usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

ScaleDB: A Scalable, Asynchronous In-Memory Database

Syed Akbar Mehdi, *The University of Texas at Austin;* Deukyeon Hwang and Simon Peter, *University of Washington;* Lorenzo Alvisi, *Cornell University*

https://www.usenix.org/conference/osdi23/presentation/mehdi

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

ScaleDB: A Scalable, Asynchronous In-Memory Database

Syed Akbar Mehdi
The University of Texas at Austin*

Deukyeon Hwang
University of Washington

Simon Peter
University of Washington

Lorenzo Alvisi
Cornell University

Abstract

ScaleDB is a serializable in-memory transactional database that achieves excellent scalability on multi-core machines by asynchronously updating range indexes. We find that asynchronous range index updates can significantly improve database scalability by applying updates in batches, reducing contention on critical sections. To avoid stale reads, ScaleDB uses small hash *indexlets* to hold delayed updates. We use indexlets to design ACC, an asynchronous concurrency control protocol providing serializability. With ACC, it is possible to delay range index updates without adverse performance effects on transaction execution in the common case. ACC delivers scalable serializable isolation for transactions, with high throughput and low abort rate. Evaluation on a dual-socket server with 36 cores shows that ScaleDB achieves 9.5× better query throughput than Peloton on the YCSB benchmark and 1.8× better transaction throughput than Cicada on the TPC-C benchmark.

1 Introduction

In-memory databases [5, 10, 15, 21, 23, 42] are becoming increasingly popular: they perform well under a wide range of workloads and support requirements, such as real-time constraints, that are challenging for their disk-based counterparts [16]. They also, however, face scalability demands that sharding can only partially address: many real-world workloads have skewed access distributions [30, 47, 50, 68, 75], and the frequent hotspots they generate in individual shards require database solutions that can scale on multi-core servers.

Unfortunately, despite years of research, scaling in-memory databases on multi-core architectures remains challenging. Existing work [62, 73, 79] eliminates the bottleneck on a shared timestamp in the concurrency control protocol. Other work [37, 38, 60, 63, 77] has focused on improving the scalability of indexing structures in isolation from the database architecture. Nonetheless, current databases scale poorly on multi-core architectures (§3.1). In particular, shared range-index structures (e.g., B$^+$ trees) continue to be a main source of contention [77], and the high cost of updates to these indexes, even by unrelated transactions, is a major factor limiting scalability [62]. As fast storage via solid-state drives and persistent memory becomes the norm, contention on these structures is intensifying.

*Currently at Google. Work done during PhD at UT Austin.

We believe that continuing to scale with these application and hardware trends requires a fresh approach. Our main observation, supported by recent work in file systems [34, 35], is that contention on shared data structures is often not fundamental, but simply an artifact of a particular system architecture. In particular, we find that contention caused by synchronous updates to sorted range-index structures is unnecessary in the common case. Our analysis (§3.2) shows that it is possible to delay many common range-index updates, without compromising on strong consistency guarantees or latency requirements for transactions. Delayed updates may be batched to reduce contention on shared range-index structures.

These observations lead us to propose a decoupled database design, centered around minimizing *unnecessary contention* among unrelated transactions. Our main technique is to decouple committing a transaction from updating the affected range indexes: we update range indexes *asynchronously*, while using scalable hash-based *indexlets* to track writes of recently committed transactions. Based on this asynchronous architecture, 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[1] detection [32]. ACC uses locks in indexlets, rather than in range indexes, to provide scalable atomic transaction commit.

We present ScaleDB, a scalable multi-core in-memory transactional database based on asynchronous concurrency control. 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. By avoiding unnecessary contention on shared data structures in the common case, ScaleDB delivers scalable serializable isolation for ACID transactions, with high throughput, low commit latency, and low abort rate.

We make the following contributions:

- An analysis of the range index scalability bottleneck and of asynchronous range-index updates as a way to alleviate that bottleneck for unrelated transactions (§3).

[1] *Phantom* anomalies arise when insertions or deletions by other concurrently committing transactions cause two identical range scans in the same transaction to return a different set of rows.

