) and cache (Redis) and show that we can get high throughput and low latency similar to in-memory systems, while lazily providing durability guarantees similar to persistent stores.

ACM Reference Format:
Tianyu Li, Badrish Chandramouli, Jose M. Faleiro, Samuel Madden, and Donald Kossmann. 2021. Asynchronous Prefix Recoverability for Fast Distributed Stores. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3458454

1 INTRODUCTION

The rise of cloud computing has resulted in applications that increasingly work with data *distributed* across machines. In a modern cloud-based relational database or key-value store, the storage tier is typically provisioned and scaled separately from the compute tier running application logic. Examples of this include raw cloud storage such as Amazon S3 [2] and Azure Storage [6], as well as database tiers that are backed by raw storage, such as Amazon DynamoDB [1]. This model incurs high per-operation latency, which cannot be hidden when applications perform dependent cross-shard operations, without introducing complex distributed coordination.

To avoid this latency penalty, many applications resort to caches (such as Redis [19]) in front of the storage tier. Caches can ameliorate the performance limitations of storage; they can immediately service read and write requests on cache-resident data, limiting synchronous interactions with the storage tier. This, however, comes at the expense of consistency, recoverability, and increased complexity of application logic. Specifically, today's caching-based solutions have no notion of a "commit" that is resilient to failure. Upon failure, the system is left in an inconsistent state, where some writes are recovered and others are not, and reads are either stale or lost. Clients must ensure idempotence of operations, or reason about application-level correctness, which is challenging [8, 45].

The problem is made worse on modern *serverless* offerings such as Azure Functions [18] or AWS Lambda [17]. Applications on such platforms are often structured as *workflows* of operators. For workflow durability [14, 15], operators interact synchronously with storage for resilient logging and state persistence, during inter-operator communication. Caching would risk creating inconsistency in the face of failures, particularly as the serverless compute layer scales at a finer grain and works with ephemeral instances [36].

In summary, no existing distributed data processing system provides the benefits of caching without renouncing strong failure guarantees and consistent recovery. Such a system can provide good performance while simplifying application logic and easing the adoption of distributed cloud storage solutions. We aim to address this with our design, called *distributed prefix recovery* (*DPR*).

Distributed Prefix Recovery (DPR)

In DPR, storage is split into disjoint partitions or shards, where each shard spans volatile memory and durable storage. We refer to such storage as a *cache-store*; examples include key-value stores such as FASTER [21], Redis, and logging systems such as Kafka [16]. Applications interact with a distributed cache-store using *sessions* of read and write operations, each operation being on a single shard. **DPR offers *prefix recoverability* to client sessions.** Upon fail-

Epochs: Takeaways

- Surprisingly effective in alleviating bottleneck

- Throughput vs. Latency Trade-off

Today -- Whirlwind Tour through Modern Transactions

- Looking Back
- Multi-node
  - Bottleneck: 2-Phase Commit
  - Single-Site Execution
  - Deterministic Transactions
  - Epoch-based Coordination
- Looking Ahead

Chasing the numbers

- Exciting.*
- Transactions 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

* For some. More on this later.

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?
  - 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
  - Hard to rock the boat on an established field

* next lecture

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 (Microsoft’s FaRM, Write-behind Logging)
- Cloud looming over the playing field
  - DBMSs fundamentally designed for shared-nothing
  - Cloud services change that
  - Open question on how to build performant, cheap, elastic cloud-native DBMS