The Dissertation Committee for Syed Akbar Mehdi
certifies that this is the approved version of the following dissertation:

Scalability through Asynchrony in Transactional Storage Systems

Committee:

Lorenzo Alvisi, Co-supervisor

Simon Peter, Co-supervisor

Calvin Lin, Co-supervisor

Christopher J Rossbach

Eddie Kohler

Immanuel Trummer

Scalability through Asynchrony in Transactional Storage Systems

by

Syed Akbar Mehdi, B.E., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2022

Dedicated to my family: Urooj, Ammar, Anwar, Jabeen and Ali

Acknowledgments

I am grateful to my amazing advisors: Lorenzo Alvisi and Simon Peter. Their guidance, mentorship and drive for excellence have shaped me as a researcher. My introduction to Lorenzo was through his awesome distributed systems course at UT Austin. I shared his enthusiasm for building systems rooted in strong theoretical frameworks and deep insights, which led to us working together. I have learned a lot from Lorenzo over the years, including how to ask the right questions and how to abstract away the details to focus on the fundamentals of a problem. Lorenzo's guidance on the Occult paper was invaluable in helping me understand the broader implications of my own idea. Simon advised me during the latter half of my PhD, together with Lorenzo. Simon's help with ScaleDB was essential. I was always impressed by the ease with which he was able to guide me about low-level system details, as well as abstract thinking about system design. I learned a lot about systems building and scientific writing from Simon that will remain with me as an engineer and researcher.

I would like to thank my committee members: Chris Rossbach, Eddie Kohler, Calvin Lin and Immanuel Trummer, for their valuable suggestions and feedback that helped improve this dissertation.

I would like to thank my co-authors from the Occult and ScaleDB papers: Cody Littley, Natacha Crooks, Wyatt Lloyd, Nathan Bronson and Deukyeon Hwang. Nathan Bronson was a great mentor during internships at Facebook and interaction with him, during that time, led to the Occult paper. I was also fortunate to have friends in UT CS including: Youngjin Kwon, Trinabh Gupta, Tyler Hunt, Josh Berlin, Lara Schmidt and Ashay Rane.

Outside of graduate school, I shared a lot of good memories with a close circle of friends in Austin: Rashid Kaleem, Aater Suleman, Ghufran Baig, Khubaib, Muqeet Ali and Danish Irfan. I also want to mention my friends from Stanford, who were great company whenever I visited the Bay Area for internships: Wajahat Qadeer, Rehan Hameed, Haider Razvi and Tahir Azim. Junaid Khalid has been a good friend from our time together at NUST and Berkeley, and I am glad we have stayed in touch over the years.

I am grateful to my family for their unconditional love and support. My parents: Anwar and Jabeen, have been a constant source of love and encouragement throughout my life and especially during my PhD. I am fortunate to have a brother like Ali who has always been my best friend. My son, Ammar, was born during my PhD and he has filled our lives with joy ever since. Finally, to my wife, Urooj, who has stood by me through the highs and lows of my PhD journey, I am grateful to have you as my partner in life.

Scalability through Asynchrony in Transactional Storage Systems

Publication No. ____________________

Syed Akbar Mehdi, Ph.D.
The University of Texas at Austin, 2022

Supervisors: Lorenzo Alvisi
Simon Peter
Calvin Lin

Modern storage systems face daunting scalability challenges. The amount of data stored worldwide is doubling every two years. Compounding this problem are growing demands for these storage systems to offer strong correctness guarantees (such as consistency and transactional isolation): these guarantees require a degree of coordination that negatively affects scalability. Prior work has shown, that avoiding coordination is the key to scalability for a variety of systems.

This dissertation explores scalability problems due to coordination that is an artifact of the *mechanisms* used to implement a system rather than a fundamental requirement of the system's correctness guarantees. We call this phenomenon *mechanism coordination.* We explore how to build scalable storage systems that minimize mechanism coordination while continuing to provide stronger consistency guarantees to their clients.

We develop the insight that assuming nothing about how clients access these systems leads to synchronous implementations, which in-turn leads to mechanism coordination and scalability bottlenecks. By letting the design of these systems be informed by how clients are going to access them in the overwhelmingly common case, it is possible to derive asynchronous implementations that minimize mechanism coordination, enabling them to scale.

We demonstrate the broad applicability of this insight by building asynchronous client-driven solutions to two different scalability problems in two different classes of storage systems. The first part of the dissertation solves the problem of "slowdown cascades" in large-scale geo-replicated and distributed storage systems that provide causal consistency to their clients. The second part of the dissertation solves the problem of CPU contention on range indexes in multi-core in-memory databases that provide serializable isolation to their clients.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern storage systems face daunting scalability challenges. The amount of data stored worldwide is roughly doubling every two years [8, 25]. By 2025, 49% of this data will be stored in the public cloud [8] and nearly 30% will be consumed in real-time. Already, the largest data stores operate at an immense scale, serving on the order of billions of reads per second [144] and tens of millions of writes per second [6, 27, 144].

These scalability challenges are accompanied by a growing trend towards providing stronger correctness guarantees by storage systems [24, 49, 69, 74, 144]. By correctness guarantees, we mean either *consistency* or transactional *isolation*. These define a contract with the clients of the system, "specifying the set of behaviors that clients can expect to observe" [76] in the face of replication of the system's state (consistency) or concurrent access to

Section 1.2.1 of this chapter is derived from a prior publication describing the Occult [126] system. The author of this dissertation conceived, designed, implemented and evaluated the Occult system, and led the writing of the prior publication: S. A. Mehdi, et al. 2017. "I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades". In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, USA, 453–468. The introduction to slowdown cascades is derived from a prior publication co-authored by this dissertation's author, which discusses the Occult system: L. Alvisi et al. "Writes: the dirty secret of causal consistency." IEEE Data Eng. Bull. 40 (2017): 15-25.

the system by multiple client transactions (isolation).

Stronger correctness guarantees are in tension with scalability because the former demand increased *coordination* – "the requirement that concurrently executing operations synchronously communicate or otherwise stall in order to complete" [51]. Avoiding coordination has been shown to be the key to scalability; whether it is in the context of databases [51], distributed systems [89, 92] or software for shared-memory multi-core machines [57, 70].

Given the specification of a correctness guarantee, some coordination is *necessary*. For instance, a database that provides *serializable* [39] isolation must behave as if concurrent transactions execute in some serial order. This requires coordination [78]; for example, to ensure that, if two concurrent transactions attempt to update the same record, then one of them is aware of the other's update before performing its own. Similarly, a replicated data store that provides *linearizability* [94] to its clients cannot avoid coordination between its replicas [89].

Avoiding coordination has been the focus of prior work on building scalable and performant systems. Given the necessity of coordination when enforcing stronger guarantees, many systems [15, 63, 67, 71, 139] avoided coordination by choosing to provide weak guarantees instead. However, that choice came at the cost of increased anomalies for applications, resulting in the demand for stronger guarantees in recent years [24, 49, 69, 74, 144]. Another approach to coordination avoidance has stepped away from traditional correctness guarantees and defined necessary coordination in terms of application invariants specified through a new interface [51]. Yet, specifying invariants manually can be tedious [159] and error-prone, and the appeal of traditional correctness guarantees remains due to the simpler and clearer abstractions they provide.

The focus of this dissertation is to explore how to avoid *unnecessary* coordination in Online Transaction Processing (OLTP) storage systems. Assuming traditional correctness guarantees and typical storage system interfaces [11, 23], we focus on avoiding coordination that is an artifact of the mechanisms used to implement a system, rather than a fundamental requirement of the guarantees provided by the system. We refer to this unnecessary coordination as *mechanism coordination*. We identify and explain instances of mechanism coordination that cause scalability problems in two different classes of storage systems. In both instances, the design artifacts that we identify as causing mechanism coordination are canonical ways of implementing these systems!

The first problem – *slowdown cascades* – arises in sharded and geo-replicated key-value stores that provide causal consistency [41] to their clients. In prior approaches, a datacenter performs a write operation only after applying all writes that causally precede it. This design guarantees that reads never block, as all replicas are always in a causally consistent state, but, in the presence of slow or failed shards, may cause writes to be buffered for arbitrarily long periods of time. These failures, common in large-scale clusters, can lead to the *slowdown cascade* phenomenon, where a single slow or failed shard negatively impacts the entire system, delaying the visibility of updates across many shards and leading to growing queues of delayed updates [42, 126].

The second problem – *range index contention* – arises in multi-core in-memory relational databases that provide serializability [39] to their clients. Prior approaches synchronously update range indexes with the implicit pessimistic assumption that all committed transactional writes could potentially be read by the same transaction as part of a range scan immediately after the writing transaction(s) commit. This results in range indexes becoming a scalability bottleneck for such databases.

1.1 Thesis Statement

We observe that, in both cases, the lack of any assumptions about client behavior leads to write-synchronous designs, which results in mechanism coordination. By seemingly making no assumptions, prior systems actually implicitly make a pessimistic assumption — i.e., that any write can be observed, immediately after it is committed, together with any other committed write on which it depends, in a client transaction or session. In the case of slowdown cascades, the focus is on dependencies due to causal ordering of writes; whereas, in the case of range index contention, the focus is on dependencies between writes to the same range index, due to the requirement of maintaining a sorted order. In either case, before a write can be committed, prior systems must synchronously update their internal state to enforce *all* possible dependencies which *may* impact the correctness guarantee they provide.

The road to solving these problems starts from the above observation. If no client is going to observe all the dependencies, soon after the writes commit, then synchronously propagating writes is wasteful. It follows, that it is more scalable to enforce the dependencies of a write when they are actually observed by client reads, since doing so avoids mechanism coordination for all dependencies. This core idea is captured by the following thesis statement:

Scalable OLTP storage systems, that avoid mechanism coordination, can be built, by asynchronously propagating committed writes, and detecting and handling potential violations of correctness guarantees on reads; predicated on the assumption that, in the common case, dependencies of writes are unlikely to be tested by client reads, soon after the writes commit.

To validate this thesis' statement, we have built two systems that solve the two problems identified above by using the statement as a design principle. Occult [126] (Chapter 4) pushes the enforcement of transactional causal consistency to clients during reads, while allowing writes to be applied without any synchronous ordering. As a result, it is immune from slowdown cascades. ScaleDB (Chapter 5) updates range indexes asynchronously and avoids stale reads by using small hash *indexlets* to hold delayed updates. Using indexlets, it provides ACC, a novel, asynchronous concurrency control protocol which provides serializability without adverse performance effects on transaction execution in the common case.

1.2 Contributions

1.2.1 Partitioned and Geo-Replicated Datastores

We make the following contributions to the the design of partitioned and geo-replicated key-value stores that guarantee causal consistency to their clients.

1. A detailed explanation for why slowdown cascades present a clear and present danger to the scalability of any data store that delays writes to enforce causal consistency internally.

2. A novel and light-weight read-centric implementation of causal consistency. By shifting enforcement to the clients, it ensures that they never observe non-causal states, while placing no restrictions on the data store.

3. A new transactional isolation level called Per-Client Parallel Snapshot Isolation (PC-PSI), a variant of Parallel Snapshot Isolation (PSI) [145], that contributes to Occult's immunity to slowdown cascades by weakening how PSI replicates transactions committed at the same replica.

4. A novel scalable protocol for providing PC-PSI that uses causal timestamps to enforce both atomicity and transaction ordering and whose commit latency is independent of the number of replicas in the system.

5. An implementation and evaluation of Occult, the first causally consistent store that implements these ideas and is immune to slowdown cascades.

1.2.2 Multi-core, In-Memory Databases

We make the following contributions to the the design of multi-core, in-memory databases that guarantee serializability to their clients.

1. An analysis of the range index scalability bottleneck and of asynchronous range-index updates as a way to alleviate the bottleneck for unrelated transactions.

2. The design and implementation of ScaleDB, a scalable in-memory database that decouples range index management from transaction execution to allow asynchronous update of range indexes in the common case.

3. Asynchronous concurrency control (ACC), a novel concurrency control protocol that provides serializability in an asynchronous database. ACC uses *phantomlets* to scalably detect phantoms in range scans and provides scalable locking on keys in indexlets to atomically commit transactions.

4. A performance evaluation of ScaleDB on a dual-socket server with 36 cores, which shows that ScaleDB scales better than the Cicada [117] database.

1.3 Thesis Organization

The rest of this thesis is structured as follows: In Chapter 2, we first discuss the two scalability problems that were identified earlier in this chapter. We explain why they are instances of mechanism coordination. Next, in Chapter 3, we discuss how these seemingly disparate problems arise because of similar implicit assumptions, why those assumptions are overly pessimistic, and why the thesis statement is the right design principle to solve these problems. In Chapters 4 and 5 we discuss the design and evaluation of Occult and ScaleDB respectively and how both systems follow the thesis statement to solve these problems. Finally we conclude in Chapter 6.

Chapter 2

Mechanism Coordination

The disastrous impact of coordination on scalability is well-known. Frequent coordination between concurrent processes (or threads) in a system, results in the vast majority of their time being spent waiting for other processes (or threads). As a result, the scalability of such systems is limited. Conversely, it has been repeatedly shown that avoiding coordination allows systems to scale [160, 57, 70, 98]. For instance, Clements et al. [70] show that whenever operations of a software interface commute, they can be implemented in a way that is multi-core scalable; coordination between two commutative and concurrent interface calls is *unnecessary* since their results are independent of their order of execution.

In the context of OLTP storage systems, prior work on this topic has focused on the inherent coordination requirement (or otherwise) of various correctness guarantees. For example, in distributed systems, the CAP The-

Section 2.1 of this chapter is derived from a prior publication describing the Occult [126] system. The author of this dissertation conceived, designed, implemented and evaluated the Occult system, and led the writing of the prior publication: S. A. Mehdi, et al. 2017. “I Can’t Believe It’s Not Causal! Scalable Causal Consistency with No Slowdown Cascades”. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI’17)*. USENIX Association, USA, 453–468.

orem [89] precludes a group of servers, which are connected by a network subject to partitions, from implementing an atomic read/write register that responds to every client request. The necessity of an always connected network implies a synchronous coordination requirement on requests. Such coordination is therefore *necessary*; further implying that the scalability of a group of servers guaranteeing an atomic register (or a provably stronger guarantee such as linearizability [94]) is inherently limited. On the other hand, as we discuss in §2.1.1, if the group of servers were to guarantee causal consistency, then such coordination is provably *unnecessary*.

At this point, a relevant question is, when is coordination necessary for an OLTP storage system? Bailis et al. [50] approach this question by providing a taxonomy of various consistency and transactional isolation guarantees according to their compatibility with *high availability* i.e., the guarantee of "a response from each non-failing server in the presence of arbitrary network partitions". High availability implies that synchronous coordination between servers is not necessary and therefore permits scalability.

Later work from Bailis et al. [51] asks the question: "when is coordination strictly necessary to maintain application level consistency?". They address this question by moving away from widely used correctness guarantees (such as serializability [39]) and instead enlisting the "aid of application programmers to specify their correctness criteria in the form of invariants". While this approach achieves sizeable performance gains, it also places a significant burden on the application programmer – "an exercise in human proof generation" [159]. Brewer [62] refers to the need to know a system's invariants as "the hidden cost of forfeiting consistency" and points out that "the subtle beauty of a consistent system is that the invariants tend to hold even when the designer does not know what they are". The Anna key-value store [160] avoids coordination by using actors and asynchronous message passing, but focuses only on consistency guarantees that are compatible with high availability. Finally, the CALM theorem [92] explores the subset of programs that have a distributed coordination-free implementation by construction.

In this dissertation, we explore a different dimension of this question. Rather than restricting the application programming model or moving away from widely used correctness guarantees, we assume traditional correctness guarantees and typical storage system interfaces [9, 34], and instead ask the question: "When is coordination unnecessary because it is an artifact of the design decisions (or mechanisms) used to implement a system rather than a fundamental requirement of the guarantees provided by the system?". We refer to such unnecessary coordination as *mechanism coordination*.

It is important to understand how mechanism coordination relates to problems like false sharing [59], lock contention [147] or unnecessary network communication in a distributed system. Mechanism coordination is a higher-level concept, related to the design choices made while implementing a correctness guarantee, that can manifest as any of these lower-level problems. It is possible, however, to have these lower-level problems exist in a system that does not have mechanism coordination. For instance, a serializable database, designed to avoid mechanism coordination, can still have false sharing due to improper cache-alignment of shared data structures. Conversely, mechanism coordination can show up in ways that might not be traditionally considered as coordination. For instance, as we show in §2.2.3.1, mechanism coordination in a serializable database can show up in the form of transactional aborts.

In the rest of this chapter, we explore two different instances of mechanism coordination in two different classes of storage systems.

2.1 Slowdown Cascades in Causal Datastores

When systems scale to sufficient size, failures become an inevitable and regular part of their operation [79, 80]. Performance anomalies, e.g., one node running with lower throughput than the rest of the system, are typical, and can be viewed as a kind of partial failure. Potential causes of such failures include abnormally-high read or write traffic, partially malfunctioning (or "fail-slow") hardware [91], bugs or misconfigurations [95, 116], or a localized network issue, like congestion in a top-of-rack switch.

A recent paper [91] studied 101 reports of fail-slow hardware incidents in large production systems. Some examples from the study included network delays of up to hundreds of milliseconds caused by loose network cables and pinched fiber optics, a degraded NIC due to a non-deterministic network driver bug in Linux that only surfaced on one machine and a power supply failure that throttled the CPUs on four machines by 50%.

In a partitioned system, a failure within a partition will inevitably affect the performance of that partition. A *slowdown cascade* occurs when the failure spills over to affect other partitions. For instance the study in [91] cites the example of a degraded NIC (from 1 Gbps to 1 Kbps) in one machine causing a chained reaction that slowed down an entire cluster of 100 machines.

2.1.1 Causal Consistency

Causal consistency [41] is a well-known consistency model that captures the potential causal relationships between operations performed by concurrent processes. When applied in the context of a storage system, it ensures that clients observe their own updates and read from a state that includes all operations that they have previously observed.

This simple guarantee has clear benefits for users of today's large-scale web applications backed by sharded and geographically replicated data stores. For example, consider two users – Alice and Bob – of a large social network backed by such a data store. Causal consistency is all that is needed to preserve operation ordering and give Alice assurance that Bob, whom she had defriended before posting her Spring-break photos, will not be able to access her pictures, even though Alice and Bob access the social network using different replicas [52, 71, 119].

Further, it has been shown that no guarantee stronger than real-time causal consistency can be provided in a replicated data store that combines high availability with convergence [123], and that, conversely, it is possible to build convergent causally-consistent data stores that can efficiently handle a large number of shards [44, 53, 83, 84, 119, 120]. This implies that in a replicated data-store that guarantees causal consistency, any synchronous coordination between replicas is *unnecessary* for answering client requests.

Thus, without imposing the high latency of stronger consistency guarantees [89, 118], causal consistency provides a sweet spot in the debate on the guarantees that a sharded and geographically replicated (geo-replicated) data store should offer. On the one hand, causal consistency maintains most of the performance edge of eventual consistency [155] over strong consistency, as all replicas are available for reads under network partitions [89, 118]. On the other hand, it minimizes the associated baggage of increased programmer complexity and user-visible anomalies. By ensuring that all clients see updates that may potentially be causally related [108] in the same order, causal consistency can, for example, address the race conditions that a VP of Engineering at Twitter in a 2013 tech talk called "the biggest problem for Twitter" [106]: when fanning out tweets from celebrities with huge followings, some feeds may receive reactions to the tweets before receiving the tweets themselves.

Despite the obvious benefits, however, causal consistency is largely not deployed in production systems, as existing implementations are liable to experience, at scale, one of the downsides of strong consistency: *slowdown cascades*.

2.1.2 How can Slowdown Cascades Impact Causal Datastores?

Industry has long identified the spectre of slowdown cascades as one of the leading reasons behind its reluctance to build strongly consistent systems [42, 58], pointing out how the slowdown of a single shard, compounded by *query amplification* (e.g., a single user request in Facebook can generate thousands of, possibly dependent, internal queries to many services), can quickly cascade to affect the entire system.

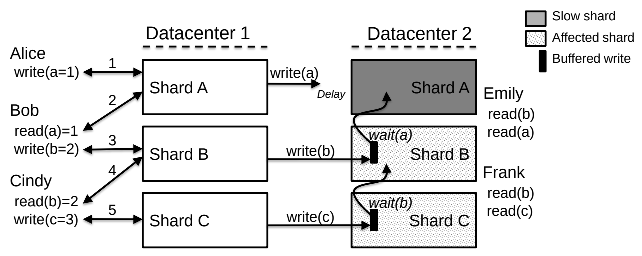


Figure 2.1: Example of a slowdown cascade in traditional causal consistency. Delayed replicated write(a) delays causally dependent replicated write(b) and write(c)

All prior causally consistent systems [53, 84, 119, 120, 165] are susceptible to slowdown cascades. The reason, in essence, is that, to present clients with a causally consistent data store, these systems delay applying a write w until after the data store reflects all the writes that causally precede w. For example, in Eiger [120] each replicated write w carries metadata that explicitly identifies the writes that directly precede w in the causal dependency graph. The datacenter then delays applying w until these dependencies have been applied locally. The visibility of a write within a shard can then become dependent on the timeliness of other shards in applying their own writes. As Figure 2.1 shows, this is a recipe for triggering slowdown cascades: because shard A of DC_2 lags behind in applying the write propagating from DC_1, all shards in DC_2 must also wait before they make their writes visible. Shard A's limping inevitably affects Emily's query, but also unnecessarily affects Frank's, which accesses exclusively shards B and C.

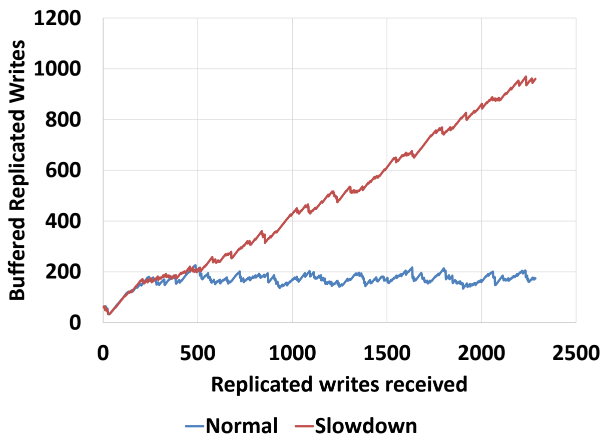


Figure 2.2: Average queue length of buffered replicated writes in Eiger under normal conditions and when a single shard is delayed by 100 ms.

In practice, even a modest delay can trigger dangerous slowdown cascades. Figure 2.2 shows how a single slow shard affects the size of the queues kept by Eiger [120] to buffer replicated writes. Our setup is geo-replicated across two datacenters in Wisconsin and Utah, each running Eiger sharded across 10 physical machines. We run a workload consisting of 95% reads and 5% writes from 10 clients in Wisconsin and a read-only workload from 10 clients in Utah. We measure the average length of the queues buffering replicated writes in Utah. Larger queues mean that newer replicated writes take longer to be applied. If all shards proceed at approximately the same speed, the average queue length remains stable. However, if *any* shard cannot keep up with the arrival rate of replicated writes, then the average queue length across *all* shards grows indefinitely.

2.1.3 Nature of Mechanism Coordination

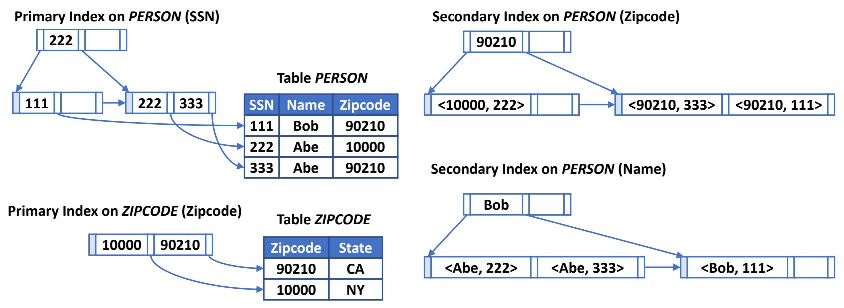
To understand the nature of mechanism coordination in this problem consider Figure 2.1. It illustrates why causal consistency can be subject to slowdown cascades despite replicating writes asynchronously: writes share the fate of all other writes on which they causally depend. If one write is slow to replicate, all subsequent writes will incur that delay. Though this delay may sometimes be necessary - Emily must wait for the delayed write to observe a causally consistent snapshot of the system - inheriting the delays of causally preceding writes can also introduce gratuitous blocking. Frank, for instance, never reads Alice's write (a): delaying writes (b) and (c) until (a) is replicated is thus unnecessary. Otherwise said, observing writes (b) and (c) while write (a) is in flight does not lead to a consistency violation.

This observation precisely captures what causal consistency *requires*; therefore, it is key to understanding why this approach encounters mechanism coordination. Causal consistency defines a contract between the data store and its users that specifies, for a given set of updates, which values the data store is allowed to return in response to user queries. In particular, it guarantees that each client observes a monotonically non-decreasing set of updates (including its own), in an order that respects potential causality between operations. Causal consistency thus mandates that Frank, upon observing write (c), also observes write (b), but remains silent on the fate of write (a). Existing causally consistent systems, however, enforce internally a stronger invariant than what causal consistency requires: to ensure that clients observe a monotonically non-decreasing set of updates, they evolve each datacenter (or replica) *only* through monotonically non-decreasing updates. It is this strengthening that leaves current implementations of causal consistency vulnerable to slowdown cascades.

2.2 Range Index Contention in In-Memory Databases

Range indexes are an efficient method for data retrieval. In addition to providing exact-match lookup of database records in logarithmic time, they also allow fast scans of records in sorted order. Figure 2.3, shows the use of range indexes in a simple database with two tables. Tables are implemented as collections of indexes and include one primary index and zero or more secondary indexes. For example, table PERSON has primary index SSN and two secondary indexes, Name and Zipcode. Table records are stored on the heap and pointed to by the table's primary index.