- The design (§4) and implementation (§5) 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.
- Asynchronous concurrency control (ACC), a novel concurrency control protocol that provides serializability in an asynchronous database (§4.2). ACC uses *phantomlets* to scalably detect phantoms in range scans and provides scalable locks in indexlets to atomically commit transactions.
- A performance evaluation of ScaleDB on a dual-socket server with 36 cores, which shows that ScaleDB scales better than Cicada and Peloton. At scale, ScaleDB achieves 9.5× better query throughput than Peloton on the YCSB benchmark and 1.8× better transaction throughput than Cicada with shared indexes on the TPC-C benchmark.

2 Background

Modern relational databases face challenging scalability demands. In addition to serving as backends for large-scale web applications [11, 17, 25, 31], they are offered as a service in public clouds [1, 4, 74], and must support applications that can be simultaneously write and read intensive; require both low transaction commit latency and high transactional throughput; and, increasingly, run analytical queries (on data from sources such as sensors, real-time analytics, and machine learning [19, 27]) that require maintaining a large number of indexes on every write.

In-memory databases are particularly suited to handle these diverse workload requirements, and their adoption is further facilitated by high-capacity non-volatile and disaggregated memories, as they allow for more data to be held in memory, with access latencies comparable to DRAM [53, 72]. Just as new memory technologies are shifting performance bottlenecks away from storage and towards multi-core CPU contention, such diverse workload requirements raise the bar for in-memory database scalability.

2.1 Prior Work

Existing efforts to improve the scalability of in-memory databases have focused on three bottlenecks: (*i*) range index structures; and serializable transaction isolation for both (*ii*) low-contention workloads and (*iii*) high-contention workloads (*i.e.,* transactions with dependencies). The work on range index structures has happened in isolation from the rest. This observation is key to the case for ScaleDB (§3).

Range index structures. 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. Despite decades of work [37, 38, 48, 59–61, 63, 65], scalability of range indexes under concurrent accesses remains elusive. This is primarily due to the hierarchical nature of these data structures. For instance, in a B^+tree index, inserting or deleting a new record can require modifying a chain of internal nodes all the way up to the root. Performing such modifications atomically while supporting concurrent access from multiple threads requires synchronization [45, 77].

One approach to synchronization uses locks [48, 59, 65]. Recent optimizations [37, 38, 63] 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) become hotspots of contention.

An alternative are lock-free data structures [46, 49, 61]: they use atomic operations and multi-versioning to avoid lock contention on critical sections. Yet, as recent work [45] points out, their theoretical guarantees are "mostly irrelevant to performance and scalability on multi-core hardware", as they cannot avoid contention on global memory locations.

A recent study [77] evaluated state-of-the-art range indexes [46, 48, 60, 61, 63] on the YCSB [40] 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%. The limited scalability of range indexes has been reported in previous work [62] and we expand on this analysis in §3.1.

Serializability for low-contention workloads. The use of a shared timestamp for ordering transactions [33, 57] made timestamp allocation a principal bottleneck to the scalability of concurrency control [78]. Even when updated using atomic hardware primitives, a shared timestamp can force *unrelated transactions* to contend and results in excessive cache coherence communication. Recent work eliminates the timestamp bottleneck, but incurs high transaction commit latency due to either a high abort rate [62, 79] or batching in group commits [73].

An approach proposed by the H-store project [54, 70] avoids coordination by partitioning the database and accessing each partition from a single thread. This approach scales well for applications whose databases can be cleanly partitioned and where most transactions only access a single partition. However, many applications do not fit this profile and can experience worse performance [69].

Serializability with dependencies. Much work has focused on scaling serializable ordering on *contended* transactional workloads [42, 44, 51, 55, 56, 58, 62, 66, 67, 76]. Serializability requires respecting data dependencies among transactions reading and writing the same database record.