A primary range index allows quick retrieval of a table's records by primary key for both point and range queries. Primary keys are often required to be unique within a table and an index can enforce this *uniqueness constraint* efficiently. Applications also use secondary indexes extensively. They support analytical queries [68] and are crucial to maintaining the consistency of the database by serving as *foreign keys*, i.e. columns of a table that refer to


Primary Index on *PERSON* (SSN)
222
111
222
333
Secondary Index on *PERSON* (Zipcode)
90210
<10000, 222>
<90210, 333>
<90210, 111>
Secondary Index on *PERSON* (Name)
Bob
<Abe, 222>
<Abe, 333>
<Bob, 111>
Primary Index on *ZIPCODE* (Zipcode)
10000
90210
Table *PERSON*
Table *ZIPCODE*

SSN	Name	Zipcode
111	Bob	90210
222	Abe	10000
333	Abe	90210

Zipcode	State
90210	CA
10000	NY

Figure 2.3: Layout of a simple database with range indexes. Tables are logical entities. Records are stored sorted by primary index key. Schema information is stored separately in a catalog (not shown). Arrows are pointers.

a primary key of another table. For example, a foreign key constraint on the Zipcode column in the *PERSON* table implies that deleting the 90210 zipcode from the *ZIPCODE* table requires deletion of all records with the 90210 zipcode from the *PERSON* table. The secondary index on the Zipcode column makes this operation efficient—in the Figure, the root node of the corresponding secondary tree points directly to the range of all SSNs in the 90210 zipcode; we can use these values as keys to traverse the primary index of the *PERSON* table.

2.2.1 Range Index Scalability

Despite decades of work [90, 113, 128, 66, 65, 124, 114, 115], scalability of range indexes under concurrent accesses remains elusive. This is primarily due to the hierarchical nature of these data structures. For instance, when inserting or deleting a single record in a B+-tree, its leaf node structure might need to change, potentially requiring the atomic update of a chain of internal nodes all the way to the root. Performing such modifications atomically while supporting concurrent access from multiple threads requires synchronization [86, 157].

One approach to synchronization uses locks [90, 113, 128]. Recent optimizations [66, 65, 124] remove shared cache line contention between readers trying to acquire a lock per node, by making them optimistic. However, readers must read a version number per node to verify their optimistic assumption, which can cause contention with writers trying to increment it. Similarly, writers still contend on cache lines, trying to acquire spinlocks on individual tree nodes. Frequently accessed nodes such as the root of a B$^+$tree or the index node at the end of the range (for append workloads) can become hotspots of contention. Figure 2.4 illustrates this further with an example.

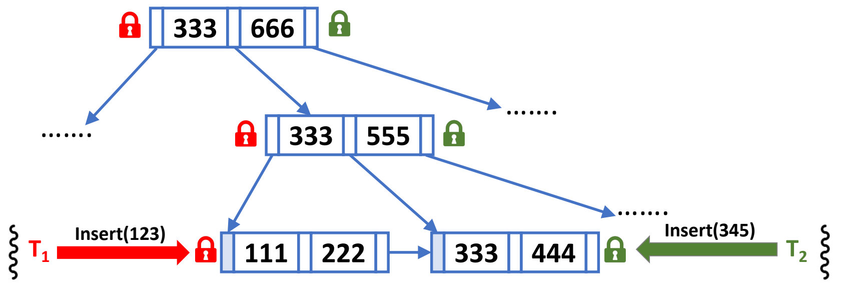


Figure 2.4: Example of contention on a B+ tree range index. Concurrent transactions T_1 and T_2, running on different threads, are inserting records with keys 123 and 345 respectively to a B+ tree. The relevant leaf nodes, $[111, 222]$ and $[333, 444]$, are full, so they need to be split and the shared parent node $[333, 555]$ needs to be updated with pointers to the new children. This results in lock contention between T_1 and T_2 on the shared parent. However, since the shared parent is full as well, the lock contention continues to the root.

An alternative is to use lock-free data structures [115, 88, 93]: they use atomic operations and multi-versioning to avoid lock contention for long critical sections. Yet, as recent work [86] points out, their theoretical guarantees are “mostly irrelevant to performance and scalability on multi-core hardware”, since they cannot avoid contention on global memory locations.

A recent study [157] evaluated state-of-the-art range indexes [124, 115, 114, 90, 88] on the YCSB [72] benchmark and showed that none of these indexes scale well. Even on a read-heavy workload with only 5% inserts, these indexes only scale up to 12× when increasing cores by 20×. On an insert-only workload with threads appending new inserts to the end of a range, their scalability collapses when going from a single NUMA node (20 cores) to two NUMA nodes (40 cores), with throughput dropping between 50% to 66%.

2.2.2 In-Memory Databases

In-memory databases [22, 16, 32, 81, 7, 33] primarily rely on main memory for storing their data. They can efficiently support real-time applications that disk-based databases cannot support [1], such as real-time bidding for online advertisements [35] and operational analytics [26]. Several application and hardware trends promise to increase the prevalence of in-memory databases as well as their scalability requirements.

Database application workloads are becoming increasingly demanding. They can be simultaneously write and read intensive; require both low transaction commit latency and high transactional throughput; and, increasingly, support analytical queries (on data from automated sources such as sensors, real-time analytics, and machine learning [38, 29, 100]). Some production databases are already serving both transactional and analytical workloads [68] requiring read and write-heavy queries to execute under tight latency demands. Analytical queries require creating and maintaining many indexes. Running simultaneous transactional workloads requires maintaining those indexes under a high rate of inserts or updates, while also continuing to provide low-latency point reads.

In-memory databases are particularly suited to handle these diverse workload requirements, and their adoption is further facilitated by the large-scale availability of high-capacity non-volatile memory (NVM), as it allows for more data to be held in memory, with access latencies comparable to DRAM [99]. This allows transaction durability with substantially lower commit latencies [47] by obviating the need to persist a log on disk or SSD (as done in traditional in-memory databases). As a consequence, however, the *variance* in commit latency due to contention on in-memory data structures is becoming more visible. Just as NVM is shifting performance bottlenecks away from storage and towards multi-core CPU contention, the diverse application workload requirements are raising the bar for in-memory database scalability.

2.2.3 In-Memory Database Scalability

Much of the last decade's work on scalable in-memory databases [111, 81, 102, 103, 117, 96, 85, 132, 156, 129] has focused on scaling serializable ordering on *contended* transactional workloads. Serializability [39] requires respecting the data dependencies that arise between transactions that read and write the same database record. Though such *true dependencies* are ultimately a barrier to full scalability, various techniques can reduce their impact, including multi-versioning [111, 85, 117], static analysis [156, 129], exploiting commutativity in some workloads [96] and backoff [117].

However, these techniques are orthogonal to scalability problems for *uncontended* transactional workloads, where the architecture of the database, through mechanism coordination, introduces scalability bottlenecks among unrelated transactions. Existing work in this area [152, 117, 164] has focused on timestamp allocation i.e. the use of a shared timestamp for ordering transactions [56, 107], which was shown to be a principal bottleneck to the scalability of concurrency control [163].

An approach proposed by the H-store [101, 137] project, avoids mechanism coordination for some applications, by partitioning the database and accessing each partition from a single thread. This scales really well for applications whose databases are amenable to a clean partitioning and where most transactions only access a single partition. However, many applications do not fit this profile and the performance in those cases can be worse [135, 136].

2.2.3.1 Mechanism Coordination on Range Indexes

Mechanism coordination can also be caused by range indexes. Concurrent inserts and deletes to different records can contend on atomically modifying the hierarchical internal structure of a range index (§2.2.1). This contention qualifies as mechanism coordination, for a serializable database; the specification of serializability [39] does not require ordering transactions that

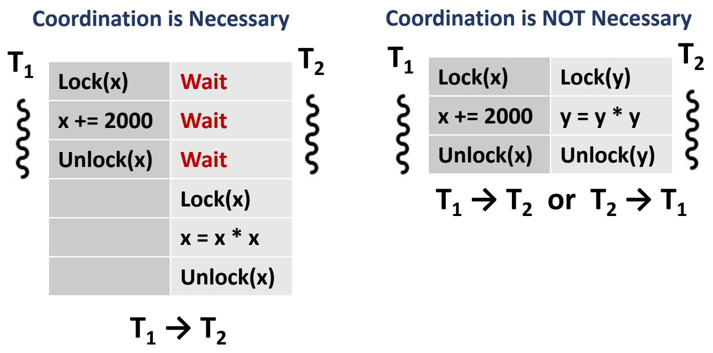


Figure 2.5: Serializability and the necessity of coordination. Two concurrent transactions T_1 and T_2 are running on separate threads. If they access the same record(s), then coordination is necessary for serializing their order (here $T_1 \rightarrow T_2$). Otherwise, coordination is unnecessary, since they can be serialized in either order. Any contention on range indexes (e.g. Figure 2.4) is therefore mechanism coordination for guaranteeing serializability.

access disjoint sets of records, since any ordering can be equivalent to a serial order. This is illustrated further by the example in Figure 2.5.

To understand the impact of range index mechanism contention[1] on database scalability, we evaluated Cicada [117], a state-of-the-art scalable in-memory database that guarantees serializability. Cicada avoids the timestamp allocation bottleneck by using loosely synchronized clocks for ordering transactions. It was shown to be more scalable than several other databases [152, 164, 102, 103, 81]. However, as we show next, it still incurs range index mechanism contention.

Figure 2.6 shows Cicada's goodput scalability (relative to a single core) on TPC-C [31], a standard OLTP benchmark simulating purchase transactions on a configurable number of independent warehouses. We use a machine with

[1]For the range index contention problem, we use the terms mechanism contention and mechanism coordination interchangeably

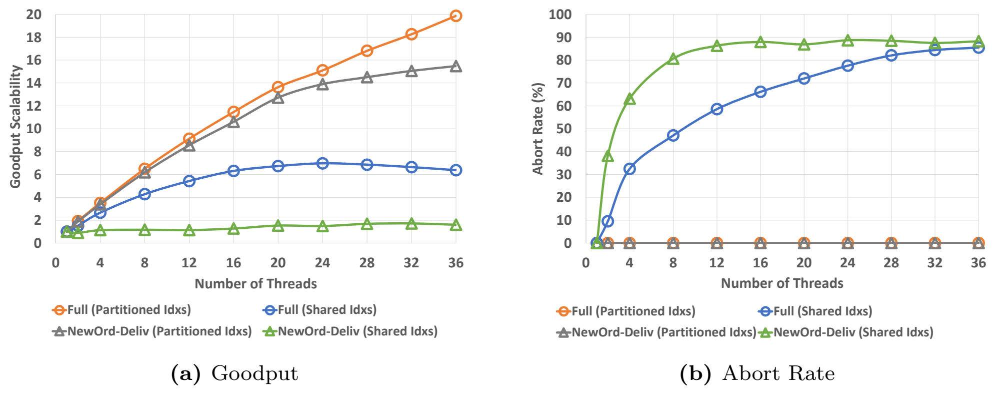


Figure 2.6: Cicada scalability on the TPC-C benchmark with partitioned vs shared indexes. Goodput counts only committed transactions. Full means the canonical TPC-C workload (i.e., 45% New-Order, 43% Payment, 4% Delivery, 4% Order Status and 4% Stock Level). NewOrd-Deliv means a 50% New-Order and 50% Delivery workload.

two CPU sockets, each with an 18-core Intel Xeon Gold 6154 CPU. We increase the number of transaction processing server threads from 1 to 36 and use as many warehouses as threads for each data point. This configuration ($C_{wh=thd}$) has very low true contention, since threads (almost always) run queries on their own warehouses, thus avoiding contention on the same records with other threads. Therefore, $C_{wh=thd}$ allows us to isolate and understand the scalability impact of mechanism contention on shared indexes.

By default, the Cicada prototype partitions (§2.2.3) 8 out of the 9 TPC-C tables and their associated indexes by the warehouse id (w_id). This is possible because, these tables have compound primary keys, formed from multiple columns of each table, with the w_id as the starting part of each key[2]. For instance, the NEW-ORDER table has a compound primary key formed

[2]The HISTORY table has no primary key index. The table is still partitioned by w_id

from concatenating three columns (in order): the warehouse id (w_id), the district id (d_id), and the new order id (o_id). When partitioned by w_id, each warehouse has a separate range index which only has keys starting with that warehouse's id. This means that if thread t_1 inserts a new order for warehouse w_1, it does not contend on the same range index structure with another thread t_2 inserting a new order for warehouse w_2. The 9th table (named ITEM) is not partitioned and has a shared primary key index on item id. However, the ITEM table receives no inserts or updates during the benchmark run. As a result, Cicada's partitioned configuration with $C_{wh=thd}$ does not have any mechanism contention.

Figure 2.6a shows how Cicada's scalability is impacted when we make all the indexes shared across all threads while keeping the tables partitioned. On the canonical TPC-C workload (labeled Full in Figure 2.6a), the partitioned index configuration scales well. However, the shared index configuration scales poorly on this workload and in fact stops scaling beyond 24 cores. As pointed out earlier in §2.2.3, while the partitioned approach works well for some applications (such as TPC-C with $C_{wh=thd}$), it does not perform well in general.

To further explore the impact of range index contention, we experiment with a workload consisting of two TPC-C transactions – New-Order and Delivery – in equal proportions (NewOrd-Deliv in Figure 2.6a). This workload exacerbates the contention on primary range indexes of the ORDER-LINE and the NEW-ORDER tables, by increasing the ratios of the two transactions from

45% (New-Order) and 4% (Delivery) in the canonical workload to 50% each. Each New-Order transaction performs one insert into the NEW-ORDER table and ten inserts (on average) into the ORDER-LINE table. Each Delivery transaction performs range scans on the NEW-ORDER and ORDER-LINE tables and also deletes a key from the NEW-ORDER table. As Figure 2.6a shows, the increased contention due to these concurrent inserts and range scans on the same indexes limits scalability for the shared index configuration.

Finally, Figure 2.6b shows the underlying reason for the poor scalability of Cicada on the shared index configuration – an increasingly high abort rate. Cicada uses multi-version concurrency control (MVCC) for both its records and indexes. These multi-version indexes were designed to reduce multi-core contention on the same index nodes by multiple threads; instead, if an index node needs to be modified, Cicada creates a new version in thread local memory and installs it into the index on successful transaction commit. However, in order to enforce serializability, any transaction that does a range scan needs to validate at transaction commit that no new records were inserted after the range scan execution, that match the range scan predicate (i.e., *avoid phantoms*). For this purpose, at transaction commit, Cicada validates all index nodes whose key range intersected with the range scan predicate. This validation can fail, resulting in transaction aborts. Consequently, range index contention in Cicada shows up as transaction aborts instead of cache-line contention on index nodes.

Figure 2.6b shows that, for the shared index configuration, the abort rate rises more sharply for the NewOrd-Deliv benchmark than the canonical TPC-C benchmark, which matches the goodput scalability difference between them. The abort rate for the partitioned configuration remained <=0.01% for both benchmarks (with NewOrd-Deliv having a higher rate than Full), thus confirming that the high abort rate was due to contention on shared indexes.

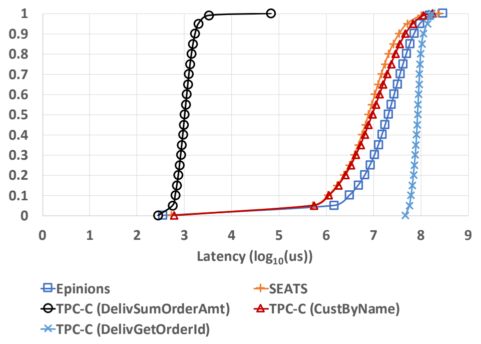
Chapter 3

Towards Scalability through Asynchrony

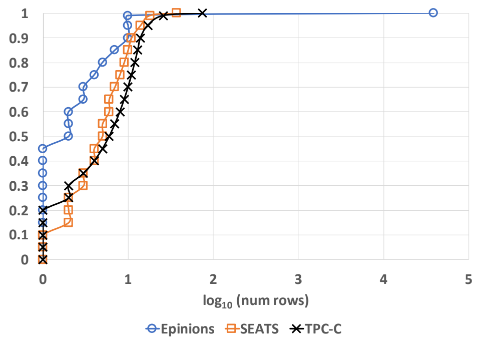
The discussion in the previous chapter (§2.1.3 and §2.2.3) explained the nature of mechanism coordination in the two problems; thus, assuring us that the coordination is not necessary according to the specification of the guarantees being provided by these systems. Yet, to move towards a solution, it is helpful to understand why those mechanisms are also overly pessimistic from the perspective of the clients accessing these systems; thus yielding no significant benefit in terms of performance. The discussion in this chapter will elucidate why the assumption in our thesis statement (§1.1) is practical.

3.1 Implicit Pessimistic Assumptions of Prior Systems

The assumption implicit in the design of causal datastores, that are susceptible to slowdown cascades, is that a single client can read either the entire datastore or a large part of it within a single session or transaction. If that were true, it would justify evolving entire datacenters only through monotonically non-decreasing updates. Yet, the published literature on OLTP systems used by organizations such as Facebook [63, 144], LinkedIn [6] and Twitter [27, 162], does not indicate any such pattern in their workloads. The number of objects



(a) Write to range scan (W-to-RS) latency



(b) Records returned by range scan

Figure 3.1: Range scan property distributions of three benchmarks.

Benchmark	Read Txns	Range Scans	Database size
TPC-C	8%	7.83%	10 warehouses
SEATS	45%	23%	100K customers
Epinions	50%	100%	200K users

Table 3.1: Benchmark details.

accessed by a single client session or transaction are a negligible fraction of an entire datacenter, given their scale.

3.1.1 In-Memory Databases

The analysis in §2.2.3 demonstrates that scalability in state-of-the-art databases is primarily limited by contention on range indexes. A key contributor to this contention is the synchronous nature of the updates to range indexes that take place once a transaction commits. Of course, all indexes, be it range or hash, need to return on a query the most recently committed record corresponding to an index key, but range indexes have an additional obligation: they need to ensure that range scans issued immediately after a transaction commits will not miss any record inserted or updated by the transaction. It is to guarantee this property that the record is inserted synchronously into all primary and secondary indexes; this requires sorting the record with respect to *all* previously inserted records in the table, but it also creates contention between otherwise non-conflicting transactions on the internal nodes of a range index.

To understand whether this is too pessimistic, we run an experiment to measure the latency between the last time a record is written (inserted or updated) and when it is read as part of a range scan (*W-to-RS latency*). We use three transactional application benchmarks from the OLTP-bench [82] suite, designed to evaluate modern cloud database workloads. As Table 3.1 shows, these benchmarks range from moderately write-heavy (Epinions) to very write-heavy (TPC-C), and from minimal range scans to all range scans as a fraction of all read queries. We ran these benchmarks on a MySQL 8.0 instance running on a 20 core (40 hardware threads) Intel Xeon machine, with as many clients as needed to saturate throughput. We emulate an in-memory database by setting the MySQL in-memory buffer pool to a large-enough size, so that in all three cases the entire database fits in memory and the workload is never disk-bound.

Figure 3.1a shows the cumulative distribution of the W-to-RS latency. We use a single curve to characterize the behavior of the different range scans in Epinions and Seats: we find that the 5th percentile W-to-RS latency is above 500ms and the median is between 8 and 85 seconds. We instead report the latency of each range scan in TPC-C separately, since they behave quite differently: DelivSumOrderAmt, a range scan on a primary index responsible for 3% of all TPC-C read queries has a median W-to-RS latency of 1ms; the other two TPC-C range scans are on secondary indexes and their median W-to-RS latencies are orders of magnitude higher. Epinions and SEATS also show lower W-to-RS latency for range scans of primary indexes, though with a much smaller ($2\times$ to $5\times$) gap. The low W-to-RS latency of DelivSumOrderAmt is due to the TPC-C Delivery txn. Delivery contains an update followed by a read on the same range in the Orderline table.

Figure 3.1b shows the distribution of the number of records read by range scans in each benchmark. For all benchmarks and all range scans, the median number of records read was at most 6, while the 99th percentile was at most 26 records. Epinions had two range scans in read-only transactions that read thousands of records. However these range scans comprised only 0.068% of all read queries in Epinions and had a median W-to-RS latency of at least 66 seconds.

For brevity, we omit similar analysis for point queries, but their behavior was mixed. For instance, in TPC-C, four point queries had median Write-to-Point-Query (W-to-PQ) latencies ranging from 350μs to 21ms. These point queries read heavily updated records in the District and Warehouse tables. They are a part of the NewOrder and Payment read-write transactions which together comprise 90% of the benchmark. On the other extreme, two point queries in TPC-C had median W-to-PQ latencies of 4 and 15 seconds.

The overall picture that emerges from this analysis is the following:

1. While point queries often read recently written records, for range queries that is the exception rather than the rule. This holds especially true for secondary indexes.

2. For the vast majority of cases, the number of records that a range query reads (especially as part of read-write transactions) is small.

3. Large range scans rarely happen. If they do happen, they are usually a part of a read-only transaction.

3.2 The Thesis Statement as a Design Principle

We now turn to the thesis statement (§1.1) and understand how it follows from the analysis presented above and why it is the right design principle for avoiding mechanism coordination.

The first aspect, clear from the analysis, is that synchronously propagating writes is neither necessary to enforce the *specification* of the correctness guarantees nor relevant for performance, given the behavior of the clients accessing these systems. Thus, writes should be done asynchronously, as stated in the first part of the thesis statement.

Some synchronization for writes is hard to avoid. For example, atomically committing the writes of a transaction in a distributed system requires running a 2-phase commit protocol, while holding locks. Yet as we discuss in §4.4.2, we can even design the transaction commit protocol in a way that, once a transaction commits, replication happens asynchronously, *even for writes within a transaction*, while still providing atomicity.

This leaves the question of how to actually enforce correctness guarantees, given that writes are no longer synchronous? We argue that enforcing guarantees on reads is a natural alternative – actually preferable to writes – since it only pays the cost of synchronization when needed. An important trend – in addition to the exponentially growing scalability demands – is that workloads are increasingly write-heavy; since, databases are increasingly storing machine-generated data, such as outputs of real-time analytics workloads or time-series data (e.g. generated by DevOps monitoring or IoT sensors) [29]. It follows, that the problems with synchronous writes are going to get worse. That makes enforcing guarantees on reads even more attractive.

Finally the question is *how* to enforce guarantees on reads? Given the asynchronous propagation of writes, reads can potentially observe stale or inconsistent data; thus, breaking the correctness guarantee. For this purpose, we need to be able to detect potential violations of correctness guarantees. If the read operation would violate the guarantee, then we can either synchronously fix the data we are reading, or if running within a transaction, we can abort to keep the correctness guarantee intact.

As we show in the next two chapters, our thesis statement is the core design principle for two systems that solve the scalability problems we described in Chapter 2.

Chapter 4

Occult: Causal Consistency with No Slowdown Cascades

Occult (**O**bservable **C**ausal **C**onsistency **U**sing **L**ossy **T**imestamps) is the first geo-replicated and sharded data store that provides causal consistency to its clients without exposing the system to slowdown cascades. To make this possible, Occult shifts the responsibility for the enforcement of causal consistency from the data store to its clients. The data store makes its updates available as soon as it receives them, and causal consistency is enforced on reads only for those updates that clients are actually interested in observing. In essence, Occult decouples the rate at which updates are applied from the performance of slow shards by optimistically *rethinking the sync* [133]: instead of enforcing causal consistency as an invariant of the data store, through its read-centric approach Occult appears to applications as *indistinguishable* from a system that does.

Because it never delays writes to enforce consistency, Occult is immune

This chapter is derived from a prior publication describing the Occult [126] system. The author of this dissertation conceived, designed, implemented and evaluated the Occult system, and led the writing of the prior publication: S. A. Mehdi, et al. 2017. "I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades". In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, USA, 453–468.

from the dangers of slowdown cascades. It may, however, delay read operations from shards that are lagging behind to ensure they appear consistent with what a user has already seen. We expect such delays to be rare in practice because a recent study of Facebook's eventually-consistent production system found that fewer than six out of every million reads were not causally consistent [122]. Our evaluation confirms this. We find that our prototype of Occult, when compared with the eventually-consistent system (Redis Cluster) it is derived from, increases the median latency by only 50μs, the 99th percentile latency by only 400μs for a read-heavy workload (4ms for a write-heavy workload), and reduces throughput by only 8.7% for a read-heavy workload (6.9% for a write-heavy workload).

Occult's read-centric approach, however, raises a thorny technical issue. Occult requires clients to determine how their local state depends on the state of the *entire* data store; such global awareness is unnecessary in systems that implement causal consistency within the data store, where simply tracking the immediate predecessors of a write is enough to determine when the write should be applied [119]. In principle, it is easy to use vector clocks [87, 125] to track causal dependencies at the granularity of objects or shards. However, their overhead at the scale that Occult targets is prohibitive. Occult instead uses *causal timestamps* that, by synthesizing a variety of techniques for compressing dependency information, can achieve high accuracy (reads do not stall waiting for updates that they do not actually depend on) at low cost. We find that 24-byte timestamps suffice to achieve an accuracy of 99.6%; 8

more bytes give an accuracy of 99.96%.

Causal timestamps also play a central role in Occult's support for scalable read-write transactions. Transactions in Occult operate under a variant of Parallel Snapshot Isolation [146]. Occult ensures that all transactions always *observe* a consistent snapshot of the system, even though the datastore no longer evolves through a sequence of monotonically increasing consistent snapshots. It uses causal timestamps to not only track transaction ordering but also atomicity (by making writes of a transaction *causally dependent on each other*). This novel approach is key to the scalability of Occult's transactions and their immunity to slowdown cascades. The responsibility for commit is again shifted to the client, which uses causal timestamps to detect if a transaction has observed an inconsistent state due to an ordering or atomicity violation and, if so, aborts it. Committed writes instead propagate asynchronously to slaves, allowing the commit logic to scale independently of the number of slave replicas.

4.1 Observable Causal Consistency

Like every consistency guarantee, causal consistency defines a contract between the data store and its users that specifies, for a given set of updates, which values the data store is allowed to return in response to user queries. In particular, causal consistency guarantees that each client observes a monotonically non-decreasing set of updates (including its own), in an order that respects potential causality between operations.

To abide by this contract, existing causally consistent data stores, when replicating writes, enforce internally a stronger invariant than the contract requires: they ensure that clients observe a monotonically non-decreasing set of updates by evolving their data store only through monotonically non-decreasing updates. This strengthening satisfies the contract but, as we saw in §2.1, leaves these systems vulnerable to slowdown cascades.

To resolve this issue, Occult moves the output commit to the clients: letting clients themselves determine when it is safe to read a value frees the data store to make writes visible to clients immediately, without having to first apply all causally preceding writes. Given the duties that many causally consistent data stores already place on their clients (such as maintaining the context of dependencies associated with each of the updates they produce [119, 120]), this is only a small step, but it is sufficient to make Occult impervious to slowdown cascades.

Furthermore, Occult no longer needs its clients to be sticky (real-world systems like Facebook sometimes bounce clients between datacenters because of failures, load balancing, and/or load testing [42]). By empowering clients to determine independently whether a read operation is safe, it is no longer problematic to expose a client to the state of a new replica R2 that may not yet reflect some of the updates the client had previously observed on a replica R1.

The general outline of a system that moves the enforcement of causal consistency to read operations is straightforward. Each client c needs to maintain some metadata to encode the most recent state of the data store that it has observed. On reading an object o, c needs to determine whether the version of o that the data store currently holds is safe to read (i.e., if it reflects all the updates encoded in c's metadata): to this end, the data store could keep, together with o, metadata of its own to encode the most recent state known to the client that *created* that version of o. If the version is deemed safe to read, then c needs to update its metadata to reflect any new dependency; if it is not, then c needs to decide how to proceed (among its options: try again; contact a master replica guaranteed to have the latest version of o; or trade safety for availability by accepting a stale version of o).

The key challenge, however, is identifying an encoding of the metadata that minimizes both overhead and read latency. Since each object in the data store must be augmented with this metadata, the importance of reducing its size is obvious; keeping metadata small, however, reduces its ability to track causal dependencies accurately. Any such loss in definition is likely to introduce spurious dependencies between updates. Although these dependencies can never lead to slowdown cascades in Occult, they can increase the chances that read operations will be unnecessarily delayed. Occult's compressed causal timestamps leverage structural and temporal properties to strike a sweet spot between metadata overhead and accuracy (§4.3).

These causal timestamps have another, perhaps more surprising consequence: they allow Occult to offer the first scalable implementation of causal read-write transactions (§4.4). Just as the data-store need not be causal, transactions need not take effect atomically in the datastore. They simply need to *appear atomic* to clients. To achieve this, Occult makes a transaction's writes *causally depend on each other*. This guarantees that clients that seek to read multiple writes from a transaction will independently determine that they must either observe all of the transactions's writes, or none. In contrast, transactions that seek to read a single of the transaction's writes will not be unnecessarily delayed until other replicas have applied writes that they are not interested in. Once again, this is a small step that yields big dividends: transactional writes need no longer be replicated synchronously for safety, obviating the possibility of slowdown cascades.

4.2 Occult: The Basic Framework

We first outline the system model and an idealized implementation of Occult's basic functionality: clients that read individual objects perceive the data store as causally consistent. We discuss how to make the protocol practical in §4.3 and sketch Occult's more advanced features (transactions) in §4.4.

4.2.1 System Model

Occult is a sharded and replicated key-value store where each replica is located in a separate datacenter with a full copy of the data. The keyspace is divided into a large number of *shards*, i.e., disjoint key ranges. There can be tens or hundreds of thousands of shards, of which multiple can be colocated on the same physical host.

We assume an asynchronous master-slave replication model, with a publicly designated master for every shard. This master shard accepts writes, and asynchronously, but in order, replicates writes to the slave shards. This design is common to several large-scale real-world systems [63, 71, 134, 139] that serve read-heavy workloads with online queries.The master shard asynchronously, but in order, replicates writes to the slave shards. We assume the master for each shard is globally known, e.g., through a separate configuration service. The masters for different shards can be in different replicas. Master-slave replication has higher write latency than multi-master schemes, but avoids the complexity of dealing with concurrent conflicting writes that can lead to lost updates [119] or require more complex programming models [77]. In addition, it provides a primary source of truth, which has been cited by some production systems [139] as the reason for their choice.

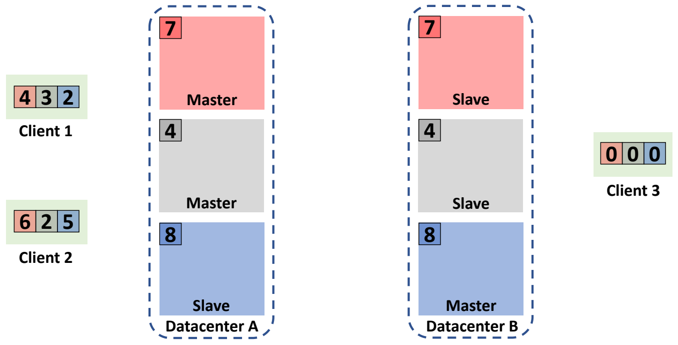
Clients in Occult are co-located with a replica in the same datacenter. Each client reads from its local replica and writes to the master shard (possibly located in a remote replica); a client library enforces causal consistency for reads and attaches meta-data to writes. While clients normally read from the shards in their replica, there is no requirement for them to be "sticky" (§4.1).

4.2.2 Causal Timestamps

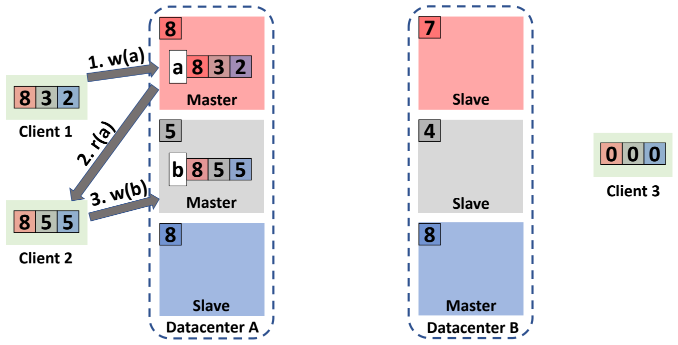
Occult tracks and enforces causal consistency using shardstamps and causal timestamps. A shard's *shardstamp* counts the writes that the shard (master or slave) has accepted. A *causal timestamp* is a vector of shardstamps that identifies a global state across all shards: each entry stores the number of known writes from the corresponding shard. Keeping an entry per shard rather than per object trades-off accuracy against meta-data overhead: in exchange for smaller timestamps, it potentially creates false dependencies among all updates to objects mapped to the same shard. We argue that this is the right trade-off to make. In the common case, each master shard will have exactly one ordered replication stream to each of its slave shards. As a result, writes to a master shard will be totally ordered in any case and therefore no false dependencies are created due to Occult's causal timestamp design.

Occult uses causal timestamps for (*i*) encoding the most recent state of the data store observed by a client and (*ii*) capturing the set of causal dependencies for write operations. An object version o created by write w is associated with a causal timestamp that encodes all writes in w's *causal history* (i.e., w and all writes that causally preceded it). Upon reading o, a client updates its causal timestamp to the element-wise maximum of its current value and that of o's causal timestamp: the resulting vector defines the earliest state of the datastore that the client is now allowed to read from to respect causal consistency.

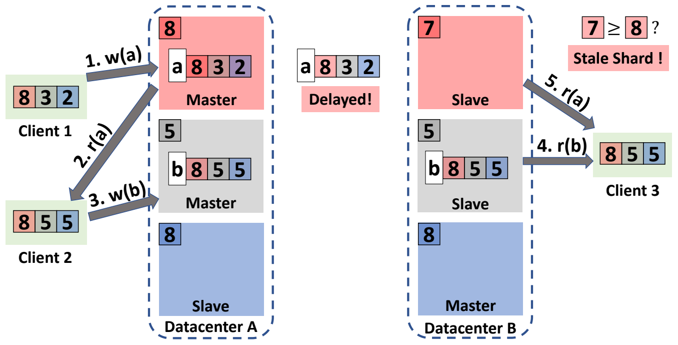
The pseudocode for causal timestamps is listed in Appendix §1.1. Figure 4.1a shows how causal timestamps fit in Occult's system model.



(a) Occult's system model and causal timestamps. 7 writes have been applied to the master of the red shard in Datacenter A and replicated to its slave in Datacenter B. Client 1's causal timestamp knows 4 writes from the red shard.



(b) Client 1 writes to object a, followed by Client 2 reading a and then writing to b. Red and grey shards have incremented shardstamps, due to these new writes. b's causal timestamp is entry-wise $>=$ a's causal timestamp, indicating that write to b is causally ordered after write to a.



(c) b's write replicates to Datacenter B but a's write is delayed. Client 3 still reads the latest value of b. Next, it tries to read from red shard, but the consistency check fails. The shard is stale.

Figure 4.1: Occult Basic Protocol Example.

4.2.3 Basic Protocol

Causal consistency in Occult results from the cooperation between servers and client libraries enabled by causal timestamps. Client libraries use them to validate reads, update them after successful operations, and attach them to writes. Servers store them along with each object, and return one during reads. In addition, servers track the state of each shard using a dedicated shardstamp; when returned in response to a read request, it helps client libraries determine whether completing the read could potentially violate causal consistency.

Next, we will describe the details of this basic protocol. The pseudocode is listed in Appendix §1.2.

Write Protocol Occult associates with any value written v a causal timestamp summarizing all of v's causal dependencies. The client library attaches its causal timestamp to every write and sends it to the master of the corresponding shard. The master increments the relevant shardstamp, updates the received causal timestamp accordingly, and stores it with the newly written value. It then asynchronously replicates the write to its slaves, before returning the shardstamp to the client library. Slaves receive writes from the master in order, along with the associated causal timestamps and shardstamps, and update their state accordingly. On receiving the shardstamp, the client library in turn updates *its* causal timestamp to reflect its current knowledge of the shard's state.

Read Protocol A client reads from its local server, which replies with the desired object's most recent value, that value's dependencies (i.e., its causal timestamp), and the current shardstamp of the appropriate shard. The returned shardstamp s makes checking for consistency straightforward. The client simply compares s with the entry of its own causal timestamp for the shard in question (call it s_c) . If s is at least s_c, then the shard already reflects all the local writes that the client has already observed.

When reading from the master shard, the consistency check is guaranteed to succeed. When reading from a slave, however, the check may fail: replication delays from the master shard in another datacenter may prevent a client from observing its own writes at the slave; or the client may have already observed a write in a different shard that *depends* on an update that has not yet reached the slave.

If the check fails (i.e., the read is *stale*), the client has two choices. It can retry reading from the local replica until the shardstamp advances enough to clear the check. Alternatively, it can send the read to the master shard, which always reflects the most recent state of the shard, at the cost of increased latency and additional load on the master. Occult adopts a hybrid strategy: it retries locally for a maximum of r times (with an exponentially increasing delay between retries) and only then reads from the master replica. This approach resolves most stales quickly, while preventing clients from overloading their local slaves with excessive retries.

Finally, the client updates its causal timestamp to reflect the dependencies included in the causal timestamp returned by the server, ensuring that future successful reads will never be inconsistent with the last read value.

Figure 4.1 shows an example illustrating how the basic protocol works.

4.3 Causal Timestamp Compression

With the basic framework for Occult in place, we now describe the refinements needed to make it practical. An obvious target for refinement are causal timestamps. So far we have assumed a vector with an entry for each shard, but in large-scale systems the number of shards N can be in the hundreds of thousands: the overhead of transferring back and forth and storing with each object causal timestamps of this size would be prohibitive. Occult compresses their size to n entries (with $n \ll N$) without introducing many spurious dependencies.

4.3.1 A First Attempt: Structural Compression

Our most straightforward attempt—*structural compression*—maps all shards whose ids are congruent modulo n to the same entry, reducing a causal timestamps' size from N to n at the cost of generating spurious dependencies [151]. The impact of these dependencies on performance (in the form of delayed reads) worsens when shards have widely different shardstamps. Suppose shards i and j map to the same entry s_c and their shardstamps read, respectively, 100 and 1000. A client that writes to j will fail the consistency check when reading from a slave of i until i has received at least 1000 writes.

In fact, if i never receives 1000 writes, the client will always failover to reading from i's master shard.

These concerns could be mitigated by requiring master shards to periodically advance their shardstamp and then replicate this advancement to their slaves, independent of the write rate from clients. However, fine-tuning the frequency and magnitude of this synchronization is difficult without explicit coordination between i and j. A better solution is instead to rely on *loosely synchronized shardstamps* based on real, rather than logical, clocks [40]. This guarantees that shardstamps differ by no more than the relative offset between their clocks, independent of the write rate on different master shards.

Finally, to reduce the impact of clock skew on creating false dependencies, the master for shard i can use the causal timestamp *ts* received from a client on a write operation to more tightly synchronize its shardstamp with those of other shards that the client has recently accessed. Rather than blindly using the current value *cl* of the physical clock of the server on which it is hosted, i can simply set its shardstamp to be larger than the maximum among (*i*) its current shardstamp; (*ii*) *cl*; and (*iii*) the highest of the values in *ts*.

4.3.2 Temporal Compression

Though using real clocks reduces the chances of generating spurious dependencies, it does not fully address the fundamental limitation of using modulo arithmetic to compress causal timestamps: it is still quite likely that shards with relatively far-apart shardstamps will be mapped to the same entry in the causal timestamp vector.

The next step in our refinement is guided by a simple intuition: recent shardstamps are more likely to generate spurious dependencies than older ones. Thus, rather than mapping a roughly equal number of shards to each of its n entries, *temporal compression* focuses a disproportionate fraction of its ability to accurately resolve dependencies on the shards with the most recent shardstamps. Adapting to our purposes a scheme first devised by Adya and Liskov [40], clients assign an individual entry in their causal timestamp to the $n - 1$ shards with the most recent shardstamps they have observed. Each entry also explicitly stores the corresponding shard id. All other shards are mapped to the vector's "catch-all" last entry. One may reasonably fear that conflating all but $n-1$ shards in the same entry will lead, when a client tries to read from one of the conflated shards, to a large number of failed consistency checks—but it need not be so. For a large-enough n, the catch-all entry will naturally reflect updates that were accepted a while ago. Thus, when a client tries to read from a conflated shard i, it is quite likely that the shardstamp of i will have already exceeded the value stored in the catch-all entry.

To allow causal timestamps to maintain the invariant of explicitly tracking the shards with the $n - 1$ highest observed shardstamps, we must slightly revise the client's read and write protocols in §4.2.3. The first change involves write operations on a shard currently mapped to the catch-all entry. When the client receives back that shard's current shardstamp, it compares it to those of the $n - 1$ shards that its causal timestamp is currently tracking explicitly. The shard with the smallest shardstamp joins the ranks of the conflated and its shardstamp, if it exceeds the current value, becomes the new value of the catch-all entry for the conflated shards. The second change occurs on reads and concerns how the client's causal timestamp is merged with the one returned with the object being read. The shardstamps in either of the two causal timestamps are sorted, and only the shards corresponding to the highest $n-1$ shardstamps are explicitly tracked going forward; the others are conflated, and the new catch-all entry updated to reflect the new dependencies it now includes.

4.3.3 Isolating Datacenters

With either structural or temporal compression, the effectiveness of loosely synchronized timestamps in curbing spurious dependencies can be significantly affected by another factor: the interplay between the time it takes for updates to replicate across datacenters and the relative skew between the datacenters' clocks. Consider two datacenters, A and B, and assume for simplicity a causal timestamp consisting of a single shardstamp. Clocks within each datacenter are closely synchronized and we can ignore their skew. Say, however, that A's clocks run s ms ahead of those in B, that the average replication delay between datacenters is r ms, and that the average interval between consecutive writes at masters is i ms. Assume now that a client c in A writes to a local master node and updates its causal timestamp with the shardstamp it receives. If c then immediately tries to read from a local slave node, c's shardstamp will be ahead of the slave's by about $(s + r + i)$ ms: until the latter catches up, no value read from it will be deemed safe. For clients in B, meanwhile, the window of inconsistency under the same circumstances would be much shorter: just $(-s + r + i)$ ms, potentially leading to substantially fewer stale reads.

This effect can be significant (§4.6.2.1). The master write interval i, even with a read-heavy Zipfian workload, is less than 1 ms in our experiments. However, the replication delay r can range from a few tens to over 100 ms and cross datacenter clock skew s can be tens of milliseconds even when using NTP [18] (clock skew between nodes in the same datacenter is often within 0.5-2ms). Thus, if masters are distributed across datacenters, the percentage of stale reads experienced by clients of different datacenters can differ by orders of magnitude.

We solve this problem using distinct causal timestamps for each datacenter. On writes, clients use the returned shardstamp to update the causal timestamp of the datacenter hosting the relevant master shard. On reads, clients update each of their datacenter-specific causal timestamps using the corresponding causal timestamps returned by the server.

Two factors mitigate the additional overhead caused by datacenter-specific causal timestamps. First, the number of causal timestamps does not grow with the number of datacenters, but rather with the number of datacenters with master shards, which can be significantly lower [63]. Second, because clocks within each datacenter are closely synchronized, these causal timestamps need fewer entries to achieve a given target in the percentage of stale reads.

4.4 Transactions

Many applications can benefit from the ability to read and write multiple objects atomically. To this end, Occult builds on the system described for single-key operations to provide general-purpose read-write transactions. To the best of our knowledge, Occult is the first causal system to support general-purpose transactions while being scalable and resilient to slowdown cascades.

Transactions in Occult run under a new isolation property called Per-Client Snapshot Isolation (PC-PSI), a variant of Parallel Snapshot Isolation (PSI) [146]. PSI is an attractive starting point because it aims to strike a careful balance between the competing concerns of strong guarantees (important for developing applications) and scalable low-latency operations. On the one hand, PSI requires that transactions read from a causally consistent snapshot and precludes concurrent conflicting writes. On the other hand, PSI takes a substantial step towards improving scalability by letting transactions first commit at their local datacenter and subsequently replicate their effects asynchronously to other sites (while preserving causal ordering). In doing so, PSI sidesteps the requirement of a total order on *all* transactions, which is the primary scalability bottleneck of Snapshot Isolation [54] (a popular guarantee in non-distributed systems).

T_1 : s(1) r(x) w(y=10) c(2) T_2 : s(3) r(y=10) w(z) c(4)
T_3 : s(5) r(a) w(b=50) c(6) T_4 : s(7) r(b=50) w(c) c(8)

Figure 4.2: PSI requires transactions to be replicated in commit order. $s(i)$ and $c(j)$ mean respectively start (commit) at timestamp i (j).

PSI's scalability, however, is ultimately undermined by the constraints its implementation imposes on the order in which transactions are to be replicated, leaving it unnecessarily vulnerable to slowdown cascades. Specifically, PSI totally orders all transactions that commit at a replica, and it requires this order to be respected when the transactions are replicated at other sites [146]. For instance, suppose the transactions in Figure 4.2 are executed by four different clients on the same replica. Under PSI, they would be totally ordered as $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$. If, when these transactions are applied at a different replica, any of the shards in charge of applying T_2 is slow, the replication of T_3 and T_4 will be delayed, even though neither has a read/write dependency on T_2.

PC-PSI removes these unnecessary constraints. Rather than totally ordering all transactions that were coincidentally located on the same replica, PC-PSI only requires transactions to be replicated in a way that respects both read/write dependencies and the order of transactions that belong to the same client session (even when the client is not sticky). This is sufficient to ensure semantically relevant dependencies, i.e., if Alice defriends Bob in one transaction and then later posts her Spring-break photos in another transaction, then Bob will not be able to view her photos, regardless of which replica he reads from. At the same time, it allows Occult to support transactions while minimizing its vulnerability to slowdown cascades. Recent work [76] showed that such a "client-centric approach decreased dependencies, per transaction, by two orders of magnitude (175x)" over PSI, while semantically providing the same isolation guarantees.

Like PSI, PC-PSI precludes concurrent conflicting writes. When implementing read-write transactions, this guarantee is crucial to removing the danger of anomalies like lost updates [54]. When writes are accepted at all replicas, as in most existing causally consistent systems [44, 83, 84, 119, 120] this guarantee comes at the cost of expensive synchronization [109], crippling scalability and driving up latency. Not so in Occult, whose master-slave architecture makes it straightforward and inexpensive to enforce, laying the basis for Occult's low-latency read/write transactions.

4.4.1 PC-PSI Specification

To specify PC-PSI, we start from PSI. We leverage recent work [76] that proves PSI is equivalent to *lazy consistency* [40]. This isolation level is known [39] to be the weakest to simultaneously provide two guarantees at the core of PC-PSI: (*i*) transactions observe a consistent snapshot of the database and (*ii*) write-write conflicts are not allowed. We thus build on the theoretical framework behind the specification of lazy consistency [39], adding to it the requirement that transactions in the same client session must be totally ordered.

Concretely, we associate with the execution H of a set of transactions a directed serialization graph $DSG(H)$, whose nodes consist of committed transactions and whose edges mark the conflicts (rw for read-write, ww for write-write, wr for write-read) that occur between them. To these, we add a fourth set of edges: $T_i \xrightarrow{sd} T_j$ if some client c first commits T_i and then T_j (sd is short for *session dependency*). The specification of PC-PSI then constrains the set of valid serialization graphs. In particular, a valid $DSG(H)$ must not exhibit any of the following anomalies:

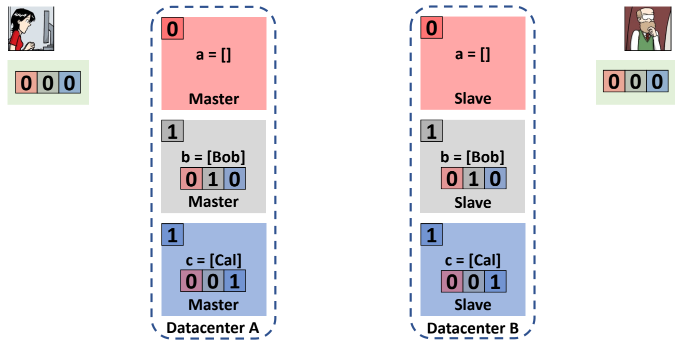
Aborted Reads A committed transaction T_2 reads some object modified by an aborted transaction T_1.

Intermediate Reads A committed transaction T_2 reads a version of an object x written by another transaction T_1 that was not T_1's final modification of x.

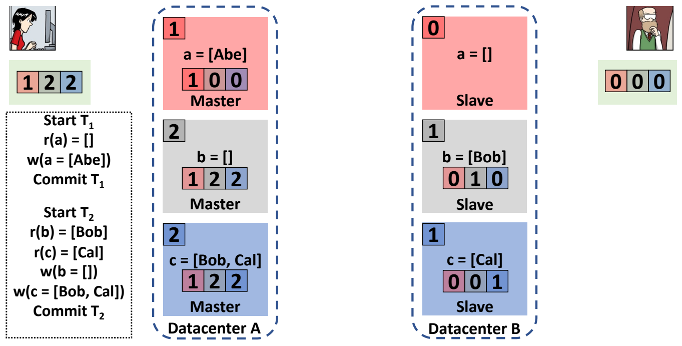
Circular Information Flow $DSG(H)$ contains a cycle consisting entirely of wr, ww and sd edges.

Missed Effects $DSG(H)$ contains a cycle that includes exactly one rw edge.

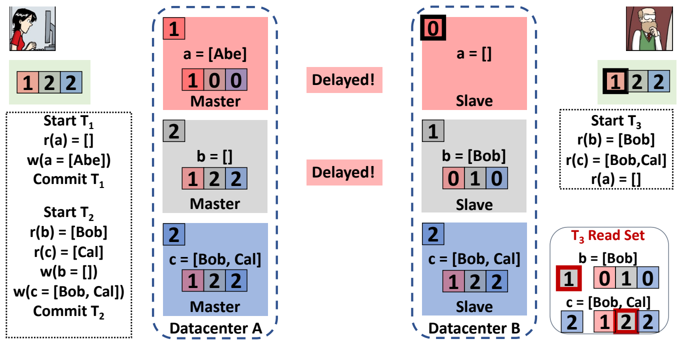
Intuitively, preventing Circular Information Flow ensures that if T_1 and T_2 commit and T_1 depends on T_2, then T_2 cannot depend on T_1. In turn, disallowing cycles with a single rw edge ensures that no committed transaction ever misses writes of another committed transaction on which it otherwise depends, i.e., committed transactions read from a consistent snapshot and write-write conflicts are prevented (§4.4.3).



(a) Alice and her advisor (close to different replicas) are managing three student lists (a, b and c), located on different shards.



(b) Observable Atomicity through Causality. Alice adds Abe to list a in transaction T_1 and then moves Bob from list b to c in transaction T_2. Thus $T_1 \xrightarrow{sd} T_2$. At T_2's commit, b's causal timestamp knows about the write to c and vice versa.



(c) Replication of writes to a and b is delayed. Alice's advisor reads all three lists in transaction T_3. Read set validation of T_3 fails due to atomicity violation: c's causal timestamp knows more writes from the grey shard than were applied to that shard at the time b was read (thick red border entries). $T_1 \xrightarrow{sd} T_2$ ordering violation due to stale red shard (thick black border entries) is also detected.

Figure 4.3: Occult Transactional Protocol Example.

4.4.2 Executing Read/Write Transactions

Occult supports read/write transactions via a three-phase optimistic concurrency protocol that, in line with the system's ethos, makes clients responsible for running the logic needed to enforce PC-PSI. First, in the *read phase*, a client c executing transaction T obtains from the appropriate shards the objects that T reads, and locally buffers T's writes. Then, in the *validation phase*, c ensures that all read objects belong to a consistent snapshot of the system that reflects the effects of all transactions that causally precede T. Finally, in the *commit phase*, c writes back atomically all objects updated by T.

Next, we will describe the details of this transactional protocol. The pseudocode is listed in Appendix §1.3.

Read phase For each object o read by T, c contacts the local server for the corresponding shard s_o, making sure, if the server is a slave, not to be reading a stale version (§4.2.3) of o—i.e., a version of o that is older than what c's causal timestamp already reflects about the state of s_o. If the read is successful, c adds o, its causal timestamp, and s_o's shardstamp to T's *read set*. Otherwise, after a tunable number of further attempts, c proceeds to read o from its master server, whose version is never stale. Meanwhile, all writes are buffered in T's *write set*. They are atomically committed to servers in the final phase. Thus only committed objects are read in this phase and cascading aborts are not possible.

Validation phase Validation involves three steps. In the first, c verifies that the objects in its read set belong to a consistent snapshot Σ_{rs}. It does so by checking that all pairs o_i and o_j of such objects are *pairwise consistent* [48], i.e., that the saved shardstamp of the shard s_{o_i} from which o_i was read is at least as up to date as the entry for s_{o_i} in the causal timestamp of o_j (and vice versa). If the check fails, T aborts.

In the second step, c attempts to lock every object o updated by a write w in T's write set by contacting the corresponding shard s_o on the master server. If c succeeds, Occult's master-slave design ensures that c has exclusive write access to the latest version of o (reads are always allowed); if not, c restarts this step of the validation phase until it succeeds (or possibly aborts T after n failed attempts). In response to a successful lock request, the master server returns two data items: 1) o's causal timestamp, and 2) the new shardstamp that will be assigned to w. c stores this information in T's *overwrite set*. Note that, since they have been obtained from the corresponding master servers, the causal timestamps of the objects in the overwrite set are guaranteed to be pairwise consistent, and therefore to define a consistent snapshot Σ_{ow}: Σ_{ow} captures the updates of all transactions that T would depend on after committing.