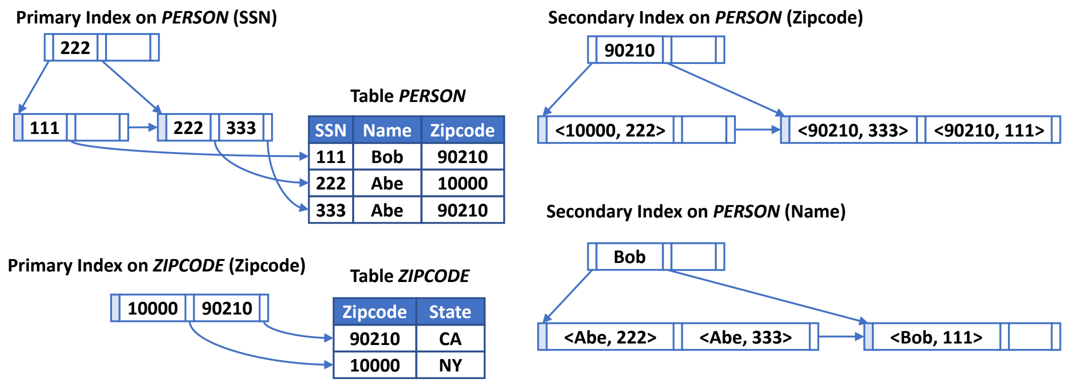
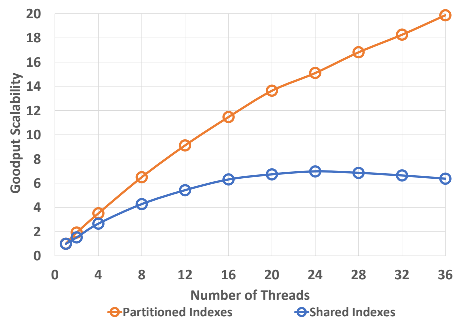
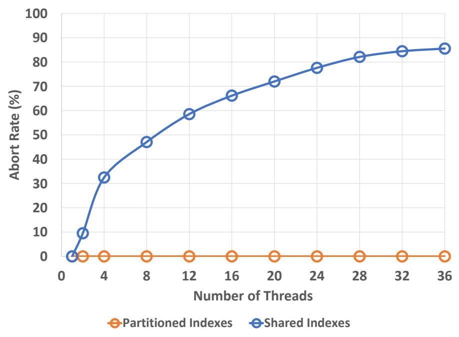


Figure 1. Simple database layout with range indexes. Tables are represented by primary indexes. Records are stored sorted by primary index key. Schema information is stored in a catalog (not shown). Arrows are pointers.



(a) Goodput.



(b) Abort rate.

Figure 2. Cicada scalability on TPC-C ($C_{wh=thd}$) with partitioned and shared indexes.

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 1. Benchmark details.

Though such dependencies are ultimately a barrier to scalability, various techniques can reduce their impact, including multi-versioning [44, 58, 62], static analysis [66, 76], exploiting commutativity in some workloads [51] and backoff [62]. These techniques are complementary to our work, which is focused on *mechanism contention*, *i.e.,* on contention that arises between unrelated transactions as an artifact of how the database implements certain mechanisms (*e.g.,* range indexes), rather than from fundamental requirements of its isolation guarantees.

3 The Case for ScaleDB

ScaleDB's main contribution lies in recognizing that removing the indexing bottleneck requires looking 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.

Range index background. To understand the significance and structure of range indexes, consider Figure 1, which shows 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.

Range indexes have many uses. A primary range index allows quick retrieval of a table's records by primary key for both point and range queries. Primary keys within a table must often be unique and an index can enforce this *uniqueness constraint* efficiently. Secondary indexes are also used extensively. They support analytical queries [39] and help maintain the consistency of the database by serving as *foreign keys*, *i.e.,* columns of a table that refer to 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 deleting 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.

Range indexes can limit scalability because concurrent updates to the same index, even if caused by unrelated transactions, can lead to contention. For instance, inserting or deleting a single record in a B+-tree (Figure 1) may alter its leaf node structure (a leaf node may split, or may coalesce with another leaf node), requiring the atomic update of potentially many internal nodes, all the way to the root.

3.1 Database Scalability Analysis

To explore the limits of database scalability, we evaluate Cicada [62], a state-of-the-art scalable in-memory database. Cicada was shown to be more scalable than several other databases [42, 55, 56, 73, 79]. However, as we show next, it still incurs range index mechanism contention.