To ensure that T is not missing any of these updates, in the final step of validation c checks that Σ_{rs} is at least as recent as Σ_{ow}. If the check fails, T aborts.

Commit phase c computes T's *commit timestamp* ts_T by first initializing it to the causal timestamp of the snapshot Σ_{rs} from which T read, and by then updating it to account for the shardstamps, saved in T's overwrite set, assigned to T's writes. The value of $ts_T[i]$ is thus set to the largest between (i) the highest value of the i-th entry of any of the causal timestamps in T's read set, and (ii) the highest shardstamp assigned to any of the writes in T's write set that update an object stored on a shard mapped to entry i. c then writes back the objects in T's write set to the appropriate master server, with ts_T as their causal timestamp. Finally, to ensure that any future transaction executed by this client will be (causally) ordered after T, c sets its own causal timestamp to ts_T.

The commit phase enforces a property that is crucial for Occult's scalability: it guarantees that transactions are atomic even though Occult replicates their writes asynchronously. Because the commit timestamp ts_T both reflects all writes that T performs and is used as the causal timestamp of every object that T updates, ts_T makes all of these updates, in effect, causally dependent on one another. As a result, any transaction whose read set includes any object o in T's write set will necessarily either become dependent on all the updates that T performed, or none of them.

Figure 4.3 shows an example illustrating the protocol.

4.4.3 Correctness

To implement PC-PSI, the protocol must prevent Aborted Reads, Intermediate Reads, Circular Information Flow, and Missed Effects. The optimistic nature of the protocol trivially yields the first two conditions, as writes are buffered locally and only written back when transactions commit. Occult also precludes Circular Information Flow. Since clients acquire write locks on all objects before modifying them, transactions that modify the same objects cannot commit concurrently and interleave their writes (no *ww* cycles). Cycles consisting only of *ww*, *wr*, and *sd* edges are instead prevented by the structure of OCC, whose read phase strictly precedes all writes: if a sequence of $ww/wr/sd$ edges leads from T_1 to T_2, then T_1 must have committed before T_2, and could not have observed the effects of T_2 or created a write with a lower causal timestamp than T_2's.

Finally, Occult's validation phase prevents Missed Effects. By contradiction, suppose that all transactions involved in a DSG cycle with a single anti-dependency (*rw*) edge have passed the validation phase. Let T be the transaction from which that edge originates, ending in T^*. Let T_{-1} immediately precede T in the cycle. Let o be the object written by T^* whose update T missed. Either T_{-1} and T^* are one, or T_{-1} $wr/ww/sd$ depends on T^*: either way, Occult's protocol ensures that the commit timestamp of T_{-1} is at least as large as that of T^*. By assumption, T missed some update to o: hence, the shardstamp for o's shard s_o in T's readset must be smaller that the corresponding entry in the commit timestamps of T^* and T_{-1}. There are three cases:

(*i*) $T_{-1} \xrightarrow{sd} T$. The client that issued both T_{-1} and T must have decreased its causal timestamp after committing T_{-1}, but the protocol ensures causal timestamps increase monotonically.

(*ii*) $T_{-1} \xrightarrow{wr} T$. Since T reads an object updated by T_{-1}, its read set contains T_{-1}'s commit timestamp. But then T would fail in validating its read set, since the object updated by T_{-1} and the version of o read by T would be pairwise inconsistent.

(*iii*) $T_{-1} \xrightarrow{ww} T$. Since T overwrites an object updated by T_{-1}, T's overwrite set must include T_{-1}'s commit timestamp. But then T would fail in validating its read set against its overwrite set, since the latter has a larger entry corresponding to s_o than the former.

Each case leads to a contradiction: hence no such cycle can occur and no effects are missed.

4.4.3.1 Missed Effects Prevention Examples

As shown by the correctness argument above, Occult's sophisticated validation phase is designed to prevent Missed Effects anomalies. To further understand why this works, consider the following examples.

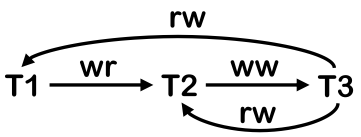
$$\begin{tikzcd}
\mathbf{T1} \arrow[r, "\mathbf{wr}"] & \mathbf{T2} \arrow[r, "\mathbf{wr}"] & \mathbf{T3} \arrow[ll, bend right, "\mathbf{rw}"']
\end{tikzcd}$$

$$\mathbf{r_3(x=0)\ r_1(x=0)\ w_1(x=1)\ c_1\ r_2(x=1)\ r_2(y=0)\ w_2(y=1)\ c_2\ r_3(y=1)\ c_3}$$

Figure 4.4: Example of inconsistent reads, prevented by read set validation.

First, lets see how validating the read set prevents Missed Effects anomalies involving *inconsistent reads* (pertaining to case (*ii*) of the correctness argument). Consider the example in Figure 4.4. Assume the objects x

and y are located on different shards, s_x and s_y respectively. Here, T_3 misses the write to x by T_1, but reads the newer version of y, that was written by T_2 *after* it had read T_1's write to x. The set of reads done by T_3 are therefore inconsistent. When T_3 attempts to commit, Occult's read set validation will detect that y's causal timestamp has a higher shardstamp for s_x (because of T_1's write to x) than the saved shardstamp for s_x at the time x was read by T_3. This indicates that the version of x, read by T_3, is potentially stale compared to the version of y read by it. As a result, Occult will abort T_3, thus preventing the anomaly.



$$\begin{tikzcd}
\mathrm{T1} \arrow[r, "\mathrm{wr}"] & \mathrm{T2} \arrow[r, "\mathrm{ww}"] & \mathrm{T3} \arrow[ll, bend right, "\mathrm{rw}"'] \arrow[l, bend left, "\mathrm{rw}"]
\end{tikzcd}$$

$r_3(x=0)\ r_3(y=0)\ r_1(x=0)\ w_1(x=1)\ c_1\ r_2(x=1)\ r_2(y=0)\ w_2(y=1)\ c_2\ w_3(y=2)\ c_3$

Figure 4.5: Example of concurrent conflicting writes, prevented by overwrite set validation.

Next, lets see how validating the overwrite set prevents Missed Effects anomalies involving *concurrent conflicting writes* (pertaining to case (*iii*) of the correctness argument). Consider the example in Figure 4.5. Here, merely validating T_3's read set is not sufficient since it is consistent. However, T_3 overwrites the newer version of y (written by T_2) that it missed. Thus, T_2's update to y is *lost* [54]. When validating T_3's overwrite set, Occult will detect that the version of y in T_3's overwrite set (from T_2's write to y) has a higher shardstamp for s_y, than the saved shardstamp for s_y at the time y was read. This indicates that T_3 is potentially overwriting a version of y that it missed during its read phase. Once again T_3 will abort, preventing the anomaly.

4.5 Fault Tolerance

4.5.1 Server Failures

Slave failures in Occult only increase read latency as slaves never accept writes and read requests to failed slaves eventually time-out and redirect to the master. Master failures are more critical. First, as in all single-master systems [149], no writes can be processed on a shard with a failed master. Second, in common with all asynchronously replicated systems [46, 53, 119, 120, 149], Occult exhibits a vulnerability window during which writes executed at the master may not yet have been replicated to slaves and may be lost if the master crashes. These missing writes may cause subsequent client requests to fail: if a client c's write to object o is lost, c cannot read o without violating causality. This scenario is common to all causal systems for which clients do not share fate with the servers to which they write. Occult's client-centric approach to causal consistency, however, creates another dangerous scenario: as datacenters are not themselves causally consistent, writes can be replicated out of order. A write y that is dependent on a write x can be replicated to another datacenter despite the loss of x, preventing any subsequent client from reading both x and y.

Occult's current prototype (built by modifying Redis Cluster) can prevent such a loss of read availability by leveraging Redis's existing failure recovery logic for slaves in which data is transferred asynchronously from the master shard to a new slave. Extending our prototype to support slave failures is relatively straightforward: data can simply be transferred asynchronously from the master shard to the new slave [148]. Any client request will, as per the traditional read logic, reroute to the master if the slave is not yet up-to-date.

Likewise, Master failures can also be handled using well-known techniques: individual machine failures within a datacenter can be handled by replicating the master locally using chain-replication [154] or Paxos [110], before replicating asynchronously to other replicas. This process can be abstracted within each write to the master, and does not not require any modifications to the Occult client's read/write protocol.

4.5.2 Client Failures

A client failure for single-key operations impacts only the failed client as neither reads nor writes create temporary server state. In transactional mode, however, clients modify server state during the commit phase: they acquire locks on objects in the transaction's write-set and write back new values. A client failure during the transaction commit process may thus cause locks to be held indefinitely by failed clients, preventing other transactions from committing. Such failures can be handled by augmenting Occult with Bernstein's cooperative termination protocol [55] for coordinator recovery [105, 166]. Upon detecting a suspected client failure, individual shards can attempt to elect themselves as backup coordinator (using an instance of Paxos to ensure that a single coordinator is elected). The backup coordinator can then appropriately terminate the transaction (by committing it if a replica shard successfully received an unlock request with the appropriate transaction timestamp using the buffered writes at every replica, or aborting it otherwise).

4.6 Evaluation

The evaluation answers three questions:

1. How well does Occult perform in terms of throughput, latency, and transaction abort rate?
2. What is its overhead when compared to an eventually-consistent system?
3. What is the effect of server slowdowns on Occult?

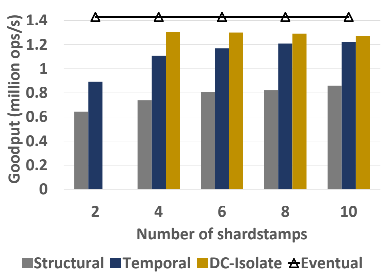
We implemented Occult by modifying Redis Cluster [20], the distributed implementation of the widely-used Redis key-value store. Redis Cluster divides the entire key-space into N logical shards (default $N = 16$K), which are then evenly distributed across the available physical servers. Our causal timestamps track shardstamps at the granularity of logical shards to avoid dependencies on the physical location of the data.

For a fair comparison with Occult, we modify our Redis Cluster baseline to allow reads from slaves (Redis Cluster by default uses primary-backup [141] replication for fault tolerance). We further modify the Redis client [10] to, like Occult, allow for client locality: the client prioritizes reading from shards in its local datacenter and executes write operations at the master shard.

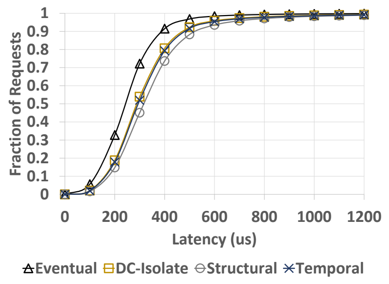
4.6.1 Experimental Setup

We run our experiments on CloudLab [2, 140] with 20 server and 20 client machines evenly divided across two datacenters in Wisconsin (WI) and South Carolina (SC); the cross-datacenter ping latency is 39ms. Each machine has dual Intel E5-2660 10-core CPUs and dual-port Intel 10Gbe NICs, with respectively 160GB memory (WI) and 256GB (SC). Our experiments use public IP addresses, routable between CloudLab sites, which are limited to 1Gbps. Each server machine runs four instances of the server process, with each server process being responsible for $N/40$ logical shards. Half of all shards have a master in WI and a slave in SC; the other half have the opposite configuration.

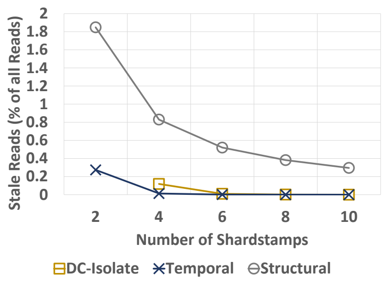
Client machines run the Yahoo! Cloud Serving Benchmark (YCSB) [73]. We run experiments with both of YCSB's Zipfian and Uniform workloads but, for brevity, show results only for the Zipfian distribution, more representative of real workloads. Prior to the experiments, we load the cluster with 10 million records following YCSB's default, i.e., keys varying in size up to 23B and 1KB values. We report results at peak goodput, running for at least 100 seconds and then excluding 10-second ramp-up and ramp-down periods. Goodput measures successful operations per second, e.g., a read that needs to be retried four times will only be counted once towards goodput. The bottleneck for all experiments is out bound network bandwidth on the hottest master. Even if we were to increase network bandwidth, each system would bottleneck at a similar throughput: the CPU on the hottest master is nearly saturated ($> 90\%$ utilization).



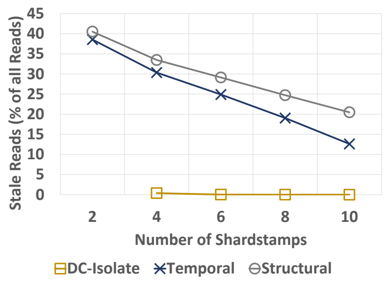
(a) Goodput (Read-heavy)



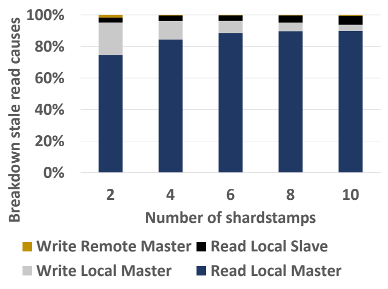
(b) Read Latency CDF (Read-heavy)



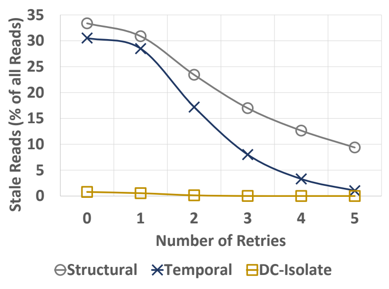
(c) Stale Reads WI (Read-heavy)



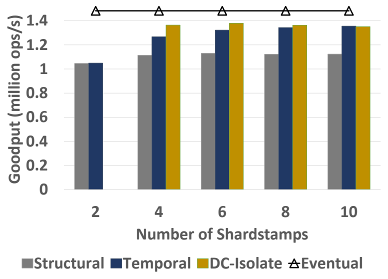
(d) Stale Reads SC (Read-heavy)



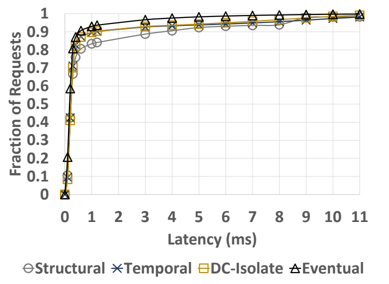
(e) Stale Read Analysis (Read-heavy)



(f) Retrying Stale reads (Read-heavy)



(g) Goodput (Write-heavy)



(h) Read Latency CDF (Write-heavy)

Figure 4.6: Measurement and analysis of Occult 's overhead for single key operations. Spatial, Temporal or DC-Isolate mean that we run Occult using those compression methods while Eventual indicates our baseline, i.e., Redis Cluster. WI means Wisconsin datacenter and SC means South Carolina datacenter.

4.6.2 Performance and Overhead

4.6.2.1 Single Key Operations

We first quantify the overhead of enforcing causal consistency in Occult. We first show results for a read-heavy (95% reads, 5% writes) workload, since it is more interesting and challenging for our system. Later we show results for a write-heavy (75% reads, 25% writes) workload as well. The write-heavy workload performed better in general, as could be expected from the system design. We include it for completeness.

We compare system throughput as a function of causal timestamp size, for each of the previously described schemes (structural, temporal, and temporal with datacenter isolation), with Redis cluster as the baseline. Temporal compression requires a minimum of two entries per causal timestamp; adding datacenter isolation (DC-Isolate), doubles this number, so that the smallest number of shardstamps used by DC-Isolate is four.

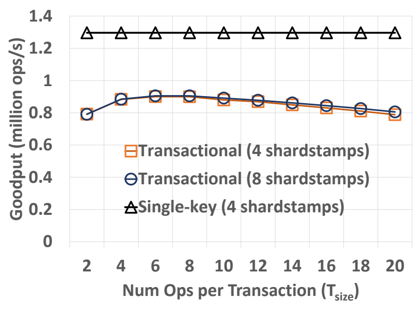
Read-heavy workload In the best case for this workload (using the DC-Isolate scheme with four-entry timestamps), Occult's performance is competitive with Redis, despite providing much stronger guarantees: its goodput is only 8.7% lower than Redis (Figure 4.6a) and its mean and tail latency are, respectively, only 50μs and 400μs higher than in Redis (Figure 4.6b). Other schemes perform either systematically worse (Structural), or require twice the number of shardstamps to achieve comparable performance (Temporal). The low performance of the structural and temporal schemes are due to their high stale read rate (Figures 4.6c and 4.6d). In contrast, DC-Isolate has very a low percentage of stale reads even with small causal timestamps. Its slight drop in goodput is primarily due to Occult's other source of overhead: the CPU, network, and storage cost of attaching and storing timestamps to requests and objects. These results highlight the tension between overhead and precision: larger causal timestamps reduce the amount of stale reads (as evidenced by the improved performance of the temporal scheme when vector size grows), but worsen overhead (the goodput of the DC-Isolate scheme actually drops slightly as the number of shardstamps increases).

Achieving a low stale read rate with few shardstamps, as DC-Isolate does, is thus crucial to achieving good performance. Key to its success is its ability to track timestamps from different datacenters independently. Consider Figures 4.6c and 4.6d: in these experiments we simply count the percentage of stale reads but do not retry locally or read from the remote master. Observe that the temporal and structural schemes suffer from a significantly higher stale read rate in the SC datacenter. To understand why, we instrumented the code to track metadata related to the last operation to modify a client's causal timestamp before it does a stale read. We discovered that almost 96% of stale reads occur when the client writes or reads from a local master node immediately before reading from a local slave node (Figure 4.6e). If the local master node runs ahead (for instance, the SC datacenter has a positive offset of about 22 ms, as measured via *ntpdate*), the temporal scheme will declare all reads to the local slave as stale. In contrast, by tracking dependencies on a per-datacenter basis, DC-Isolate side-steps this issue, producing a low stale rate across datacenters.

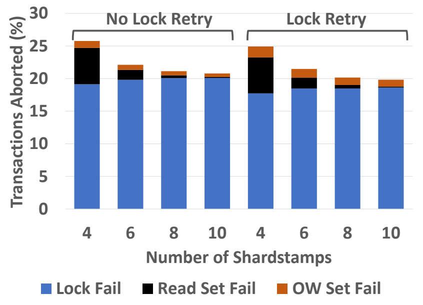
Write-heavy workload Figures 4.6g and 4.6h show evaluation of Occult on a write-heavy workload (75% reads, 25% writes) with a Zipfian distribution of operations. Overall Occult suffers less goodput overhead (6.9%) over Redis on this workload than the read-heavy workload. The median latency increase over Redis is still 50μs but tail latency increases by 4ms.

4.6.2.2 Transactions

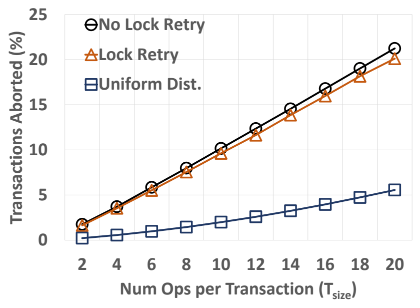
To evaluate transactions, we modify the workload generator of the YCSB benchmark to issue *start* and *commit* operations in addition to reads and writes. Operations are dispatched serially, i.e., operation i must complete before operation $i + 1$ is issued. The resulting long duration of transactions are worst case scenario for Occult. The generator is parameterized with the required number of operations per transaction (T_{size}). We use the DC-Isolate scheme for Occult and the read-heavy workload of YCSB in all these experiments.



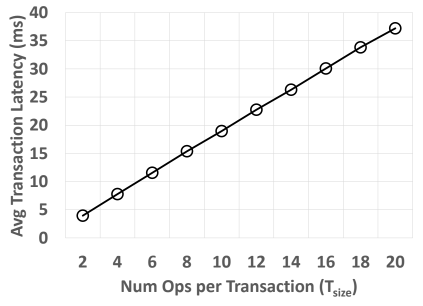
(a) Goodput



(b) Abort Rate as $f(shardstamps)$ ($T_{size} = 20$)



(c) Abort Rate as $f(T_{size})$ ($shardstamps = 8$)



(d) Average Latency

Figure 4.7: Transactions in Occult

We show results for increasing values of T_{size}. For smaller values, most transactions in the workload are read-only, and as T_{size} increases most transactions become read-write. As Figure 4.7a shows, the overall goodput remains within 2/3 of the goodput of non transactional Occult (varying from 60% to 70%), even as T_{size} increases and aborts become more likely. Figures 4.7b and 4.7c analyze the causes of these aborts. Recall from §4.4.2 that aborts can occur because of either (i) validation failures of the read/overwrite sets or (ii) failure to acquire locks on keys being written.

Figure 4.7b fixes $T_{size} = 20$ and classifies aborts into these three categories. We find that aborts are dominated by the failure to acquire locks. Furthermore, due to the highly skewed nature of the YCSB zipfian workload, >80% of these lock-fail aborts are due to contention on the 50 hottest keys. This high contention also explains the limited benefit of retrying to acquire locks. Figure 4.7b also shows that increasing the number of shardstamps almost completely eliminates aborts due to failed validations of the read set and roughly halves aborts due to failed validations of the overwrite set. Figure 4.7c shows that the abort rate increases linearly with increasing T_{size} and that retrying lock acquisition has slightly better impact at larger values of T_{size} when most transactions are read-write. For comparison, we show the abort rate on a uniform distribution. Finally, Figure 4.7d shows the average commit latency of transactions from *start* to *commit* as a function of T_{size}. The linear rise in latency is because operations in our workload are dispatched serially.

4.6.2.3 Resource Overhead

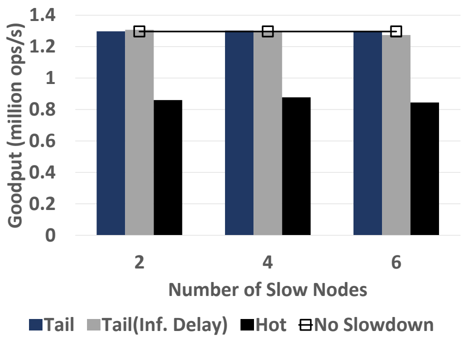
To quantify the resource overhead of Occult over Redis Cluster, we measure the CPU usage (using *getrusage*()) and the total bytes sent and received over 120 secs for both systems at the same throughput (1.27Mop/s) and report the average of five runs, averaged over the 80 server processes.

Overall CPU usage increases by 7% with a slightly higher increase on slaves (8%) than masters (6%). This difference is due to stale read retries in Occult. Output bandwidth increases by 8.8%, while input bandwidth increases by 49%, as attaching metadata to read requests with a key size of at most 23B has a much larger impact than attaching it to replies carrying 1KB values.

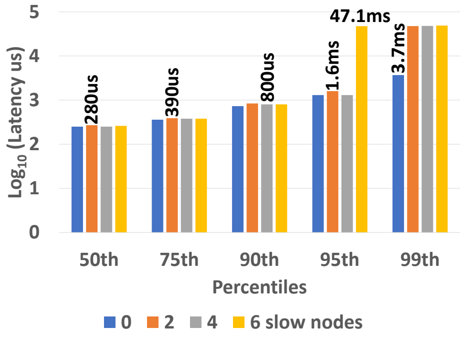
Finally, we measure storage overhead by loading both Redis and Occult with 10 million records and measuring the increase in memory usage of each server process. Storing four shardstamps with each key results in an increase, on average, of 3% for Occult over Redis. Storing 10 shardstamps instead results in an increase of 4.9%.

4.6.3 Impact of Slow Nodes

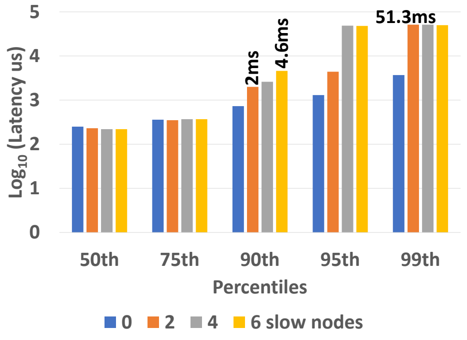
Occult is by design immune to the slowdown of a server cascading to affect the entire system. Nonetheless, the slowdown of a server does introduce additional overhead relative to an eventually-consistent system. In particular, slowing down slaves increases the stale rate, which in turn increases retries on that slave and remote reads from its corresponding master. We measure these effects by artificially slowing down the replication of writes at a number of slave nodes, symmetrically increasing their number from one to three per datacenter—two to six overall. This causes a slowdown of around 2.5% to 7.5% of all nodes. The workload for this experiment was read-heavy with a Zipfian distribution of operations.



(a) Goodput



(b) Latency(slow tail nodes)



(c) Latency(slow hot nodes)

Figure 4.8: Effect on overall goodput and read latency due to slow nodes in Occult

We notice that, at peak throughput, the node containing the hottest key serves around 3× more operations than the nodes serving keys in the tail of the distribution. We evaluate slowdowns of the tail nodes separately from the hot nodes, which we slowdown in decreasing order of load, starting from the hottest node.

We first delay replicated writes on tail nodes by 100 ms, which, as Figure 4.8a shows, does not affect throughput: even at peak throughput for the cluster, only the hottest nodes are actually CPU or network saturated. As such, tail nodes (master or slave) still have spare capacity. When clients failover to the master (after n local retries), this spare capacity absorbs the additional load. In contrast, read latency is affected (Fig 4.8b). Though median, 75th, and 90th percentile latencies remain unchanged because reads to non-slow nodes are unaffected by the presence of slow servers, tail latencies increase significantly as the likelihood of hitting a lagging server and reading a stale value increases. Thus, increasing slow nodes from two to six first makes the 99th percentile and then the 95th percentile latency jump to around 48ms. This includes $n = 4$ local retries by the client (after delays of 0, 1, 2, and 4 ms) and finally contacting the master in a remote datacenter (39 ms away). Having a large delay of 100 ms and $n = 4$ means that our experiment actually evaluates an arbitrarily large slowdown, since almost all client reads to slow slaves eventually fail over to the master. We confirm this by setting the delay to infinite: the results for both throughput (Figure 4.8a) and latency (not shown) are identical to the 100 ms case.

Slowing down the hot nodes impacts both throughput and latency. The YCSB workload we use completely saturates the hottest master and its slave. Unlike in the previous experiments, the hot master does not have any spare capacity to handle failovers, and throughput suffers (Figure 4.8a). Slowing more than two slave nodes does not decrease throughput further because their respective masters have spare capacity. Figure 4.8c shows that, as expected given the skewed workload, slowing down an increasing number of hot nodes increases the 99th and 95th percentile latencies faster than slowing down tail nodes (Figure 4.8b). The median and 75th percentile latencies remain unchanged as before.

4.7 Related Work

4.7.1 Scalable Causal Consistency

COPS [119] tracks causal consistency with explicit dependencies and then enforces it pessimistically by checking these dependencies before applying remote writes at a replica. COPS strives to limit the loss of throughput caused by the metadata and messages needed to check dependencies by exploiting transitivity. ChainReaction [44], Orbe [83], and GentleRain [84] show how to reduce these dependencies further by using Bloom filters, dependency matrices, and a single timestamp, respectively. These techniques reduce metadata by making it more coarse-grained, which actually exacerbates slowdown cascades. Eiger [120] builds on COPS with a more general data model, write-only transactions, and an improved read-only transaction algorithm. BoltOn [53] shows how to use shim layers to add pessimistic causal consistency to an existing eventually consistent storage system. COPS-SNOW [121] provides a new latency-optimal read-only transaction algorithm. Occult improves on this line of research by identifying the problem of slowdown cascades and showing how an optimistic approach to causal consistency can overcome them. In addition, all of these systems provide weaker forms of transactions than Occult: Eiger provides read-only and write-only transactions, while all other systems provide only read-only transactions or no transactions at all.

Pileus [149] and Tuba [46] (which adds reconfigurability to Pileus) provide a range of consistency models that clients can dynamically choose between by specifying an SLA that assigns utilities to different combinations of consistency and latency. Pileus has several design choices that are similar to Occult: it uses a single master, applies writes at replicas without delay (i.e., is optimistic), uses a timestamp to determine if a read value meets a given consistency level (including causal consistency), and can issue reads across different datacenters to meet a given consistency level. However, Pileus is not scalable as it uses a single logical timestamp as the client's state (which we show in our evaluation has a very high false positive stale rate) and evaluates with only a single node per replica. We consider an interesting avenue of future work to see if we can combine the focus of Pileus (consistency choice and SLAs) with Occult.

Cure [43] is a causally consistent storage system that provides read-write transactions. Cure is pessimistic and uses a single timestamp per replica to track and enforce causal dependencies. Cure provides a restricted form of read-write transactions that requires all operations to be on convergent and commutative replicated data types (CRDTs) [143]. Using CRDTs allows Cure to avoid coordination for writes and instead eventually merges conflicting writes, including those issued as part of read-write transactions. Occult, in contrast, is an optimistic system that provides read-write transactions for the normal data types that programmers are familiar with. Saturn [61], like Occult, tries to strike a balance between metadata overhead and false sharing by relying on "small labels" (like Cure) while selecting serializations at datacenters that minimize spurious dependencies.

4.7.2 Read/Write Transactions

Many recent research systems with read/write transactions are limited to a single datacenter (e.g., [112, 130, 158, 161]) whereas most production systems are geo-replicated. Some geo-replicated research systems cannot scale to large clusters because they have a single point of serialization per datacenter [77, 146] while others are limited to transactions with known read and write sets [131, 150, 167].

Scalable geo-replicated transactional systems include Spanner [74], MDCC [105] and TAPIR [166]. Spanner is a production system at Google that uses synchronized clocks to reduce coordination for strictly serializable transactions. MDCC uses Generalized Paxos [110] to reduce wide-area commit latency. TAPIR avoids coordination in both replication and concurrency control to be able to sometimes commit a transaction in a single wide-area round trip. All of these systems provide strict serializability, a much stronger consistency level than what Occult provides. As a result, they require heavier-weight mechanisms for deciding to abort or commit transactions and will abort more often.

4.7.3 Rethinking the Output Commit Step

We were inspired to rethink the output commit step for causal consistency by a number of previous systems: Rethink the Sync [133], which did it for local file I/O; Blizzard [127], which did it for cloud storage; Zyzzyva [104], which did it for Byzantine fault tolerance; and Speculative Paxos [138], which did it for Paxos.

4.8 Limitations

The limitations of Occult's design include: its master-slave architecture, the blocking nature of its read-write transactions and a vulnerability to buggy clients due to its client-centric approach.

Occult's master-slave architecture has performance and latency disadvantages, compared to prior causal data-stores (e.g., [84, 119, 120]), which had multi-master designs. A hot master shard can become the bottleneck for a skewed workload, and in a geo-replicated setup, writing to the master can incur significant latency for operations by distant clients. We explained the reasons for this choice in §4.2.1, including the fact that this is the design choice of some of the largest real-world systems [63, 71, 134, 139]. An additional benefit, is that it allowed us to build scalable read-write transactions with stronger correctness guarantees than previous causal systems. Having all replicas accept writes, with asynchronous replication, restricts a system from providing stronger guarantees (such as a variant of snapshot isolation) for its read-write transactions, because of concurrent conflicting writes and lost updates. This can be a problem for real-world systems [139], which cited having a primary source of truth as the reason for their choice of a master-slave architecture.

Occult's read-write transactions also have the limitation of requiring locks during the validation phase, in contrast to previous systems [120], which provided non-blocking write-only transactions. Blocking can cause a performance hit for workloads with high true contention, such as the heavily skewed YCSB transactional workload in our evaluation (§4.6.2.2). Yet, the utility of just providing write-only and read-only transactions is limited. It remains an interesting avenue for future work, on how to provide read-write transactions in a causal datastore with a stronger isolation level like PC-PSI, while eliminating blocking for a significant subset of transactions involving writes.

Finally, Occult's client-centric design is vulnerable to buggy clients. Such a client could, e.g., claim to know all writes done until the year 2100 at shard 3. If this client writes to an object with its causal timestamp, then later clients now have to wait for shardstamps of slaves of shard 3, to catch up to the year 2100, before being able to read. There are a couple of mitigating factors for this limitation. First, most large-scale datastores do not directly expose their storage system's API to public clients, for security reasons. For these systems, the same engineering team that builds the storage system, also maintains any internal clients, which reduces the chances of such errors. Pushing the enforcement of consistency to the client, for these systems, means internal clients, not web browsers. Second, simple sanity checking can greatly mitigate these sorts of bugs. For instance, each master server can estimate bounds on the clock values of other master servers (e.g., worst case clock drift multiplied by 2) and reject a write if it is higher than those bounds.

Chapter 5

ScaleDB: An Asynchronous In-Memory Database

ScaleDB is a serializable transactional database designed for scalability. The discussion and analysis in §2.2.3 showed that shared range-index structures continue to be a main source of mechanism coordination, and the high cost of updates to these indexes, even by unrelated transactions, is a major factor limiting scalability. Further, the analysis in §3.1.1 showed that contention caused by synchronous updates to sorted range-index structures is unnecessary in the common case; it is possible to delay many common range-index updates, without compromising on strong consistency guarantees or latency requirements for transactions. The foundation of ScaleDB's design, building on that analysis, is that range indexes are *asynchronously* updated to provide scalability.

ScaleDB's design, minimizes unnecessary contention among unrelated transactions by decoupling the commit of a transaction from the update to the affected range indexes; we update range indexes asynchronously (in batches), while using scalable hash-based *indexlets* to track writes of recently committed transactions. By decoupling transaction execution from range index updates, ScaleDB can focus on improving the scalability of the former in isolation from the latter and without undesirable performance tradeoffs.

Based on this asynchronous architecture at the core of ScaleDB, we design *Asynchronous Concurrency Control* (ACC), a novel concurrency control protocol that provides serializability for concurrent transactions without compromising scalability, commit latency, or throughput. ACC is an optimistic concurrency control protocol that builds on indexlets to provide *phantomlets* for scalable phantom detection [54] and uses transactional locks in indexlets, rather than in range indexes, to provide scalable atomic transaction commit. By avoiding unnecessary contention on shared data structures in the common case, ScaleDB uses ACC to guarantee scalable serializable isolation for ACID transactions, with high throughput, low commit latency, and low abort rate.

5.1 Design Rationale and Overview

ScaleDB's main contribution lies in recognizing that removing the indexing bottleneck requires to look beyond range index structures; instead, it is necessary to understand and correct the architectural design decisions that make range indexes a hotspot of contention in today's in-memory databases. This follows from the findings in §3.1.1 which open up an opportunity to fundamentally rethink how to maintain range indexes within in-memory databases. If, in the common case, synchronous updates to range indexes are not necessary to produce consistent range scans, it may be possible to design new scalable data structures that can synchronously store record updates and hold them temporarily, until they are asynchronously applied to the range indexes. Of course, range scans should be *always* consistent, not just in the common case, and the mechanisms needed to enforce this guarantee should themselves be scalable. These are the opportunities and challenges that shape the design of ScaleDB.

Why are asynchronous range index updates scalable? Asynchronously updating range indexes offers a host of opportunities that we seek to exploit. Accumulating a number of updates, so they can be applied as a batch to the range index, is more efficient than applying individual updates, as it avoids repeated walks of the index tree (e.g. inserts to the same B+ tree leaf node). Given the cache contention arising from concurrent walks of the range index, batched updates benefit CPU cache locality and improve performance isolation among CPU cores. They also incur less overhead for repeated lock operations, since they allow us to acquire locks only once for several updates. We can facilitate this process by sorting accumulated updates before applying them to the range index, outside of a critical section. Finally, for skewed access distributions that update the same record repeatedly within a short time span, only the last update in the batch needs to be applied to the range index, reducing the overall work required.

The basic idea of updating range indexes asynchronously raises some questions. How can this asynchronous architecture provide scalable transaction processing? And how can serializable isolation be guaranteed when range indexes are no longer kept synchronously consistent? Next, we answer these questions.

5.1.1 Scalable Transaction Processing with Indexlets

To asynchronously update range indexes, we need a temporary store for writes that can be scalably maintained and flushed with minimal overhead. To tackle this problem, we introduce a new data structure: hash-based *indexlets*. Indexlets temporarily and synchronously record all range index writes. Hash indexes have a flat structure. As a result, they can avoid contention on their internal structure for updates to unrelated records. The exception is *rehashing*—resizing the hash index when it is at capacity [21]. Database hash indexes require rehashing, as their size cannot be known a-priori. Instead, indexlets only temporarily hold updates and are periodically merged by ScaleDB into range indexes. Thus, we can avoid rehashing by bounding the maximum number of delayed writes held in an indexlet based on the W-to-RS latency and expected write rate to the underlying table. We describe indexlets and how to efficiently size and scalably merge them in §5.2.1.

5.1.2 Serializability with Asynchronous Range Index Updates

We design *asynchronous concurrency control* (ACC), a concurrency control protocol that provides serializability in an asynchronous database architecure. ACC is based on optimistic concurrency control (OCC) [107, 152], which it integrates with asynchronous range index updates. Both approaches are optimistic. Just as OCC assumes that most transactions do not contend, asynchronous range index updates assume that most W-to-RS latencies allow us to leave range indexes temporarily stale without performance consequences.

Since recent writes are held in indexlets, asynchronously enforcing serializability with good performance requires first checking indexlets on any point read, and, for range scans, efficiently detecting the small number of instances when a scan has accessed a stale portion of a range index. This check is necessary to avoid *phantoms* [54], i.e., anomalies where a range scan fails to include a prior write (insert or delete) that modified the number of keys returned by a range scan, as well as for making sure that the transaction read the most recent value of each key returned by the scan. New or updated records can simply be found in the indexlet.

ACC's technique for avoiding phantoms due to newly inserted records relies on *phantom indicators*, which leverage ACC's asynchronous design to scalably signal the existence of a phantom to range-scanning transactions. Using the leaf nodes of the range index as partitions of its keyspace, writing transactions can produce a unique phantom indicator for each range covered by a leaf node. Each leaf node evolves through a series of version changes that happen whenever a merge to a range index affects that leaf node. Phantom indicators, uniquely derived from leaf nodes and their current version, are inserted by writing transactions into phantom detection indexlets (or *phantomlets*). Maintained for each range index, phantomlets allow range scanning transactions to detect phantoms at commit time. We describe ACC in detail in §5.2.2.

5.1.3 Durability

To provide durability, ScaleDB uses write-ahead redo logging, relying on a system-wide clock to assign globally-ordered timestamps to transactions. Hardware trends and experimental evidence (§5.3) indicate that system-wide clocks will remain in future servers. As a result, threads can scalably log their transactions without coordination at commit time while pushing the overhead of merging the logs to recovery.

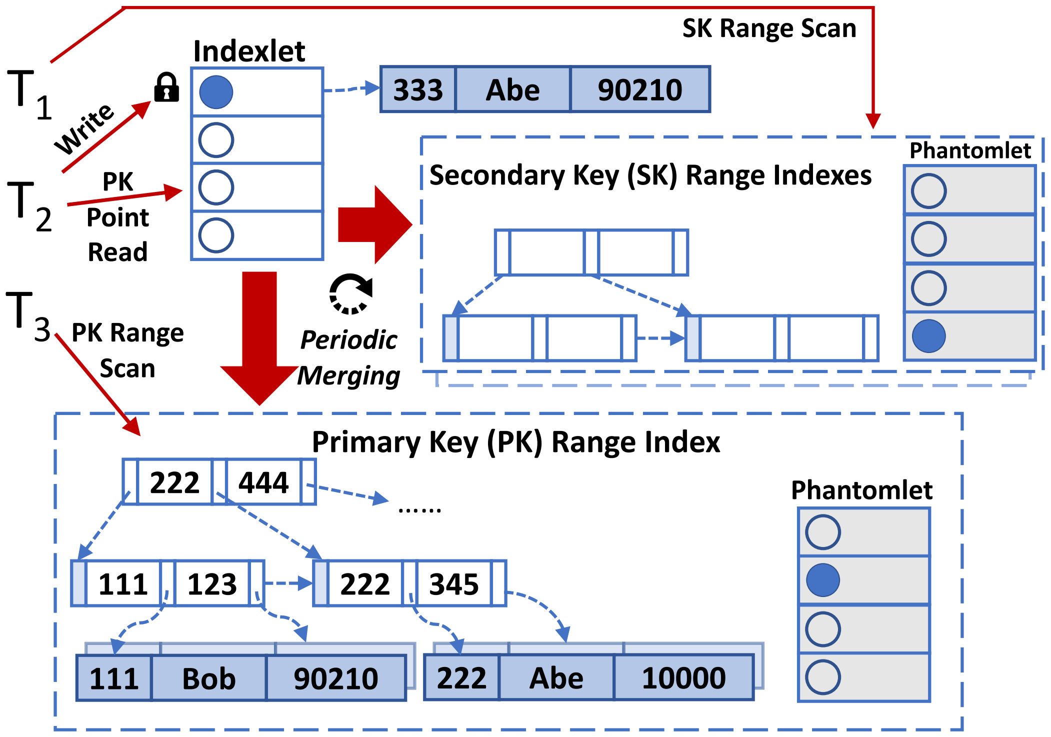


Figure 5.1: Asynchronous range index update for the *PERSON* table.

5.1.4 Example

To see how it all fits together, consider the example in Figure 5.1 (which continues from Figure 2.3). Transaction T_1 does a range scan by zipcode, which is executed on the appropriate secondary range index. T_3 does a range scan by SSN (the table's primary key), which is executed on the primary range index.

Concurrently, T_2 inserts the record with SSN 333 into the *PERSON* table. Instead of synchronously updating the range indexes and potentially contending with other writers, ScaleDB inserts the new record, using its primary key (SSN), in the table's indexlet and marks it as valid (filled circle). It does this atomically by acquiring a write lock on the indexlet entry. This may cause true contention if concurrent transactions access the same key, but it does not cause mechanism contention—if further concurrent transactions (not shown) insert more records into the *PERSON* table, ScaleDB can insert references to them into the indexlet without contention (in contrast to the example in Figure 2.4). T_2 also does a point read for an SSN. To do so, it first checks the indexlet for the latest version of the record, temporarily holding a read lock on the record's indexlet entry. It is not found there (empty circle), so T_2 next reads from the primary range index.

Periodically, the contents of the indexlet are merged into the underlying primary and secondary range indexes. The indexes are concurrent, so conflicting accesses by reading and merging threads are synchronized. We discuss the details of ScaleDB's concurrent range index in §5.3.3.

Range-scanning transactions consult phantomlets to detect phantoms due to newly inserted records. They do this for each leaf node of the range index, traversed as part of the range scan. To aid phantom detection, each writing transaction indicates *once* per version for a leaf node that it has inserted records. Here, T_2 inserts a phantom indicator for the $[222, 345]$ leaf node into the phantomlet, indicating a possible later merge of the key with SSN 333 into that range index node. Upon a merge, not all updates might fit in the $[222, 345]$ node and the structure of the range index might be altered during a merge. However, phantom indication is only required for unmerged records. We discard phantom indicators when the indicated records are merged. A reading transaction scanning just the $[111, 123]$ node does not abort, as there are no phantoms indicated for this node.

5.2 Design Details

5.2.1 Asynchronous Range Index Updates

To update range indexes asynchronously, we record delayed writes in indexlets for the duration of a per-indexlet and per-thread *merge epoch*. At the end of an epoch, a thread merges its writes from the indexlet into the associated range indexes, and starts a new epoch. For a given indexlet and thread, the merge epoch ends as soon as either (*i*) the maximum epoch duration has been reached; or (*ii*) the thread has filled a maximum *batch size* of entries in the indexlet. Both epoch duration and batch size are configured separately for every indexlet, and each thread decides independently for each indexlet when it has reached the end of its merge epoch.

5.2.1.1 Indexlets

ScaleDB uses hash-table-based indexlets (§5.3.1) with open addressing [75, 64, 5] to synchronously and scalably absorb concurrent, committed writes that affect range indexes. Thus, indexlets are associated with tables that have range indexes. For each table with range indexes, ScaleDB creates an indexlet, indexed by the table's primary key. If there is no primary key, ScaleDB creates an implicit primary key (a common practice [17]). The per-table indexlet naturally covers writes that affect secondary indexes, as secondary indexes refer to the primary index (as shown in Figure 2.3).

Recorded writes include insertions, updates, and deletions. Insertions and updates affecting a range index are simply recorded in the corresponding indexlet, and the record is updated on the heap, in per-thread arenas to avoid contention on memory allocation. Special care is required to ensure that deletes are handled consistently. Indexlets mark a record as deleted instead of deleting its key from the indexlet. This ensures that a later read of the same key finds the deleted record in the indexlet rather than finding an older version in a range index. It also allows coalescing a delete of a key, followed immediately by an insert of the same key, without merging the delete into the range indexes.

5.2.1.2 Merge Epoch

Each thread independently decides when its merge epoch ends after which it merges the keys and record references into the table's ranges indexes – their permanent home. Each thread t has a maximum batch size b_i of entries that it can occupy in indexlet i, before it has to merge those entries, i.e. the end of its merge epoch. Too small a b_i causes similar contention as synchronous merging into range indexes. Too large a b_i results in stale range scans, which can cause transaction aborts. We use $b_i =$ Expected write rate$(table_i)\times$ W-to-RS latency$(table_i)$.

During quiescent periods for write transaction activity, threads may not approach their maximum batch size quickly enough, leaving range indexes stale for too long. To avoid this, threads keep a timer since the start of the current epoch and merge when either the maximum batch size or an epoch duration is reached.

Indexlets must be sized (s_i) appropriately to minimize collisions. We use $s_i = 4 \times \#t \times b_i$ to minimize hash collisions. Given that each entry in an indexlet only occupies a single cache line, this results in modest memory consumption even for tables with a high write rate.

5.2.1.3 Asynchronous Merging

Each thread keeps a list of indexlet entries where it performed a write. At the end of its merge epoch, it sorts the list in primary range index key order. Then it iterates through the sorted list, atomically merging (using the per-entry lock) each individual record. Sorting facilitates range index updates for compact ranges of writes (§5.1). Repeated writes to the same key are coalesced within the indexlet at this point. Merging involves updating the range index and removing the record from the indexlet while holding the per-entry lock, thus ensuring atomicity for each key's merge. Each lock can be released as soon as the key is merged into the primary index. Finally, the thread invalidates all phantom indicators from the index's phantomlet. We discuss the details of this process in the next section.

If secondary indexes exist, the merging thread additionally retains private copies of each record reference in thread-local storage. After the primary range index is merged, the thread then merges each secondary index, one at a time, in the same way, and in deterministic order, using the thread-local copies. The copies are discarded at the end of this process. Once all indexes are merged, the overall merge ends.

5.2.2 Asynchronous Concurrency Control

We design asynchronous concurrency control (ACC), a concurrency control protocol that provides serializability within an asynchronous database. ACC is based on optimistic concurrency control (OCC), which it integrates with asynchronous range index updates. To do so, ACC uses two novel constructs: transactional locks in indexlets for atomic commit of writes (§5.2.2.2) and phantom indicators (§5.2.2.3).

OCC. OCC minimizes transaction contention by optimistically executing transactional reads and atomically publishing a transaction's writes at the end of its execution. To do so, OCC transactions execute in three phases—*read, validation, and commit*. During the *read* phase, reads are done optimistically, without holding locks, and are tracked in a transaction's private *read set*; writes instead are buffered in a private *write set*. The *validation* phase ensures that transactions may commit atomically. To do so, the database acquires locks on all values identified in the write set and then validates that collected values in the read set have not been altered by concurrently executing transactions. If the reads are validated, the *commit* phase commits the transaction's writes and releases its locks. Otherwise, the transaction aborts.

ACC. ACC extends the OCC phases and integrates them with asynchronous range index updates. During the read phase, point reads search the indexlet first, and, if they miss, search the primary range index. The same process is followed for updates and deletes, during the validation phase, allowing existing records to be brought into the primary indexlet first, before being updated in place. This guarantees that point queries always read the latest value of a record. On the other hand, range scans (from primary or secondary indexes) are executed directly on the range indexes, but need to check for phantoms at commit. We discuss phantom detection in §5.2.2.3.

5.2.2.1 Repairing Stale Range Scans

During the read phase, ACC can repair stale scans before returning them to reduce the chance of a later transaction abort. This is typically done for scans used in a later update or delete query. For instance, the TPC-C Delivery transaction has a range scan that returns the earliest order within a district in the NewOrder table and then deletes that order in the next query. This transaction can abort, even for a single thread, if the scan is done on the range index, but the earliest order returned by the scan has already been marked deleted in the indexlet in a previous Delivery transaction.

ACC repairs such scans, prior to returning them, by looking up each key in the indexlet to check if it has been updated or deleted. If so, it repairs the scan to return the latest version. To avoid paying this cost for all range scans, the client can explicitly set this option in the query for scans that will be updated or deleted.

ACC also maintains a per-thread per-table index of the keys which were inserted by each thread during its current merge epoch. When returning a range scan, ACC repairs it by merging any records returned by running the same scan on the local index as well. This avoids spurious aborts by the phantom detection algorithm (§5.2.2.3), due to keys that were inserted by the same thread in a prior transaction and are waiting to be merged into the range indexes.

5.2.2.2 Atomic Commit

ACC holds transactional locks on keys between the validation and commit phases, in order to atomically publish a transaction's writes. Since ScaleDB writes are asynchronous, ACC locks need to cover records referenced by indexlets. Indexlets never rehash, allowing ACC to hold locks directly in indexlet *entries* as a way to hold locks on *records*.

To build transactions, ACC provides two types of locks on records: *LockUniqueInsert* is used to atomically insert a record with uniqueness constraints, while *LockUpdDel* is used to atomically update or delete an existing record. These locks are acquired on a transaction's write set at the start of the validation phase, and released either at transaction abort or at the end of the commit phase.

LockUniqueInsert. To lock for the unique atomic insert of a record, ACC performs two steps:

(*i*) First, it searches for a duplicate record in the indexlet and, if not present, acquires a lock on an empty indexlet entry for the record to be inserted. This step is done atomically by calling *LockInsHashTbl*, provided by the indexlet hash table.

(*ii*) Once LockInsHashTbl has been acquired, ACC searches the primary range index to make sure that the key has not already been inserted there.

If either step fails, the transaction aborts. If both succeed, a lock for unique insert has been acquired.

The design of LockInsHashTbl is made tricky by the requirement of scalably and atomically running two searches i.e., for an empty entry and a possible duplicate record. These searches are run simultaneously in our open-addressing hash table (§5.1.1), which maintains a per-entry spinlock, as well as metadata indicating whether each entry is empty or used. For correctly terminating searches, however, additional metadata is needed: otherwise, the deletion of a record r would erroneously terminate future searches for records displaced by r (located later along the hash probe path). Thus, we also maintain metadata indicating whether each entry is a *search terminator* (details in §5.3.1).

LockInsHashTbl acquires per-entry spinlocks along the hash probe path. A probe can end when it finds an entry that is both empty and a search terminator. ACC sets e_{ins} – the future location of the record being inserted – to that entry's index. If, instead, the probe reaches an entry that is empty but not a search terminator, ACC still sets e_{ins} to that entry's index, but continues the search for a duplicate record, until the probe lands on a search terminator entry. Our open addressing scheme probes indexlet entries in a deterministic order for each record. Hence, contending transactions attempting to insert the same record are serialized behind this insert. If a duplicate is found, all spinlocks are released and the transaction is aborted. If a duplicate is found, but marked deleted, then ACC uses the deleted record's entry as e_{ins}. If a duplicate is not found, then LockInsHashTbl is successful. In that case, ACC releases any acquired spinlocks on entries after e_{ins} in the probe path. Spinlocks on e_{ins} and entries before it in the probe path, are held until the larger LockUniqueInsert is released: this allows atomically inserting the record and updating the search termination metadata (see §5.3.1) at transaction commit. With a properly sized indexlet, probe lengths are short and there is negligible mechanism contention for unique inserts.