We run TPC-C [22], a standard OLTP benchmark simulating purchase transactions on a configurable number of independent warehouses. We use a machine with 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 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 range indexes.

Figure 2a shows Cicada's goodput scalability relative to a single core. Cicada stops scaling beyond 24 cores on the canonical TPC-C workload, with indexes shared between between threads. To show that scalability is limited by range index mechanism contention, we also evaluate a configuration where 8 out of the 9 TPC-C tables, and their associated indexes, are partitioned by warehouse id[2]. This configuration scales well for $C_{wh=thd}$, but it does not generalize to more skewed workloads.

Figure 2b shows that Cicada's poor scalability on the shared index configuration is due to an increasingly high abort rate. To reduce multi-core contention on the same index nodes by multiple threads, Cicada uses multi-version concurrency control (MVCC) for both its records and indexes; 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, to enforce serializability, transactions that perform a range scan must, at commit time, validate that no new record matching the range predicate was inserted since the scan (*i.e.,* must avoid *phantoms*). For this purpose, at transaction commit, Cicada validates all index nodes whose key range intersected with the range scan predicate; if this validation fails, the transaction aborts. Thus, range index contention manifests in Cicada as a higher rate of transaction aborts instead of contention on index nodes.

[2]The default configuration of the Cicada prototype

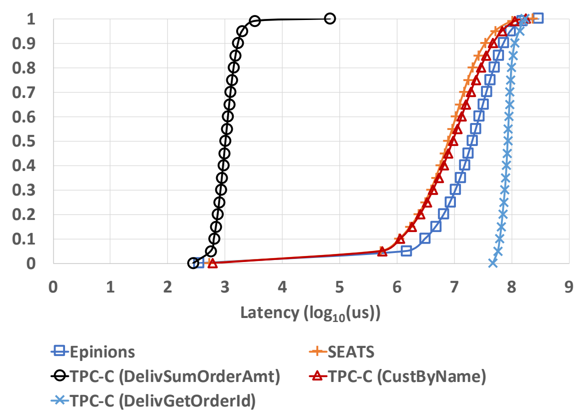
3.2 When Can Range Indexes Scale?

The previous analysis demonstrates that scalability in state-of-the-art databases is primarily limited by contention on range indexes. A key contributor to this contention is that the updates to range indexes, that take place once a transaction commits, are performed *synchronously.* Of course, all indexes, whether range or hash, must, on a query, return the most recently committed record corresponding to an index key, but range indexes have an additional obligation: they must ensure that range scans issued immediately after a transaction commits will not miss any record inserted or updated by that transaction. It is to discharge this obligation that records are inserted synchronously into all primary and secondary indexes – which not only requires sorting these records with respect to *all* records already in the table, but also creates contention on the internal nodes of a range index among otherwise non-conflicting transactions.

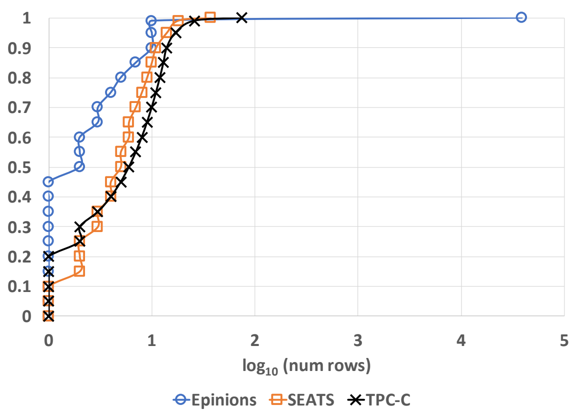
Our design is then motivated by a simple question: *can this obligation be met without triggering a synchronous cascade of updates over shared data structures?* To move towards an answer, we run an experiment to measure the latency between the last time a record is written (inserted or updated) and when it is read as part of a range scan (*W-to-RS latency*).

We use three transactional application benchmarks (Table 1) from the OLTP-bench [43] suite, designed to evaluate modern cloud database workloads. These benchmarks range from moderately write-heavy (Epinions) to very write-heavy (TPC-C), and the percentage of read queries involving a range scan varies from a single digit to 100%. 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 we are never disk-bound.