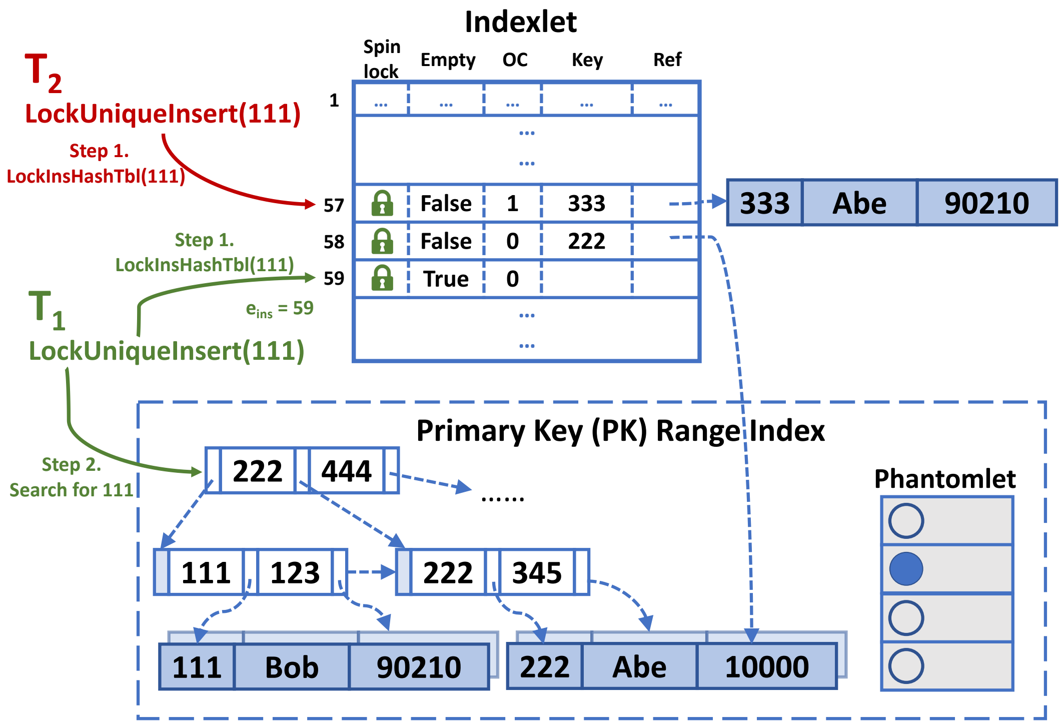


Figure 5.2: LockUniqueInsert Example.

Figure 5.2 shows an example illustrating LockUniqueInsert. Transactions T_1 and T_2, on different threads, are in their validation phase. They are concurrently trying to acquire LockUniqueInsert for a record with primary key (on SSN) 111. T_1 acquires LockInsHashTbl in step 1. Its hash probe starts at entry 57 in the indexlet, which is currently occupied by a record with key 333 – inserted by a recently committed transaction. Subsequently, another transaction brought the record with key 222 into the indexlet, for an update; it was inserted into entry 58 due to collision with key 333. T_1's hash probe acquires spinlocks along its probe path, until it lands on entry 59, which is both empty and a search terminator: thus, successfully acquiring LockInsHashTbl for key 111. Here, overflow counts (OC in the figure, see §5.3.1) are used to terminate searches (when OC == 0).

In step 2, T_1 searches the primary range index for key 111, to ensure uniqueness; since it finds the record, it will abort. If T_1 had been able to commit, it would have incremented the OC for entries 57 and 58 and inserted the new record (with key 111) into $e_{ins} = 59$, before releasing the spinlocks. Meanwhile, T_2 gets serialized behind T_1 (on entry 57's spinlock), trying to acquire LockInsHashTbl. It will eventually abort as well.

LockUpdDel. To acquire a LockUpdDel, ACC performs two steps:

(*i*) First, it searches the indexlet for the record and, if found, locks the entry. This step is done atomically by calling *LockRUDHashTbl*, provided by the indexlet hash table.

(*ii*) If LockRUDHashTbl fails, because the record was not found in the indexlet, ACC acquires LockInsHashTbl for the record (in the indexlet), fetches the record from the range index, inserts a reference to the record in e_{ins} and then downgrades the lock to LockRUDHashTbl: which involves releasing the spinlocks on the entries before e_{ins} in the probe path.

LockRUDHashTbl is simpler than LockInsHashTbl, since it does not need to atomically enforce uniqueness or maintain the metadata for search termination. It acquires per-entry spinlocks along the hash probe path, but releases each spinlock as it moves to lock the next entry in the path. A probe can end when it either finds the record or lands on a search terminator entry. In addition to its use in the first step of LockUpdDel, LockRUDHashTbl is also used to atomically search the indexlet for point queries, during the read phase of the transaction. For certains reads, the hash table also provides a *LockFreeRdHashTbl* call, as an optimization (see §5.3.2).

For range updates or deletes, we search the range indexes directly and acquire LockUpdDel for every key satisfying the predicate. If there is not enough space in the indexlet, the transaction aborts. In this rare case, the indexlet is merged and temporarily disabled to retry the transaction synchronously, re-enabling the indexlet after the transaction commits.

5.2.2.3 Asynchronous Phantom Detection

Phantom detection is difficult in a database with asynchronously updated range indexes, as phantoms may occur in indexlets, which do not support efficient range lookup. ACC's technique for detecting phantoms leverages the leaf nodes of a range index which undergo coarse-grained version changes due to asynchronous merges by different threads. To track these changes, each leaf node l maintains a version number v_l which is incremented only when an insert or delete is merged into that node. If l splits due to an insert, then half of its keys are moved to a sibling leaf node m with $v_m = 0$ while v_l is incremented.

To detect phantoms, ScaleDB uses a *phantomlet* per-range index to perform a scalable variant of index node validation [152]. Phantomlets use the indexlet architecture, as described in §5.2.1, but do not need merging, as they carry only phantom metadata. Writing transactions atomically write to phantomlets at transaction commit, indicating that they have inserted a phantom into the corresponding range index. ScaleDB makes this scalable in the common case by using the leaf nodes of a range index as partitions of its keyspace. This allows inserting transactions and range scanning transactions to coordinate phantoms inserted into a leaf node's range by using a unique *phantom indicator* derived from the current version of that leaf node. For a leaf index node l at version v_l, this phantom indicator is composed of the concatenation of the leaf node's memory address M_l and v_l.

At commit time, for each inserted key k, the inserting transaction asks the range index for the phantom indicator $< M_l,\ v_l >$ of the leaf node l that currently covers the range intersecting with k. Then, after successfully acquiring LockUniqueInsert on k (and locking the rest of the write set) it acquires a LockInsHashTbl on $< M_l,\ v_l >$ in the phantomlet. If that fails, because the phantom indicator already exists in the phantomlet, it instead acquires a LockRUDHashTbl. This lock allows it to atomically increment the value of the phantom indicator (initially 0), if the transaction later successfully validates. Threads keep track of the phantom indicators they have inserted and decrement their values at the end of their merge epoch. The last thread which decrements the value to 0, removes it from the phantomlet.

When validating a range scan, a reading transaction can use the same phantom indicator to check whether a phantom was inserted in a range covered by the leaf node *at the version it read*. To do so, ACC splits OCC's read set into two parts and extends them with additional information. For each point query, the key of a record r is stored along with a copy t_r^{PS} of the record's current commit timestamp t_r (§5.2.3) in a *point read set*. Storing the commit timestamp allows efficiently verifying whether the record changed, later during validation; it also enables an optimization for reducing aborts (discussed below). For every range scan, ACC stores the keys of the scan results in the point read set, but also stores in a *range read set*, a phantom indicator for each range index leaf node encountered during the scan. It also stores the range scan predicate in the range read set.

Read set validation happens differently for the point read set and the range read set. For the point read set, ACC reads from the indexlet and (if not found) then searches in the primary range index. If the key of record r is not found in either index or $t_r^{PS} \neq t_r$ (r received a write), the transaction is aborted. An optimization here is to only abort if $t_r < t_T$, where t_T is the timestamp allocated by this committing transaction (§5.2.3).

ACC validates the range read set as follows. For each range scan, it asks the range index for the current list c of phantom indicators that match the range scan predicate. If c is different in length than the original list o, stored in the range read set, it aborts. If not, then there is still the chance that phantoms were inserted, but, either they have not been merged yet, or they were merged but did not result in leaf node splits. To rule out these possibilities, ACC goes through each pair of phantom indicators in c and o, at the same index in the lists, and verifies two things: first, that the pair are identical, and second, that performing a LockFreeRdHashTbl (§5.3.2) for this phantom indicator on the phantomlet, returns nothing. If any of these checks fail, then again it aborts.

Figure 5.3 shows a simple example illustrating asynchronous phantom detection.

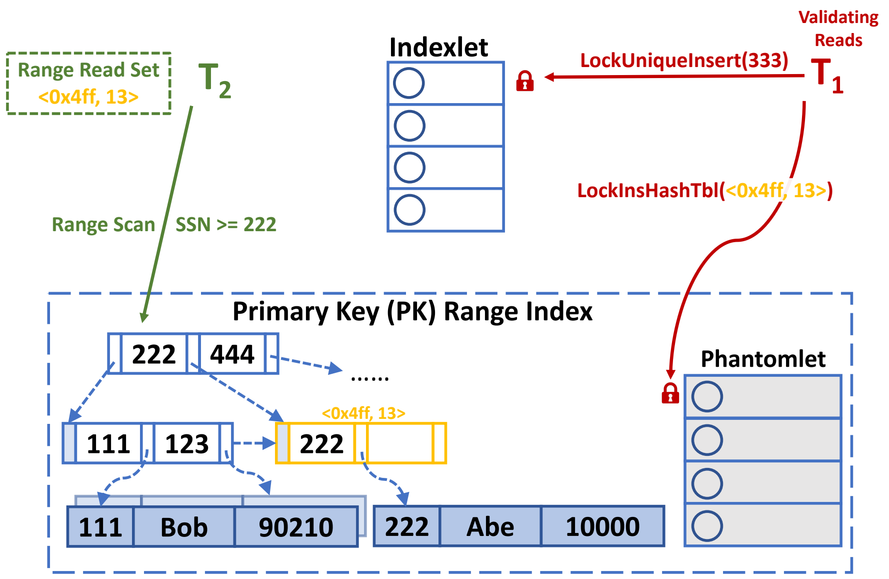
5.2.3 Durability

ScaleDB achieves durability using write-ahead logging to a redo log. Each worker thread writes to its own separate log, without coordinating with any other worker thread. To ensure that transactions do not read values that have not been made durable, a thread only releases write locks and replies back to the client once it has logged the transaction to its redo log. Each redo log entry contains the new values of the keys written by the transaction T as well as a commit timestamp t_T assigned to it during the validation phase, after all the locks have been acquired by ACC. This timestamp, unique for each transaction, is derived from a scalable system-wide clock (§5.3) and is thus consistent with T's place in the serializable order. During recovery, ScaleDB first merges all the per-thread transaction logs in timestamp order, and then replays them.

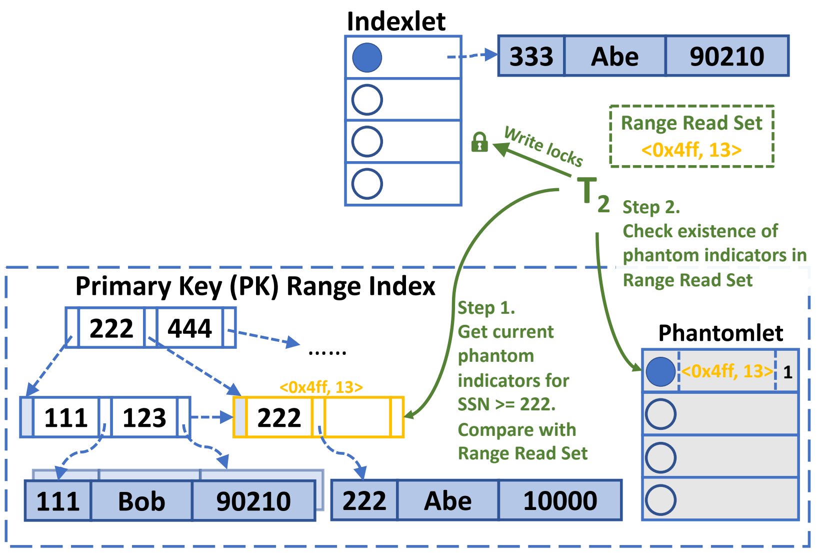
To see why ScaleDB is recoverable despite uncoordinated logging, consider the example of three transactions $T_1 \xrightarrow{ww} T_2 \xrightarrow{rw} T_3$, each of them running on a separate thread. T_1 writes $x_1 = 42$ and T_2 read-modify-writes that value to $x_2 = 52$, thus creating both a write-after-write dependency (ww) and read-after-write (wr) dependency with T_1. Next, T_2 reads $y_1 = 33$; later T_3 read-modify-writes it to $y_2 = 36$, thus creating a write-after-read (rw) dependency between T_2 and T_3.

Because ScaleDB only releases write-locks after the log entry has been made durable, if T_1 is not logged, then T_2 will either read x_0 or it will wait for T_1's write lock to be released to read x_1 (enforced by indexlets' lock-free read operation). Thus, after a crash, it cannot be the case that T_2 is logged but T_1 isn't. This argument extends transitively to a chain of such direct dependencies.

The second possibility is that after a crash T_2 is not logged, but both T_1 and T_3 are. In this case, ScaleDB must not have committed T_2 and replied back to the client. Thus, it will recover only T_1 and T_3, in order, which is fine. Notice that, if infact T_2 *does* get successfully logged, ScaleDB's system-wide timestamps allow correctly ordering T_2 and T_3's log entries at recovery, despite the fact that there was no direct communication between them.



(a) Before validating successfully, transaction T_1 acquires locks for atomically inserting the record with SSN = 333 and a phantom indicator <0x4ff, 13>, corresponding to leaf index node [222,]. This node covers the range containing 333. Concurrently, transaction T_2 does a range scan for SSN >= 222, during its read phase.


Indexlet
333
Abe
90210
Range Read Set
<0x4ff, 13>
Write locks
T2
Step 2.
Check existence of phantom indicators in Range Read Set
Primary Key (PK) Range Index
222
444
......
Step 1.
Get current phantom indicators for SSN >= 222.
Compare with Range Read Set
Phantomlet
<0x4ff, 13>
1
<0x4ff, 13>
111
123
222
111
Bob
90210
222
Abe
10000

(b) Transaction T_2 detects phantom indicator <0x4ff, 13> corresponding to [222,] while validating the range scan SSN >= 222. It will abort: T_1 committed earlier, but T_2's range scan missed the record with SSN = 333, inserted by T_1 in the indexlet.

Figure 5.3: Asynchronous Phantom Detection Example.

5.2.4 Correctness

Using ACC, ScaleDB guarantees serializability [39], with the additional guarantee that the equivalent serial order is one where transactions are ordered by their commit timestamps. We provide a detailed proof of correctness for ScaleDB's transactional guarantees in Appendix 2.

ACC derives its correctness guarantees, in part, from the guarantees provided by the locks (§5.2.2.2) and data structures (§5.2.1.1, §5.3.3), it builds upon, as well as its descendence from OCC: which guarantees serializability [152]. The key difference from OCC is that ACC must deal with ScaleDB's asynchronous updates to range indexes. As our proof of correctness shows, ACC's Atomic Commit and Asynchronous Phantom Detection protocols ensure, that despite ScaleDB's asychronous architecture, it continues to guarantee serializability.

5.3 Implementation

We implement ScaleDB by modifying the Peloton [28] in-memory SQL database, written in C++. We replace the storage back-end while retaining the code for networking, SQL parsing, query planning and query optimization.

5.3.1 Indexlet and Phantomlet Hash Table

Our indexlet and phantomlet implementations build on a simple open-addressing [75] hash table which uses linear probing for resolving collisions. We considered more sophisticated open-addressing schemes like Cuckoo hashing [3, 12] but found that the ability to hold transactional locks would have been complicated by displacement of keys and the fact that the cuckoo hashing probe path is an undirected graph with a possible cycle, which could have caused deadlocks. Also recall that our hash table does not need to rehash (§5.1.1): thus we can avoid mechanism coordination on maintaining a count of occupied entries in the entire hash table.

One issue with using an open addressing hash table is how to ensure that searches terminate correctly after merging. When removing a record r from an indexlet entry we cannot simply mark the entry as empty, because then any records displaced by r would not be found on a subsequent lookup (the search would terminate at r). Tombstones, which are traditionally used in open addressing tables, have the problem of accumulating and making search probes ever longer. Instead, we used a scheme used by the recent non-concurrent F14 hash table [64, 45]. Each entry maintains an *overflow count*, that is incremented whenever an insert probe finds the entry already occupied. When removing a record reference at the end of a merge epoch, we atomically decrement the overflow counts on its probe path before marking it as free. Similarly, when inserting a record reference, we atomically increment the overflow counts on its probe path. An entry whose overflow count is zero is a search terminator (§5.2.2.2).

5.3.2 Lock-Free Reads

To avoid reader contention on the same indexlet or phantomlet entries, ScaleDB provides *LockFreeRdHashTbl* (based on seqlocks [4]). To implement these, we add a version number to the per-entry spinlocks in the indexlet or phantomlet hash tables. Writers (doing inserts, updates or deletes) increment the version number after acquiring the spinlock but before doing any writes. At spinlock release, the version number is incremented again. Readers do not acquire the spinlocks but instead read the version number, before and after they perform the read. If the version number changed during the read or it is initially odd in value, then there was interference from a concurrent writer and the reader retries.

The limitation of this design is that it cannot be used if the data being read has internal pointers; otherwise, writers could invalidate pointers that a reader had already followed. To solve this, we can use Read-Copy-Update (RCU) [36] for implementing LockFreeRdHashTbl [152]. However, RCU can add significant complexity to the design; e.g., it requires garbage collection of previous versions of the data, after ensuring that no readers are actively reading it.

Our current prototype does not implement RCU. Instead, we only use LockFreeRdHashTbl for use-cases where the data does not have internal pointers; e.g., during the ACC validation phase, we use it to atomically search phantomlets, thus avoiding mechanism coordination between threads searching for phantom indicators for the same leaf node version of a range index. We also use LockFreeRdHashTbl to validate point reads in indexlets with fixed-length keys. If the data has internal pointers, we instead use LockRUDHashTbl (§5.2.2.2).

5.3.3 Concurrent Range Index

Our range index implementation is a B+ tree with optimistic latch coupling (OLC) [114], used in a recent study [157] that compared the scalability of state-of-the-art range indexes. In the OLC tree, reads do not acquire the per-node spinlocks when traversing the tree. Instead, they validate a per-node version number by reading it before and after reading the node's contents. If the two versions are not the same, they restart their traversal. Writers intially traverse like readers but restart and acquire spinlocks along the path if they detect interference from another writer or if they need to modify a node.

5.3.4 System-wide Synchronized Clock

For scalable durability, ScaleDB assigns timestamps to transactions derived from a system-wide synchronized clock. Synchronized hardware clocks are available on modern multi-core processors, such as the timestamp counter (TSC) on recent Intel x86 processors, which runs at a constant rate. Intel has indicated [97] that "this is the architectural behavior moving forward" and that "the OS may use invariant TSC for wall clock timer service". As a result Linux uses the TSC as the clock source on x86 across multiple CPU sockets, after running boot-time tests to ensure synchronization [13, 14]. Recent work [57, 60] on multi-core filesystems has used it for scalable ordering across cores. Finally, virtual machines also provide synchronized virtual TSCs by either using the underlying hardware (fast) or emulating it if not synchronized (slow) and even across migrations [30, 37].

On architectures where a system-wide TSC counter is not available, it is possible to use a dedicated timing thread [142] for handing out timestamps to threads. This approach requires a core dedicated to the timing thread, which continuously increments a local variable and then stores the value to a global time variable. A thread requiring a global timestamp simply reads the time variable. On the Intel Skylake architecuture, such a timing thread increments the local variable every 0.87 cycles which is actually 15% faster than the TSC [142].

5.4 Evaluation

Our evaluation aims to understand how ScaleDB performs in terms of throughput scalability of committed transactions on various workloads, including YCSB and TPC-C, and how the various ideas in the design of ScaleDB contribute to performance. Our comparison baselines are Peloton, upon which ScaleDB is built, and Cicada.

Our evaluation answers the following questions:

1. Why is the ScaleDB asynchronous architecture a scalable design (§5.4.1)? We evaluate the scalability of indexlets (phantomlets) and system-wide

timestamps given that these mechanisms are necessary for a scalable, asynchronous database.

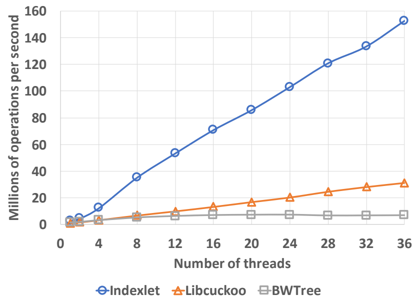
2. What is the application-level query scalability of an asynchronous database (§5.4.2)? We use the non-transactional YCSB benchmark to answer this question.

3. How is ScaleDB scalability affected when guaranteeing serializability for transactions (§5.4.3)? Is the transaction abort rate affected? We evaluate on the TPC-C benchmark, to answer this question.

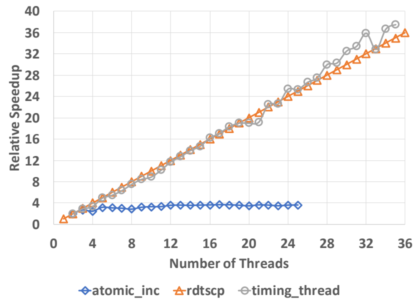
Testbed. All machines in the evaluation have 2×18-core Intel Xeon Gold 6154 CPUs with 36 cores and 72 hardware threads. 192GB of memory is divided across two NUMA nodes. Each machine has a Mellanox ConnectX-5 network card, operating at 100Gb/s. For networked benchmarks, we run a single database server and 4 client machines. Each client machine runs as many processes of the OLTP benchmark suite [19] as needed to saturate the database server. Accordingly, all experiments report peak throughput.

5.4.1 ScaleDB Mechanisms

Indexlets. We evaluate indexlet scalability against libcuckoo [12], an optimized concurrent hash table, and the BwTree [115, 157], a recent, lock-free range index structure. This evaluation is performed on a microbenchmark (included with libcuckoo) with a 50% read and 50% insert workload consisting of 64-bit integer keys and values. As Figure 5.4a shows, indexlets achieve nearly



(a) Indexlet scalability



(b) System-wide timestamp scalability

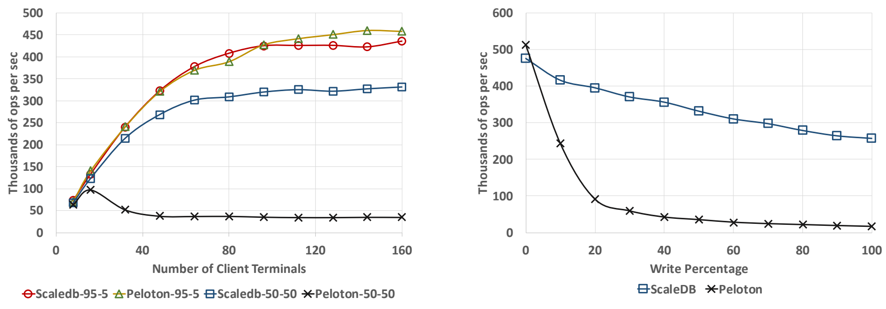
Figure 5.4: ScaleDB Mechanisms

5×libcuckoo and 25×BwTree throughput at 36 cores. Open addressing in indexlets provides better scalability than cuckoo hashing and the flat structure of hash tables scales better than tree indexes.

System-wide timestamps. We evaluate the TSC counters and timing thread approach described in §5.3 and compare them with an atomic increment as a global timestamp. As Figure 5.4b shows, both the timing thread and TSC approaches scale linearly to 36 cores, while the atomic increment approach does not scale beyond 4 cores.

5.4.2 Asynchronous Index Update

We evaluate ScaleDB's scalability of asynchronous updates to a single range index. A single range index is a conservative case for ScaleDB; the benefit of asynchronous updates grows with the number of range indexes that are updated in this way. For this purpose, we use the Yahoo! Cloud Serving Benchmark (YCSB) [72], which is a non-transactional workload. To generate



(a) Throughput **(b)** Write sensitivity

Figure 5.5: YCSB read-insert workload. 95-5 means 95% reads and 5% inserts. 50-50 means 50% reads and 50% inserts.

enough load, we access the database from 4 networked YCSB benchmark client machines. The YCSB benchmark defines a single table with an integer primary key and 10 string columns, each of size 100 bytes. Peloton uses the lock-free Bw-Tree [157] as the underlying primary range index on the integer key.

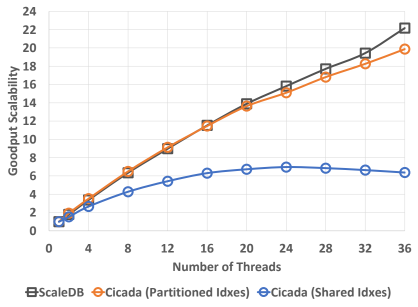
All experiments use 36 server threads, with each thread pinned to a separate core. We show scalability by increasing the number of client terminals sending operations to the database server. For ScaleDB, we set the maximum merge epoch duration to 100 ms and the maximum batch size per thread to 1,000 entries. Prior to running each experiment, we load the table with 1 million records. We use a Zipfian distribution of operations for reads and updates with $\theta = 0.99$, to simulate a skewed workload. For inserts, each client thread adds new records sequentially within its own interval of the primary key space, starting after the already inserted 1 million records, to avoid uniqueness conflicts.

Mechanism contention. Figure 5.5a shows terminal scalability of a read-insert workload for two points of read-insert intensity. The read-insert workload has only mechanism contention—reads and inserts are to disjoint keys. For 95% reads, both ScaleDB and Peloton scale with similar performance until all server cores are saturated. This is not surprising. Peloton's range index scales well when a workload is read-intensive. For a write-intensive workload with 50% inserts, Peloton's throughput collapses, while ScaleDB maintains 9.5× Peloton's throughput at scale.

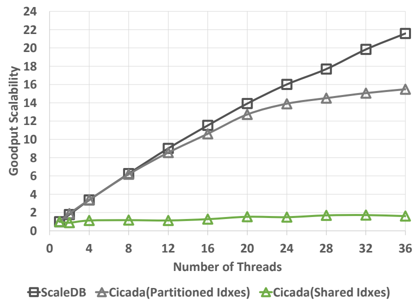
To show this effect in detail, we examine the sensitivity of both systems to an increasingly write-heavy load by varying the percentage of inserts in the workload, fixing the number of terminals to 160. Figure 5.5b shows that Peloton's throughput quickly collapses with increasing write intensity (knee-point at 20% inserts), while ScaleDB's throughput gradually declines. ScaleDB loses 46% of its peak throughput when the workload is write-only.