Figure 3a shows the cumulative distribution of the W-to-RS latency. For Epinions and Seats, we use a single curve each to characterize the behavior of all their range scans: 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× to 5×) gap. The low W-to-RS latency of DelivSumOrderAmt is due to the TPC-C Delivery transaction, which contains an update followed by a read on the same range in the Orderline table. We discuss in §4.2.3 how ScaleDB's design avoids unnecessary aborts in such situations and therefore performs well on TPC-C (§6).



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



(b) Records returned by range scan.

Figure 3. Range scan property distributions of three application benchmarks.

Figure 3b 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 a 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 – from the NewOrder and Payment read-write transactions, which together comprise 90% of the benchmark – read heavily updated records in the District and Warehouse tables. At the other extreme, two point queries in TPC-C had median W-to-PQ latencies of 4 and 15 seconds.

Conclusion. 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. In 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 and, when they do, they are usually within a read-only transaction.

These findings suggest 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. We will see in §4.1 that asynchronous updates are scalable, while relieving the underlying range index structure of fine-grained locking, multi-versioning, and lock-free techniques. This simplifies serializability, as we will see in §4.2.

4 ScaleDB Design

The foundation of ScaleDB's design, building on the analysis in §3, is that range indexes are asynchronously updated to provide scalability. But 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?

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. We tackle this problem with a new data structure: hash-based *indexlets* that temporarily and synchronously record all range index writes. Indexlets leverage the flat structure of hash indexes to avoid contention among updates to unrelated records. A common issue with hash indexes is *rehashing* – resizing the hash index when it is at capacity [14]. Database hash indexes require rehashing, as their size cannot be known a-priori. Instead, indexlets only hold updates temporarily and are periodically merged by ScaleDB into range indexes. Thus, rehashing can be avoided by bounding the maximum number of delayed writes held in an indexlet based on the W-to-RS latency and write rate to the underlying table. We describe indexlets and how to efficiently size and scalably merge them in §4.1.

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 integrates optimistic concurrency control (OCC) [57, 73] with asynchronous range index updates. Both are optimistic approaches: 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 negatively affecting goodput.

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* [32], as well as to ensure that transactions read the most recent value of each key returned by a scan.

ACC's technique for avoiding phantoms relies on *phantom indicators*, which leverage ACC's asynchronous design to scalably indicate 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 transactions into phantom detection indexlets (or *phantomlets*). Maintained for each range index, phantomlets allow range scanning transactions to scalably detect phantoms at commit time. We detail ACC and phantomlets in §4.2.

Durability. To provide durability, ScaleDB uses write-ahead redo logging. Transactions receive a globally-ordered timestamp from a system-wide clock, a hardware feature that industry trends and experimental evidence (§5.4) indicate 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 logs to recovery.

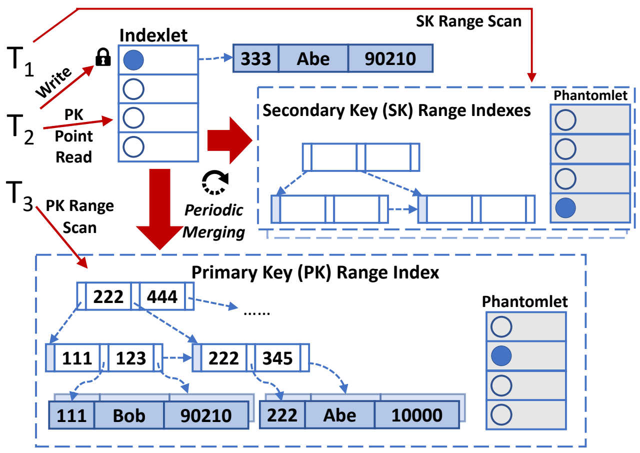


Figure 4. Asynchronous range index update for the *PERSON* table.

Example. To see how it all fits together, consider the example in Figure 4. Transaction T_1 does a range scan by zipcode, which is executed on the appropriate secondary range index. Concurrently, T_2 inserts the record with SSN 333 into the *PERSON* table and does a point read for an SSN from the same table. T_3 does a range scan by SSN, which is executed on the primary range index.