5.4.3 Serializability

We now evaluate asynchronous scalability with serializable transactions on the TPC-C benchmark, which has multiple tables and several primary and secondary range indexes. We compare with the Cicada [117] database. Cicada's prototype does not have a network layer and it uses a TPC-C implementation linked with the database binary, calling directly into the Cicada API as opposed to sending calls across the network. To make this an apples-to-apples comparison we do the same for ScaleDB. Also, Cicada's prototype



(a) Full



(b) NewOrd-Deliv

Figure 5.6: ScaleDB vs Cicada goodput scalability comparison on the TPC-C benchmark with partitioned and shared indexes for Cicada. Goodput counts only committed transactions. Full means the canonical TPC-C workload (i.e., 45% New-Order, 43% Payment, 4% Delivery, 4% Order Status and 4% Stock Level). NewOrd-Deliv means a 50% New-Order and 50% Delivery workload.

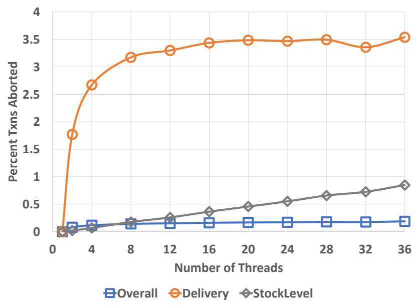
pre-allocates all of its memory using huge pages. Recent work from Huang et al. [96] has recommended avoiding this strategy since it “changes system dynamics significantly – for instance, preallocated indexes never change size”. Further, Huang et al. show that Cicada, with pre-allocation, experiences a performance collapse at high core counts due to memory exhaustion (which we observed in our early experimentation as well). Therefore, we modify Cicada to instead call into jemalloc, which is what ScaleDB uses for memory allocation.

The maximum per-thread batch size for these experiments is set to 2048 for most tables, calculated using the method outlined in §5.2.1.2. Three of the exceptions are the WAREHOUSE, DISTRICT and ITEM tables. Cicada uses hash indexes for these tables, since TPC-C does not run range scans on them. We do the same for ScaleDB, by setting the batch size to infinity for these tables, so that they are served from the respective indexlets, without merging. The final exception is the NEW-ORDER table, for which we set the batch size to 0 (i.e., synchronous merging into range index). This is needed to avoid excessive aborts of the Delivery transaction, which has a very small W-to-RS latency for inserts done by the New-Order transaction on this table.

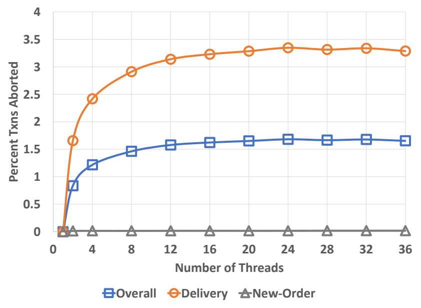
Figure 5.6 shows the TPC-C evaluation. The setup for these experiments is the same as that of Figure 2.6 (§2.2.3.1). ScaleDB does not have a partitioned index configuration, so the results for ScaleDB are with a shared index configuration.

Despite shared indexes, ScaleDB scales slightly better than even Cicada's partitioned index configuration and significantly better than Cicada's shared index configuration, on the canonical TPC-C benchmark (Figure 5.6a). On the more index contended NewOrd-Deliv benchmark, ScaleDB maintains its scalability even though Cicada's scalability is impacted on both the partitioned and shared index configurations. These results show the efficacy of ACC and asynchronous index updates in decoupling transaction commit from contention on range index updates.

Given ScaleDB's asynchronous design and the fact that transactions can do stale reads from range indexes in between batch merges, an important concern is how the abort rate behaves with increasing number of threads. Figures 5.7a and 5.7b show this evaluation. On the canonical TPC-C benchmark, only the Delivery and StockLevel transactions have a non-negligeable abort rate. The Delivery abort rate stabilizes at 3.5% around 16 cores (for both workloads), which implies that this should not be a problem for scalability even beyond 36 cores. Similarly the Stock-Level abort rate is quite



(a) Full



(b) NewOrd-Deliv

Figure 5.7: ScaleDB Abort Rate. Full means the canonical TPC-C workload (i.e., 45% New-Order, 43% Payment, 4% Delivery, 4% Order Status and 4% Stock Level). NewOrd-Deliv means a 50% New-Order and 50% Delivery workload.

manageable, staying under 1 percent even for 36 cores.

5.5 Limitations

Limitations of ScaleDB's design include: a need to know the W-to-RS latency and expected write rate of database tables for performance tuning, and workloads that can cause performance degradation due to excessive aborts or high true-contention.

Recall from §5.2.1.2 that ScaleDB uses the W-to-RS latency and expected write rate of each database table, in order to calculate the maximum batch size per-thread. This is not a big problem if these parameters remain stable, since it requires doing workload analysis once. For real-world workloads, however, diurnal patterns and load spikes can cause changes. Recent work [153] on automatically tuning database knobs using machine learning has shown promising results. Applying this approach to ScaleDB's design can mitigate or solve this limitation.

Another limitation is workloads that immediately perform range scans on recent inserts. Our analysis in §3.1.1 shows that for such workloads, it is likely that a small number of tables in the database are impacted. The simple solution is to selectively switch to synchronous merging for just those tables. Notice, that even in the absolute worst case, where *all* tables are being synchronously merged, ScaleDB just loses its scalability advantage; its scalability will be no worse than that of a regular OCC database with synchronous merging.

Finally, our current design does not include optimizations for workloads that have high true contention. These can include workloads with contention on the same records or true contention on the range index e.g., if all transactions try to append to the end of the table. Recent work [96] has shown that OCC databases can outperform Multi-Version Concurrency Control (MVCC) databases on high-contention workloads. Integrating these techniques with ScaleDB's design is a promising avenue for future work.

Chapter 6

Conclusions

This dissertation identified mechanism coordination as a form of unnecessary coordination that can limit the scalability of OLTP storage systems. We described two instances of mechanism coordination that impact two different classes of storage systems. We identified how slowdown cascades are a fundamental limitation of enforcing causal consistency as a global property of distributed and replicated data-stores. We also identified range index contention as a form of mechanism coordination for in-memory databases that provide serializability to their clients. We showed how these seemingly disparate problems arise because of similar implicit pessimistic assumptions about client behavior, which leads to write-synchronous implementations. Based on this insight, we proposed a thesis statement that outlined a general approach to building asynchronous systems that avoid mechanism coordination. We validated the thesis statement by building two systems which solve the mechanism coordination problems identified earlier.

We described the Occult system which solves the slowdown casade problem by moving the output commit to the client: the data store makes its updates available as soon as it receives them. Clients then enforce causal consistency on reads only for updates that they are actually interested in observing, using compressed timestamps to track causality. Occult follows the same philosophy for its scalable general-purpose transaction protocol: by ensuring that transactions read from a consistent snapshot and using timestamps to guarantee atomicity, it guarantees the strong properties of Parallel Snapshot Isolation while avoiding its scalability bottleneck.

We also described the ScaleDB system which solves the mechanism contention problem on range indexes. ScaleDB is an asynchronous in-memory database with SQL and networking support that provides scalable and serializable transactions. ScaleDB asynchronously updates range indexes by temporarily holding delayed writes in indexlets that are merged periodically into read-scalable range indexes. ScaleDB uses asynchronous consistency control (ACC) to provide serializability for transactions within an asynchronous database. ACC extends OCC with asynchronous phantom detection via phantomlets and atomic transcation commit using locks in indexlets.

Appendix

Appendix 1

Occult Pseudocode

1.1 Causal Timestamp Interface

The listing below shows the interface and basic implementation of a causal timestamp (§4.2.2), but skips compression schemes (§4.3). However, compression only changes the implementation of this interface.

Listing 1.1: Interface of a Causal Timestamp

```
class CausalTimestamp:
  def init(N):
    V = [0] * N

  # Get shardstamp for shard_id
  def getSS(shard_id):
    return V[shard_id]

  # Return the shardstamp with maximum value
  def maxSS():
    return max(V)

  # Update the shardstamp for shard_id to new_ss
  def updateSS(shard_id, new_ss):
    V[shard_id] = max(V[shard_id], new_ss)

  # Merge another CausalTimestamp into this object
  def mergeCTS(other_cts):
    for i in range(0, len(V)):
      V[i] = max(V[i], other_cts[i])
```

The pseudocode listings in this appendix were published in a paper describing the Occult [126] system. The author of this dissertation conceived, designed, implemented and evaluated the Occult system, and led the writing of the prior publication: S. A. Mehdi, et al. 2017. "I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades". In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, USA, 453–468.

1.2 Basic Protocol

The listings below show the client and server-side pseudocode for the Basic Occult Protocol as described in §4.2.3.

Listing 1.2: Server-side Basic Protocol

```
def write(key, value, deps): #(on masters)
  shard_id = shard(key)
  shardstamps[shard_id] += 1
  shardstamp = shardstamps[shard_id]
  deps[shard_id] = shardstamp
  store(key, value, deps)
  for s in mySlaves(shard_id):
    async(s.replicate(key, value, deps, shardstamp))
  return shardstamp

def replicate(key, value, deps, shardstamp): #(on slaves)
  shardstamps[shard(key)] = shardstamp
  storeValue(key, value, deps)

def read(key):
  shardstamp = shardstamps[shard(key)]
  return (getValue(key), getDeps(key), shardstamp)
```

Listing 1.3: Client-side Basic Protocol

```
# cli_ts is the client's causal timestamp
def write(key, value):
  shard_id = shard(key)
  master_server = master(shard_id)
  shardstamp = master_server.write(key, value, cli_ts)
  cli_ts.updateSS(shard_id, shardstamp)

def read(key):
  shard_id = shard(key)
  local_server = local(shard_id)
  value, deps, shardstamp = local_server.read(key)
  if isSlave(local_server) and shardstamp < cli_ts.getSS(shard_id):
    return finishStaleRead(key)
  else:
    cli_ts.mergeCTS(deps)
    return value
```

1.3 Transactional Protocol

The listings below show the pseudocode for the Occult transactional protocol, as described in §4.4.2. Note that if multiple transactions concurrently update different objects in the same shard s, in the commit phase each write w is applied at s (and at its slaves) in the (total) order determined by the value of the shardstamp assigned to w during the validation phase. The server-side pseudocode below achieves this property by locking shards instead of objects during the validation phase.

Listing 1.4: Server-side Transactional Protocol

```
# allocate new shardstamp using loosely synchronized
# clocks as described in Section 5
def newShardstamp(max_cli_ss, shard_id):
  new_ss = max(currentSysTime(), max_cli_ss)
  if new_ss < shardstamps[shard_id]:
    return shardstamps[shard_id] + 1
  else:
    return new_ss + 1

def read(key):
  shardstamp = shardstamps[shard(key)]
  return (getValue(key),getDeps(key),shardstamp)

def prepare(tid, key, value, max_cli_ss):
  if not islocked(shard(key)):
    lockwrites(shard(key))
    if tid not in prepKV: # prepared txns key vals
      prepKV[tid] = list()
    prepKV[tid].append((key, value))
    shardstamp = shardstamps[shard(key)]
    new_ss = newShardstamp(max_cli_ss, shard(key))
    return (new_ss, getDeps(key))
  else:
    throw LOCKED

def commit_server(tid, deps):
  for key, value in prepKV[tid]:
    shardstamps[shard(key)] = deps.maxSS()
    store(key, value, deps)
    shardstamp = shardstamps[shard(key)])
    unlockwrites(shard(key))
    for s in mySlaves():
      async(s.replicate(key,value,deps,shardstamp)

def abort_server(tid):
  for key, value in prepKV[tid]:
    unlockwrites(shard(key))
```

Listing 1.5: Client-side Transactional Protocol

```
# cli_ts is the client's causal timestamp
def startTransaction():
  TID = newTransactionID()
  ReadSet = set()
  OWSet = set() # Overwrite Set
  Writes = dict() # Writes done by this transaction
  cli_ts_save = copy(cli_ts)

def write(key, value):
  Writes[key] = value

def read(key):
  if key in Writes:
    return Writes[key] # Return the value we wrote
  else:
    shard_id = shard(key)
    local_server = local(shard_id)
    cli_ss = cli_ts.getSS(shard_id)
    value,deps,shardstamp = local_server.read(key)
    if isSlave(local_server) and shardstamp < cli_ss:
      value,deps,shardstamp = finishStaleRead(key)

    ReadSet.add(Elem(key,shard_id,deps,shardstamp))
    cli_ts.mergeCTS(deps)
    return value

def validate(S1, S2):
  for x in S1:
    for y in S2:
      if x.shardstamp < y.deps.getSS(x.shard_id):
        return False
  return True

def abortTransaction(prepared_servers, tid):
  cli_ts = cli_ts_save
  for server in prepared_servers:
    server.abort_server(tid)
  return False

def commitTransaction():
  prepared_servers = set()
  if not validate(ReadSet, ReadSet):
    return abortTransaction(prepared_servers, TID)

  for key, value in Writes:
    master_server = master(shard(key))
    try:
      max_ss = cli_ts.maxSS()
      new_ss,deps =
        master_server.prepare(TID,key,value,max_ss)
      cli_ts.updateSS(shard(key), new_ss)
      OWSet.add(Elem(key,shard(key),deps))
      prepared_servers.add(master_server)
    except LOCKED: #can retry lock here before abort
      return abortTransaction(prepared_servers, TID)

  if not validate(ReadSet, OWSet):
    return abortTransaction(prepared_servers, TID)
  else:
    for server in prepared_servers:
      server.commit_server(TID, cli_ts)
    return True
```

Appendix 2

ScaleDB Serializability Proof

ScaleDB, using Asynchronous Concurrency Control (ACC), guarantees that any schedule of committed transactions is equivalent to the serial schedule, where those transactions are ordered by their commit timestamps (§5.2.3). To prove this, we start with some assumptions:

Assumption 1. For simplicity, we assume a database with a single table, indexed by a primary key range index, along with the corresponding indexlet and phantomlet. Further, we assume that the following queries can be executed on this table, using its primary key: point read of a record, range scan read using a predicate, point delete of a record and point update of a record.

Assumption 2. The concurrent range index data structure (e.g., §5.3.3), used for the primary key range index, atomically executes individual queries (listed in Assumption 1).

Assumption 3. A spinlock on an indexlet entry serializes concurrent accesses to that entry, with mutual exclusion, while the lock is held.

Next, we argue a series of lemmas showing that various kinds of locks on a single record are serialized w.r.t. each other as well as the merging of the record into the primary range index. We will later build on these lemmas for arguing the correctness of transactions.

Lemma 1. *Concurrent requests for LockUniqueInsert on the same key k are serialized. If a transaction T_i successfully obtains a LockUniqueInsert on a key k of record r, then two guarantees are provided: first, no other transaction obtains a LockUniqueInsert on the same key k of another record $r^{'}$, while T_i holds the lock, and second, no other record $r^{''}$ with the same key k exists in the table.*

PROOF. Since LockUniqueInsert is composed of two steps (§5.2.2.2), we first argue that concurrent calls to LockInsHashTbl are serialized. Assume that they are not, and that key k of record r hashes to entry e_{init} in the indexlet. Concurrent requests for LockInsHashTbl on k will first conflict on the spinlock of e_{init} (Figure 5.2) and only one transaction will be able to obtain the spinlock, by Assumption 3. However, this contradicts our current assumption. Hence, concurrent calls to LockInsHashTbl are serialized.

Since all transactions attempting to acquire LockUniqueInsert, acquire LockInsHashTbl in the first step, this also proves that calls to LockUniqueInsert are serialized. Further, once a transaction successfully obtains LockInsHashTbl, it continues to hold the spinlocks on entries from e_{init} to e_{ins}, while

LockInsHashTbl is held. Since LockInsHashTbl is held as long as LockUniqueInsert is held, the first guarantee in the lemma is ensured.

Assume now, that T_i successfully acquires LockUniqueInsert, but the second guarantee in the lemma is not ensured. Further, assume that the duplicate record r'' with key k is inserted by another transaction T_a. We have already shown that T_a will be serialized w.r.t. T_i, on to the call to LockUniqueInsert. If T_a is serialized after T_i, then it cannot be responsible for the second guarantee being violated. Instead, if T_a is serialized before T_i, then it must have committed before T_i started, since it was able to successfully insert r'' and the LockUniqueInsert is only released in ACC at the end of the commit phase. There are three sub-cases here:

(i) r'' is in the indexlet, at the time T_i acquires the spinlock on e_{init}. In this case, T_i will find r'' (see protocol in §5.2.2.2) and LockInsHashTbl will be unsuccessful. However, this contradicts our assumption that T_i successfully acquired LockUniqueInsert.

(ii) r'' is in the primary range index, at the time T_i acquires the spinlock on e_{init}. In this case, T_i will find it, when it searches the primary range index in the second step of acquiring LockUniqueInsert. Again, this contradicts our assumption that T_i successfully acquired LockUniqueInsert.

(iii) r'' is in the indexlet, but it is in the process of being merged into the primary range index, at the time T_i acquires the spinlock on e_{init}. In this case, the thread merging r'' will acquire spinlocks, from e_{init} to e_{ins}, in order

to atomically update the overflow counts on entries (§5.3.1). It will hold those spinlocks, until r'' is merged into the primary range index. *Thus T_i will be serialized w.r.t. the merging of r''*. If T_i is ordered before the merging thread, then we are back in case (i). Instead, if T_i is ordered after the merging thread, then we are back in case (ii). Since neither case is possible, we have again arrived at a contradiction.

Finally, since all three cases (i), (ii) and (iii) are impossible, the second guarantee in the lemma is ensured.

Lemma 2. *Concurrent requests for LockUpdDel on the same record r are serialized. If a transaction T_i successfully obtains a LockUpdDel on a record r, then no other transaction can obtain LockUpdDel on r, while T_i holds the lock.*

PROOF. For simplicity, we do not consider the case where record r is not found in the table. Then, there are two cases for LockUpdDel, based on its two steps (§5.2.2.2):

(i) If there are concurrent calls to LockRUDHashTbl in the first step, one of those calls is successful, i.e., it finds the record r in the indexlet.

We first argue that concurrent calls to LockRUDHashTbl are serialized. Assume that they are not, and that key k of record r hashes to entry e_{init} in the indexlet, and is eventually inserted in e_{ins}. Since LockRUDHashTbl releases spinlocks along its probe path, until it finds k in e_{ins}, concurrent requests for LockRUDHashTbl on k will eventually conflict on the spinlock of e_{ins}; only one transaction will be able to obtain the spinlock, by Assumption 3. However, this contradicts our assumption. Hence, concurrent calls to LockRUDHashTbl are serialized.

By implication, this also proves, for case (i), that concurrent calls to LockUpdDel are serialized. Once a transaction successfully obtains LockUpdDel, it continues to hold the spinlock on e_{ins}, while LockRUDHashTbl is held. Since LockInsHashTbl is held as long as LockUpdDel is held, this proves the lemma for case (i).

(ii) If LockRUDHashTbl fails, because the record was not found in the indexlet, ACC acquires LockInsHashTbl for the record (in the indexlet) and then fetches the record from the range index.

We have already shown in Lemma 1 that concurrent calls to LockInsHashTbl are properly serialized. The only remaining case here is when this call to LockInsHashTbl is concurrent with a LockRUDHashTbl call, arising from a concurrent LockUpdDel call. We observe that they will conflict on any one of the spinlocks from e_{init} to e_{ins}, since LockInsHashTbl holds on to all of them. By Assumption 3, they will be serialized on that entry. Since LockInsHashTbl is held as long as LockUpdDel is held, this proves the lemma for case (ii) as well.

Since both cases are proven, this proves the lemma.

Lemma 3. *Concurrent requests for LockUpdDel and LockUniqueInsert on the same key k, are serialized.*

We omit a detailed proof of Lemma 3 to avoid repetition with the proofs of Lemmas 1 and 2. Part of this proof has already been argued in case (*ii*) of Lemma 2.

Lemma 4. *Merging of a record* r *with key* k*, from the indexlet to the primary range index, is atomic and serialized w.r.t. concurrent requests for LockUpdDel or LockUniqueInsert on the same key* k

Again, we omit a detailed proof to avoid repetition. We have already argued in case (*iii*) of Lemma 1, why merging of a record r with key k is serialized w.r.t. a concurrent request for LockUniqueInsert on k. The argument for serialization with LockUpdDel is similar.

Further, the thread that merges a record, holds the same spinlocks, as the previous LockInsHashTbl call that brought the record into the indexlet. It holds these spinlocks until it has merged the record into the range index. By Assumption 2, the concurrent range index provides atomic queries (including inserts). Thus the merge of each record from the indexlet to the range index is atomic.

We now argue a series of lemmas that will help prove that the transactional protocol guarantees serializability.

Lemma 5. *The writes of a transaction in ScaleDB, appear atomically, at transaction commit.*

PROOF. Assume otherwise. Concretely, assume that a transaction T_i does multiple writes, and another transaction T_j does point reads of the records (or phantom indicators) written by T_i, but it misses some of T_i's writes. There are four cases:

(*i*) T_i did not acquire locks on its complete write set before releasing locks on some of its writes. However, this is not possible according to the protocol (§5.2.2.2).

(*ii*) T_i acquired locks on its complete write set, but T_j's LockRUDHashTbl calls, for doing its point reads from the indexlet, did not serialize correctly with T_i's write locks. This is not possible according to Lemmas 2 and 3.

(*iii*) Some (or all) of T_i's writes to the indexlet were merged into the range index. However, according to Lemma 4, merges are atomic with LockUpdDel (and by implication with LockRUDHashTbl). Thus T_j will either find these records in the indexlet (before the merge), or the range index (after the merge).

(*iv*) Some (or all) of T_i's phantom indicators, inserted into the phantomlet, were cleared, before T_j could read them. However, according to the protocol (§5.2.2.3), phantom indicators are only decremented at the end of a merge epoch, after the corresponding inserts have been merged into the primary range index. Thus, T_j will either find a phantom indicator in the phantomlet or the corresponding inserted record in the range index.

Since none of the cases are possible, the assumption does not hold and the lemma is proved.

Lemma 6. *The writes in any schedule of committed transactions in ScaleDB, have the same order, as the writes in the serial schedule, where those transactions are ordered by their commit timestamps.*

PROOF. Recall that ScaleDB allocates unique timestamps for each transaction derived from a system-wide synchronized clock (§5.2.3). WLOG, consider any two transactions T_i and T_j in a schedule of committed transactions. There are two cases:

(i) T_i and T_j do not intersect in their write sets. As a result they do not acquire locks on the same records. This lemma is trivially true in this case.

(i) T_i and T_j intersect in at least one record, in their write sets. Lemmas 1, 2 and 3 prove that transactional lock requests on the same record are serialized in ScaleDB. In addition, Lemma 4 proves that the merging process for a record is serialized w.r.t. these transactional locks. So, in this case, T_i and T_j will be serialized on a lock, on one of the records, in the intersection of their write sets. WLOG, assume that T_i acquires that lock first, while T_j is ordered behind it. T_i will therefore allocate an earlier commit timestamp than T_j, after it has acquired all its locks, and by the structure of ACC (§5.2.2), all its writes will be ordered before all the writes of T_j.

Lemma 7. *For any schedule of committed transactions in ScaleDB, and a transaction T_i in that schedule, a point read done by T_i returns the same result, as it would in the serial schedule, where those transactions are ordered by their* *commit timestamps.*

PROOF. To prove this, we argue the following:

(i) Transactions never do any Intermediate or Aborted Reads [39]. This is clear from the optimistic nature of ACC. Since writes are published atomically at transaction commit, after acquiring all the locks, Intermediate reads are not possible. Similarly, if a transaction decides to abort, it is during the validation stage, *before* any writes have been published: Aborted Reads are impossible as well.

(ii) There do not exist two committed transactions T_a and T_i, s.t. $t_{T_a} < t_{T_i}$ and T_i does a point read on key k, which misses a write by T_a to record r with key k, where t_{T_a} and t_{T_i} are the commit timestamps of T_a and T_i respectively. Assume the contrary. Since $t_{T_a} < t_{T_i}$, there are two cases:

1. T_a commits before T_i begins its read phase. In this case, the only possibility of missing a write is if T_i's read executes concurrently with the merging of r from the indexlet to the primary range index. However, by Lemma 4, merging of r is atomic w.r.t. LockRUDHashTbl on k in the indexlet. So T_i's call to LockRUDHashTbl on k executes either before or after the merge of record r. If it executes before the merge, then it will find r in the indexlet. If it executes after the merge, then it will not find r in the indexlet, but it will find it during the later search of the primary range index. In either case, it cannot miss the write to r, which contradicts our assumption for case (ii).

2. T_a is concurrent with T_i. Since $t_{T_a} < t_{T_i}$, T_a commits first. Therefore, either T_i does its read before T_a's acquisition of a write lock (i.e., LockUniqueInsert or LockUpdDel) on r, or it reads afterwards.

   (a) If T_i reads before, then it will miss the write. However, since $t_{T_a} < t_{T_i}$, the read validation of T_i happens *after* T_a has acquired its write locks. Therefore, T_i's acquisition of LockRUDHashTbl on k, for read validation of r, will be ordered after T_a commits its write to r. As a result, T_i will read the newly written record and its read validation will fail, resulting in an abort. But we assumed that T_i committed, which is a contradiction. So this case is impossible.

   (b) If T_i reads after, then there is no write missed, which contradicts our original assumption for case (*ii*).

Since none of the cases are possible, the lemma is proved.

Lemma 8. *For any schedule of committed transactions in ScaleDB, and a transaction T_i in the schedule, a range scan done by T_i returns the same result, as it would in the serial schedule, where those transactions are ordered by their commit timestamps.*

PROOF. We have already argued in Lemma 7, why aborted and intermediate reads are not possible. So all we need to argue is that, there do not exist two committed transactions T_a and T_i, s.t. $t_{T_a} < t_{T_i}$, and T_i does a range scan with predicate p which matches key k, and T_i's range scan misses an insert by T_a of a record r with key k. Assume the contrary. Since $t_{T_a} < t_{T_i}$, there are two cases:

(i) T_a commits before T_i begins its read phase. There are two sub-cases:

1. T_a's insert is merged into the range index *before* T_i performs its range scan. In this, T_i must have read k, since range scans are executed directly on the range index. But that contradicts the assumption. Hence this case is impossible.