Instead of synchronously updating the range indexes and potentially contending with other transactions, ScaleDB inserts T_2's 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. 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. Range indexes have been read-only, and thus scalable, for this execution.

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. Because merging is periodic, it occurs in a coordinated and concentrated fashion when compared with synchronous range index updates.

Range-scanning transactions consult phantomlets to detect phantoms due to newly inserted records. They do this for each range index leaf node 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.

4.1 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 thread has filled a maximum *batch size* of entries in the indexlet; or (ii) a maximum epoch duration has been reached. Both batch size and maximum epoch duration are configured separately for every indexlet, and each thread decides independently for each indexlet when it has reached the end of its merge epoch.

Indexlets. ScaleDB uses hash-table-based indexlets with open addressing [41] to synchronously and scalably absorb concurrent, committed writes that affect range indexes. Thus, indexlets are associated with tables that have range indexes. For each such table, 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 [12]). The per-table indexlet naturally covers writes that affect secondary indexes, as secondary indexes refer to the primary index (as shown in Figure 1).

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 approach has two benefits: it 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; and it allows coalescing a key deletion followed immediately by an insert of the same key, without merging the delete into the range indexes.

Merge epoch. Each thread independently decides when its merge epoch ends, after which it merges the keys and record references into the table's range indexes. A thread can occupy a maximum *batch size* of b_i entries in any given indexlet i before it has to merge them into the range indexes. Too small a b_i causes contention similar to synchronous merging into range indexes. Too large a b_i results in stale range scans, which can lead to transaction aborts. We use b_i = Expected write rate$(table_i) \times$ W-to-RS latency$(table_i)$.

During quiescent periods for write transactions, threads may not reach their maximum batch size quickly enough, leaving range indexes stale for too long. To avoid this, we cap the length of the merge epoch of each indexlet separately, based on the W-to-RS latency of that indexlet's table: thus, a thread's merge epoch ends when it either reaches its maximum batch size or its maximum merge epoch length.

To make hash collisions rare, the size of indexlet i is set to $s_i = 4 \times \#t \times b_i$, where $\#t$ denotes the number of threads. 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 (§6.3).

Asynchronous merging. Each thread keeps a list of indexlet entries where it performed a write. At the end of its merge epoch, it sorts this list in primary range index key order (§3.2), and then iterates through the list, atomically merging each individual record. Merging involves updating the range index and removing the record from the indexlet, while holding a per-entry lock, thus ensuring atomicity for each key's merge. Each lock is released as soon as the key is merged into the primary index.

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, using these copies.

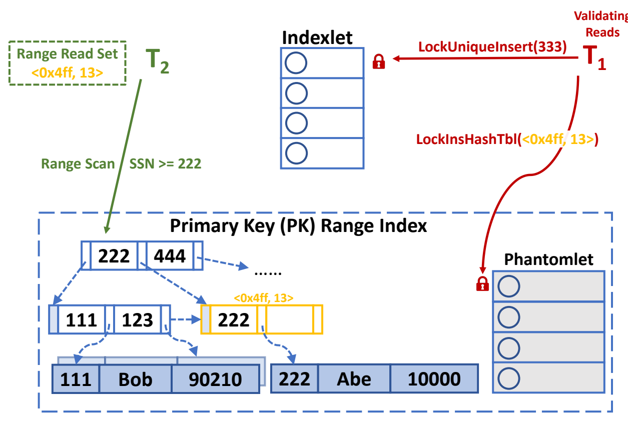
After merging each range index (primary or secondary), the merging thread also decrements any phantom indicators that it had inserted into the corresponding phantomlet during the concluded merge epoch (§4.2.1).