2. T_a's insert is merged into the range index *after* T_i performs its range scan. In this case, the inserted record r is in the indexlet, at the time T_i performs its range scan. Thus, T_i will miss r in its range scan, during its read phase. Assume that the key k of r is covered by the range index leaf node l and that the version of l, at the time T_i performed its range scan, is v. WLOG, we assume that T_a is the last committed transaction to insert a key into the range covered by l, prior to the range scan by T_i. From Lemma 5, we know that the phantom indicator $< addr(l), v >$ must be atomically inserted (or incremented) into the phantomlet, along with the insert of r, at T_a's commit. Thus, T_i will find this phantom indicator in the phantomlet when it validates its transaction, resulting in an abort. However, we assumed at the start of this proof that T_i commits, which is a contradiction. Hence, this case is impossible.

3. T_a's insert is merged into the range index *concurrently* with T_i's range

scan. From Lemma 4, merging of a record r is atomic w.r.t. queries on the range index. So T_i either reads before the merge, which is the same as case 1, or it reads after the merge, which is the same as case 2. Hence, this case is impossible as well.

(ii) T_a is concurrent with T_i. Since $t_{T_a} < t_{T_i}$, T_a commits first. T_i searches the phantomlet, for the phantom indicator $< addr(l), v >$, when validating the range scan; it either does this *before* T_a's acquisition of a LockUniqueInsert on $< addr(l), v >$ in the phantomlet, or *afterwards*.

1. If T_i searches for the phantom indicator before, then it will miss it. However, since T_i is in its validation stage at this point, it must have already acquired its write locks and allocated its commit timestamp t_{T_i}. On the other hand, by assumption, T_a has not yet acquired a LockUniqueInsert on the phantom indicator. This implies that $t_{T_i} < t_{T_a}$, contradicting our initial assumption that $t_{T_a} < t_{T_i}$. Hence, this case is impossible.

2. If T_i searches for the phantom indicator after, then it will find it and it will abort. However, we assumed at the start of the proof that T_i commits, resulting in a contradiction. Hence this case is impossible as well.

Since both cases result in contradictions, the lemma is proved.

Theorem 1. *ScaleDB guarantees that any schedule of committed transactions, is equivalent to the serial schedule, where those transactions are ordered* *by their commit timestamps.*

PROOF. This follows directly from Lemmas 5, 6, 7 and 8.

Bibliography

[1] Case study: How customers are using memsql for free. https://www.memsql.com/blog/case-stduy-how-customers-are-using-memsql-for-free/.

[2] CloudLab. https://www.cloudlab.us/.

[3] Cuckoo Hashing. https://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/13/Small13.pdf.

[4] Driver porting: mutual exclusion with seqlocks. https://lwn.net/Articles/22818.

[5] F14 Hash Table: Why Probing. https://github.com/facebook/folly/blob/master/folly/container/F14.md#why-probing.

[6] How LinkedIn customizes Apache Kafka for 7 trillion messages per day. https://engineering.linkedin.com/blog/2019/apache-kafka-trillion-messages.

[7] HyPer – A Hybrid OLTP&OLAP High Performance DBMS. https://hyper-db.de/.

[8] IDC: Expect 175 zettabytes of data worldwide by 2025. https://www.networkworld.com/article/3325397/idc-expect-175-zettabytes-of-data-worldwide-by-2025.html.

[9] ISO/IEC 9075-1:2016 Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework). https://www.iso.org/standard/63555.html.

[10] Jedis. https://github.com/xetorthio/jedis.

[11] Key-value Database. https://en.wikipedia.org/wiki/Key-value_database.

[12] libcuckoo github repository. https://github.com/efficient/libcuckoo.

[13] Linux TSC Cross Socket Reliability. https://github.com/torvalds/linux/blob/c2131f7e73c9e9365613e323d65c7b9e5b910f56/arch/x86/kernel/cpu/intel.c#L249.

[14] Linux TSC Synchronization. https://github.com/torvalds/linux/blob/master/arch/x86/kernel/tsc_sync.c.

[15] Manhattan, our real-time, multi-tenant distributed database for twitter scale. https://blog.twitter.com/engin
our-real-time-multi-tenant-distributed-database-for-twitter-scale.

[16] MemSQL. https://www.memsql.com/.

[17] MySQL 8.0 Reference Manual: Clustered and Secondary Indexes. https://dev.mysql.com/doc/refman/8.0/en/innodb-index-types.html.

[18] Network Time Protocol. https://www.eecis.udel.edu/~mills/ntp.html.

[19] Oltp-bench github repository. https://github.com/oltpbenchmark/oltpbench.

[20] Redis Cluster Specification. http://redis.io/topics/cluster-spec.

[21] Resizing Hash Tables. https://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf.

[22] SAP HANA. https://www.sap.com/products/hana.html.

[23] SQL. https://en.wikipedia.org/wiki/SQL.

[24] Strong consistency in Manhattan. https://blog.twitter.com/engineering/en_us/a/2016/strong-consistency-in-manhattan.

[25] The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm.

[26] The Forrester Wave^{TM}: In-Memory Databases, Q1 2017. http://www.oracle.com/us/corporate/analystsreports/forrester-imdb-wave-2017-3616348.pdf.

[27] The Infrastructure Behind Twitter: Scale. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.

[28] The Peloton self-driving SQL database management system. https://github.com/cmu-db/peloton.

[29] Time-series data: Why (and how) to use a relational database instead of NoSQL. .

[30] Timekeeping in VMware Virtual Machines. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf.

[31] TPC-C Benchmark. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[32] VoltDB. https://www.oracle.com/database/technologies/related/timesten.html.

[33] VoltDB. https://www.voltdb.com/.

[34] What is a key-value database? https://redis.com/nosql/key-value-databases/.

[35] What is an in-memory database? https://aws.amazon.com/nosql/in-memory.

[36] What is RCU, Fundamentally? https://lwn.net/Articles/262464/.

[37] Xen TSC (time stamp counter) and timekeeping discussion. http://xenbits.xen.org/docs/4.13-testing/man/xen-tscmode.7.html.

[38] ABUZAID, F., BAILIS, P., DING, J., GAN, E., MADDEN, S., NARAYANAN, D., RONG, K., AND SURI, S. Macrobase: Prioritizing attention in fast data. *ACM Trans. Database Syst. 43*, 4 (Dec. 2018), 15:1–15:45.

[39] ADYA, A. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, MIT, 1999.

[40] ADYA, A., AND LISKOV, B. Lazy Consistency Using Loosely Synchronized Clocks. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing* (Santa Barbara, California, USA, 1997), PODC '97, ACM, pp. 73–82.

[41] AHAMAD, M., NEIGER, G., BURNS, J., KOHLI, P., AND HUTTO, P. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing 9*, 1 (1995), 37–49.

[42] AJOUX, P., BRONSON, N., KUMAR, S., LLOYD, W., AND VEERARAGHAVAN, K. Challenges to Adopting Stronger Consistency at Scale. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland, 2015), HOTOS'15, USENIX Association.

[43] AKKOORATH, D. D., TOMSIC, A. Z., BRAVO, M., LI, Z., CRAIN, T., BIENIUSA, A., PREGUIÇA, N., AND SHAPIRO, M. Cure: Strong Semantics Meets High Availability and Low Latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)* (June 2016), pp. 405–414.

[44] Almeida, S., Leitão, J. a., and Rodrigues, L. Chainreaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic, 2013), EuroSys '13, ACM, pp. 85–98.

[45] Amble, O., and Knuth, D. E. Ordered hash tables. *The Computer Journal 17*, 2 (01 1974), 135–142.

[46] Ardekani, M. S., and Terry, D. B. A Self-Configurable Geo-Replicated Cloud Storage System. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), OSDI '14, USENIX Association, pp. 367–381.

[47] Arulraj, J., Perron, M., and Pavlo, A. Write-behind logging. *Proc. VLDB Endow. 10*, 4 (Nov. 2016), 337–348.

[48] Babaoğlu, O., and Marzullo, K. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In *Distributed Systems (2nd Ed.)*, S. Mullender, Ed. ACM Press/Addison-Wesley Publishing Co., 1993, pp. 55–96.

[49] Bacon, D. F., Bales, N., Bruno, N., Cooper, B. F., Dickinson, A., Fikes, A., Fraser, C., Gubarev, A., Joshi, M., Kogan, E., Lloyd, A., Melnik, S., Rao, R., Shue, D., Taylor, C., van der Holst, M., and Woodford, D. Spanner: Becoming a sql system. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 331–343.

[50] Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow. 7*, 3 (Nov. 2013), 181–192.

[51] Bailis, P., Fekete, A., Franklin, M. J., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Coordination avoidance in database systems. *Proc. VLDB Endow. 8*, 3 (Nov. 2014), 185–196.

[52] Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J. M., and Stoica, I. The Potential Dangers of Causal Consistency and an Explicit Solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing* (San Jose, California, 2012), SoCC '12, ACM, pp. 22:1–22:7.

[53] Bailis, P., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Bolt-On Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, 2013), SIGMOD '13, ACM, pp. 761–772.

[54] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, USA, 1995), SIGMOD '95, ACM, pp. 1–10.

[55] Bernstein, P., and Newcomer, E. *Principles of Transaction Processing: For the Systems Professional.* Morgan Kaufmann Publishers Inc., 1997.

[56] Bernstein, P. A., and Goodman, N. Multiversion concurrency control&mdash;theory and algorithms. *ACM Trans. Database Syst. 8*, 4 (Dec. 1983), 465–483.

[57] Bhat, S. S., Eqbal, R., Clements, A. T., Kaashoek, M. F., and Zeldovich, N. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 69–86.

[58] Birman, K., Chockler, G., and van Renesse, R. Toward a Cloud Computing Research Agenda. *SIGACT News 40*, 2 (June 2009), 68–80.

[59] Bolosky, W. J., and Scott, M. L. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4* (USA, 1993), Sedms'93, USENIX Association, p. 3.

[60] Boyd-Wickizer, S., Kaashoek, M. F., Morris, R., and Zeldovich, N. Oplog: a library for scaling update-heavy data structures. Tech. Rep. MIT-CSAIL-TR-2014-019, MIT, CSAIL, September 2014.

[61] Bravo, M., Rodrigues, L., and Van Roy, P. Saturn: a distributed metadata service for causal consistency. In *Proceedings of the 12th ACM European Conference on Computer Systems* (2017), EuroSys '17, ACM.

[62] Brewer, E. Cap twelve years later: How the "rules" have changed. *Computer 45*, 2 (2012), 23–29.

[63] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., Marchukov, M., Petrov, D., Puzar, L., Song, Y. J., and Venkataramani, V. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual*

Technical Conference (San Jose, CA, 2013), USENIX ATC'13, USENIX Association, pp. 49–60.

[64] Bronson, N., and Shi, X. Open-sourcing F14 for faster, more memory-efficient hash tables. https://engineering.fb.com/developer-tools/f14/.

[65] Bronson, N. G., Casper, J., Chafi, H., and Olukotun, K. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, Association for Computing Machinery, pp. 257–268.

[66] Cha, S. K., Hwang, S., Kim, K., and Kwon, K. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *Proceedings of the 27th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 2001), VLDB '01, Morgan Kaufmann Publishers Inc., pp. 181–190.

[67] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA, 2006), OSDI '06, USENIX Association, pp. 15–15.

[68] Chattopadhyay, B., Mittal, S., Ebenstein, R., Mikhaylin, N., Lee, H.-c., Zhao, X., Xu, T., Perez, L., Shahmohammadi, F., Bui, T., McKay, N., Dutta, P., Aya, S., Lychagina, V., Elliott, B., Liu, W., Tinn, O., Mccormick, A., Mokashi, A., and Lomax, D. Procella: unifying serving and

analytical data at youtube. *Proceedings of the VLDB Endowment 12* (08 2019), 2022–2034.

[69] Cheng, A., Shi, X., Pan, L., Simpson, A., Wheaton, N., Lawande, S., Bronson, N., Bailis, P., Crooks, N., and Stoica, I. RAMP-TAO. *Proceedings of the VLDB Endowment 14*, 12 (jul 2021), 3014–3027.

[70] Clements, A. T., Kaashoek, M. F., Zeldovich, N., Morris, R. T., and Kohler, E. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst. 32*, 4 (Jan. 2015).

[71] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment 1*, 2 (Aug. 2008), 1277–1288.

[72] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, 2010), SoCC '10, ACM, pp. 143–154.

[73] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, 2010), SoCC '10, ACM, pp. 143–154.

[74] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kan-

thak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[75] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009, ch. 11.

[76] Crooks, N., Pu, Y., Alvisi, L., and Clement, A. Seeing is believing: A client-centric specification of database isolation. PODC '17, Association for Computing Machinery, pp. 73–82.

[77] Crooks, N., Pu, Y., Estrada, N., Gupta, T., Alvisi, L., and Clement, A. TARDiS: A Branch-and-Merge Approach to Weak Consistency. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data* (San Francisco, California, 2016), SIGMOD '16, ACM, pp. 1615–1628.

[78] Davidson, S. B., Garcia-Molina, H., and Skeen, D. Consistency in a partitioned network: a survey. *ACM Computing Surveys 17*, 3 (sep 1985), 341–370.

[79] Dean, J., and Barroso, L. A. The Tail at Scale. *Communications of the ACM 56*, 2 (Feb. 2013), 74–80.

[80] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 2004), OSDI'04, USENIX Association, pp. 137–149.

[81] Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N., and Zwilling, M. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 1243–1254.

[82] Difallah, D. E., Pavlo, A., Curino, C., and Cudre-Mauroux, P. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow. 7*, 4 (Dec. 2013), 277–288.

[83] Du, J., Elnikety, S., Roy, A., and Zwaenepoel, W. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proceedings of the 4th ACM Symposium on Cloud Computing* (Santa Clara, California, 2013), SOCC '13, ACM, pp. 11:1–11:14.

[84] Du, J., Iorgulescu, C., Roy, A., and Zwaenepoel, W. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *Proceedings of the 5th ACM Symposium on Cloud Computing* (2014), SOCC '14, ACM.

[85] Faleiro, J. M., and Abadi, D. J. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow. 8*, 11 (July 2015), 1190–1201.

[86] Faleiro, J. M., and Abadi, D. J. Latch-free synchronization in database systems: Silver bullet or fool's gold? In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings* (2017), www.cidrdb.org, p. 9.

[87] Fidge, C. J. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference (ACSC'88)* (February 1988), pp. 56–66.

[88] Fomitchev, M., and Ruppert, E. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2004), PODC '04, Association for Computing Machinery, pp. 50–59.

[89] Gilbert, S., and Lynch, N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News 33*, 2 (June 2002), 51–59.

[90] Graefe, G. A survey of b-tree locking techniques. *ACM Trans. Database Syst. 35*, 3 (July 2010), 16:1–16:26.

[91] Gunawi, H. S., Suminto, R. O., Sears, R., Golliher, C., Sundararaman, S., Lin, X., Emami, T., Sheng, W., Bidokhti, N., McCaffrey, C., Grider, G., Fields, P. M., Harms, K., Ross, R. B., Jacobson, A., Ricci, R., Webb, K., Alvaro, P., Runesha, H. B., Hao, M., and Li, H. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)* (Oakland, CA, Feb. 2018), USENIX Association, pp. 1–14.

[92] Hellerstein, J. M., and Alvaro, P. Keeping CALM: when distributed consistency is easy. *CoRR abs/1901.01930* (2019).

[93] Herlihy, M., Lev, Y., Luchangco, V., and Shavit, N. A simple optimistic skiplist algorithm. In *Proceedings of the 14th International Conference on Structural*

Information and Communication Complexity (Berlin, Heidelberg, 2007), SIROCCO'07, Springer-Verlag, pp. 124–138.

[94] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[95] Huang, P., Guo, C., Zhou, L., Lorch, J. R., Dang, Y., Chintalapati, M., and Yao, R. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2017), HotOS '17, Association for Computing Machinery, pp. 150–155.

[96] Huang, Y., Qian, W., Kohler, E., Liskov, B., and Shrira, L. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *Proc. VLDB Endow. 13*, 5 (Jan. 2020), 629–642.

[97] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, vol. 3B. Nov 2018, ch. 17, pp. 17–41.

[98] Israeli, A., and Rappoport, L. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing - PODC '94* (1994), ACM Press.

[99] Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y. J., Wang, Z., Xu, Y., Dulloor, S. R., Zhao, J., and Swanson, S. Basic performance measurements of the intel optane DC persistent memory module. *CoRR abs/1903.05714* (2019).

[100] Jankov, D., Luo, S., Yuan, B., Cai, Z., Zou, J., Jermaine, C., and Gao, Z. J. Declarative recursive computation on an RDBMS. *Proceedings of the VLDB*

Endowment 12, 7 (mar 2019), 822–835.

[101] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. H-store. *Proceedings of the VLDB Endowment 1*, 2 (aug 2008), 1496–1499.

[102] Kim, K., Wang, T., Johnson, R., and Pandis, I. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, Association for Computing Machinery, pp. 1675–1687.

[103] Kimura, H. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, Association for Computing Machinery, pp. 691–706.

[104] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems 27*, 4 (Jan. 2010), 7:1–7:39.

[105] Kraska, T., Pang, G., Franklin, M. J., Madden, S., and Fekete, A. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic, 2013), EuroSys '13, ACM, pp. 113–126.

[106] Krikorian, R. Twitter Timelines at Scale (video link. consistency discussion at 26m). http://www.infoq.com/presentations/Twitter-Timeline-Scalability,

2013.

[107] Kung, H. T., and Robinson, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.

[108] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[109] Lamport, L. The Part-Time Parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[110] Lamport, L. Generalized Consensus and Paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research, 2004.

[111] Larson, P.-r., Blanas, S., Diaconu, C., Freedman, C., Patel, J. M., and Zwilling, M. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow. 5*, 4 (Dec. 2011), 298–309.

[112] Lee, C., Park, S. J., Kejriwal, A., Matsushita, S., and Ousterhout, J. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 71–86.

[113] Lehman, P. L., and Yao, s. B. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst. 6*, 4 (Dec. 1981), 650–670.

[114] Leis, V., Scheibner, F., Kemper, A., and Neumann, T. The art of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (New York, NY, USA, 2016), DaMoN '16, ACM, pp. 3:1–3:8.

[115] Levandoski, J. J., Lomet, D. B., and Sengupta, S. The bw-tree: A b-tree for new hardware platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)* (Washington, DC, USA, 2013), ICDE '13, IEEE Computer Society, pp. 302–313.

[116] Li, J., Chen, Y., Liu, H., Lu, S., Zhang, Y., Gunawi, H. S., Gu, X., Lu, X., and Li, D. Pcatch: Automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, Association for Computing Machinery.

[117] Lim, H., Kaminsky, M., and Andersen, D. G. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 21–35.

[118] Lipton, R. J., and Sandberg, J. PRAM: A Scalable Shared Memory. Tech. Rep. TR-180-88, Princeton University, Department of Computer Science, August 1988.

[119] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), SOSP '11, ACM, pp. 401–416.

[120] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL, 2013), NSDI '13, USENIX Association, pp. 313–328.

[121] Lu, H., Hodsdon, C., Ngo, K., Mu, S., and Lloyd, W. The SNOW Theorem and Latency-Optimal Read-Only Transactions. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), OSDI'16, USENIX Association, pp. 135–150.

[122] Lu, H., Veeraraghavan, K., Ajoux, P., Hunt, J., Song, Y. J., Tobagus, W., Kumar, S., and Lloyd, W. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 295–310.

[123] Mahajan, P., Alvisi, L., and Dahlin, M. Consistency, Availability, and Convergence. Tech. Rep. UTCS TR-11-22, Department of Computer Science, The University of Texas at Austin, 2011.

[124] Mao, Y., Kohler, E., and Morris, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 183–196.

[125] Mattern, F. Virtual Time and Global States of Distributed Systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms* (1989), North-Holland/Elsevier, pp. 215–226.

[126] Mehdi, S. A., Littley, C., Crooks, N., Alvisi, L., Bronson, N., and Lloyd, W. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 453–468.

[127] Mickens, J., Nightingale, E. B., Elson, J., Gehring, D., Fan, B., Kadav, A., Chidambaram, V., Khan, O., and Nareddy, K. Blizzard: Fast, Cloud-Scale Block Storage for Cloud-Oblivious Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), USENIX Association, pp. 257–273.

[128] Mohan, C., and Levine, F. E. Aries/im: An efficient and high concurrency index management method using write-ahead logging. In *SIGMOD Conference* (1992).

[129] Mu, S., Angel, S., and Shasha, D. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[130] Mu, S., Cui, Y., Zhang, Y., Lloyd, W., and Li, J. Extracting More Concurrency from Distributed Transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO, 2014), OSDI'14, USENIX Association, pp. 479–494.

[131] Mu, S., Nelson, L., Lloyd, W., and Li, J. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), OSDI'16, USENIX Association.

[132] Neumann, T., Mühlbauer, T., and Kemper, A. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015*

ACM SIGMOD International Conference on Management of Data (New York, NY, USA, 2015), SIGMOD '15, Association for Computing Machinery, pp. 677–689.

[133] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the Sync. *ACM Transactions on Computer Systems 26*, 3 (Sept. 2008), 6:1–6:26.

[134] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL, 2013), NSDI '13, USENIX Association, pp. 385–398.

[135] Pavlo, A. SIGMOD keynote: What are we doing with our lives?: Nobody cares about our concurrency control research. https://www.youtube.com/watch?v=M2MEcvMHzkY&t=3525s, May 2017.

[136] Pavlo, A. What are we doing with our lives? In *Proceedings of the 2017 ACM International Conference on Management of Data* (May 2017), ACM.

[137] Pavlo, A., Curino, C., and Zdonik, S. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 international conference on Management of Data - SIGMOD '12* (2012), ACM Press.

[138] Ports, D. R., Li, J., Liu, V., Sharma, N. K., and Krishnamurthy, A. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015), USENIX Association, pp. 43–57.

[139] Qiao, L., Surlaker, K., Das, S., Quiggle, T., Schulman, B., Ghosh, B., Curtis, A., Seeliger, O., Zhang, Z., Auradar, A., Beaver, C., Brandt, G., Gandhi, M., Gopalakrishna, K., Ip, W., Jgadish, S., Lu, S., Pachev, A., Ramesh, A., Sebastian, A., Shanbhag, R., Subramaniam, S., Sun, Y., Topiwala, S., Tran, C., Westerman, J., and Zhang, D. On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, 2013), SIGMOD '13, ACM, pp. 1135–1146.

[140] Ricci, R., Eide, E., and The CloudLab Team. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *USENIX ;login: 39*, 6 (Dec. 2014).

[141] Schneider, F. B. Replication Management Using the State-Machine Approach. In *Distributed Systems (2nd Ed.)*, S. Mullender, Ed. ACM Press/Addison-Wesley Publishing Co., 1993, pp. 169–197.

[142] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., and Mangard, S. Malware guard extension: Using sgx to conceal cache attacks, 2019.

[143] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Tech. Rep. HAL Id: inria-00555588, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[144] Shi, X., Pruett, S., Doherty, K., Han, J., Petrov, D., Carrig, J., Hugg, J., and Bronson, N. Flighttracker: Consistency across read-optimized online stores

at facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 407–423.

[145] Sovran, Y., Power, R., Aguilera, M. K., and Li, J. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 385–400.

[146] Sovran, Y., Power, R., Aguilera, M. K., and Li, J. Transactional Storage for Geo-Replicated Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), SOSP '11, ACM, pp. 385–400.

[147] Tallent, N. R., Mellor-Crummey, J. M., and Porterfield, A. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, Association for Computing Machinery, pp. 269–280.

[148] Terry, D. B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M. K., and Abu-Libdeh, H. Consistency-based service level agreements for cloud storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 309–324.

[149] Terry, D. B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M. K., and Abu-Libdeh, H. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farmington, Pennsylvania, 2013), SOSP '13, ACM, pp. 309–324.

[150] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In

Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (2012), ACM, pp. 1–12.

[151] Torres-Rojas, F. J., and Ahamad, M. Plausible clocks: Constant size logical clocks for distributed systems. *Distributed Computing 12*, 4 (Sept. 1999), 179–195.

[152] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 18–32.

[153] Van Aken, D., Yang, D., Brillard, S., Fiorino, A., Zhang, B., Bilien, C., and Pavlo, A. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proc. VLDB Endow. 14*, 7 (mar 2021), 1241–1253.

[154] van Renesse, R., and Schneider, F. B. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 2004), OSDI'04, USENIX Association, pp. 91–104.

[155] Vogels, W. Eventually Consistent. *Commun. ACM 52*, 1 (Jan. 2009), 40–44.

[156] Wang, Z., Mu, S., Cui, Y., Yi, H., Chen, H., and Li, J. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, Association for Computing Machinery, pp. 1643–1658.

[157] Wang, Z., Pavlo, A., Lim, H., Leis, V., Zhang, H., Kaminsky, M., and Andersen, D. G. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 ACM International Conference on Management of Data* (2018), SIGMOD '18, pp. 473–488.

[158] Wei, X., Shi, J., Chen, Y., Chen, R., and Chen, H. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 87–104.

[159] Whittaker, M., and Hellerstein, J. M. Checking invariant confluence, in whole or in parts. *SIGMOD Rec. 49*, 1 (sep 2020), 7–14.

[160] Wu, C., Faleiro, J., Lin, Y., and Hellerstein, J. Anna: A KVS for any scale. *IEEE Transactions on Knowledge and Data Engineering* (2019), 1–1.

[161] Xie, C., Su, C., Littley, C., Alvisi, L., Kapritsos, M., and Wang, Y. High-Performance ACID via Modular Concurrency Control. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 279–294.

[162] Yang, J., Yue, Y., and Rashmi, K. V. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 191–208.

[163] Yu, X., Bezerra, G., Pavlo, A., Devadas, S., and Stonebraker, M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc.*

VLDB Endow. 8, 3 (Nov. 2014), 209–220.

[164] Yu, X., Pavlo, A., Sanchez, D., and Devadas, S. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 1629–1642.

[165] Zawirski, M., Preguiça, N., Duarte, S., Bieniusa, A., Balegas, V., and Shapiro, M. Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In *Proceedings of the 16th Annual Middleware Conference* (Vancouver, BC, Canada, 2015), Middleware '15, ACM, pp. 75–87.

[166] Zhang, I., Sharma, N. K., Szekeres, A., Krishnamurthy, A., and Ports, D. R. K. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 263–278.

[167] Zhang, Y., Power, R., Zhou, S., Sovran, Y., Aguilera, M. K., and Li, J. Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farmington, Pennsylvania, 2013), SOSP '13, ACM, pp. 276–291.