4.2 Asynchronous Concurrency Control

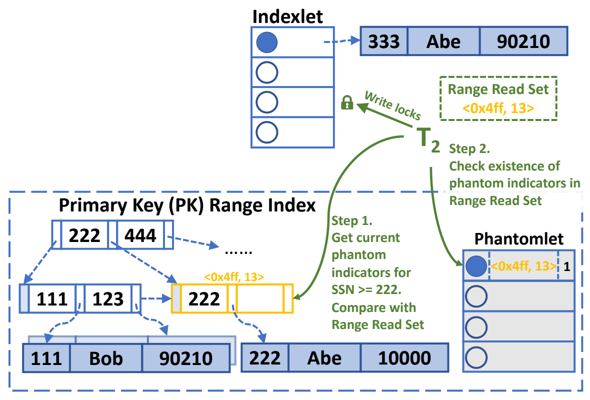
We design asynchronous concurrency control (ACC), a concurrency control protocol that provides serializability within an asynchronous database. ACC is based on optimistic concurrency control (OCC), which it integrates with asynchronous range index updates. To do so, ACC uses two novel constructs: phantom indicators (§4.2.1) and locks in indexlets for atomic commit of transactional writes (§4.2.2).

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 (releasing locks as well).

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



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



(b) 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. Asynchronous phantom detection example.

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.

4.2.1 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* [73]. Phantomlets use the indexlet architecture (§4.1), but do not need merging. Inserting transactions atomically insert *phantom indicators* into phantomlets at transaction commit, indicating that they have inserted a phantom into a corresponding range index leaf node. The phantom indicator is composed of the concatenation of a leaf node l's memory address M_l and version 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. If the phantom indicator does not exist in the phantomlet, it is inserted. If the transaction validates, it atomically increments the value of the phantom indicator (initially 0). This is accomplished by locking phantom indicators as part of locking the transaction's write set (using LockInsHashTbl or LockRUDHashTbl on the phantomlets, see §4.2.2).

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 read, the key of a record r is stored along with a copy t_r^{PS} of the record's current commit timestamp t_r (§4.3) in a *point read set*. Storing the commit timestamp allows efficiently verifying whether the record changed, later during validation. 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. Finally, it 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) 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 (§4.3).
- For each range scan, ACC 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 they have not been merged yet or they were merged but did not result in leaf node splits. ACC goes through each corresponding pair of phantom indicators in c and o, at the same index in the lists, verifying that the pair is identical, and that performing a LockFreeRdHashTbl (§5.2) for this phantom indicator on the phantomlet returns nothing. If any of these checks fail, it aborts.

Figure 5 shows a simple example illustrating asynchronous phantom detection.

4.2.2 Atomic Commit

ACC holds 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 uses 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.

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

1. ACC 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.
   LockInsHashTbl acquires per-entry spinlocks along the hash probe path. If it finds an empty entry, it sets e_{ins}, the future location of the record being inserted, to that entry's index. If the entry is not a search terminator (§5.1), it continues the search for a duplicate record, until the probe lands on a search terminator. If a duplicate is found, all spinlocks are released and the transaction is aborted. Otherwise, LockInsHashTbl is successful. In that case, it 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 LockUniqueInsert is released: this allows atomically inserting a record and updating search termination metadata (see §5.1) at transaction commit. With a properly sized indexlet, probe lengths are short and there is negligible mechanism contention for unique inserts.
2. 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. 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.

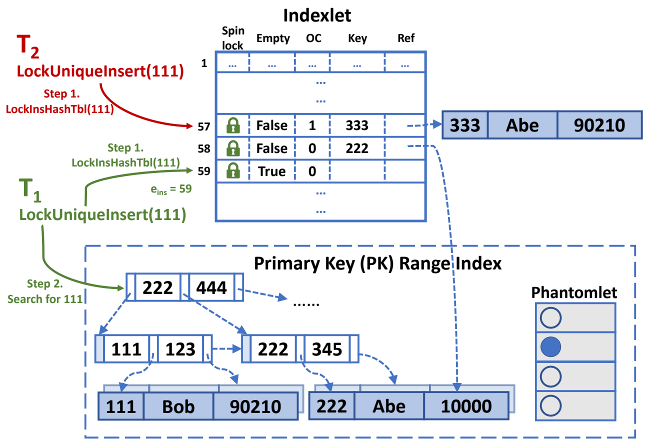


Figure 6. LockUniqueInsert Example.

Figure 6 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 §5.1) 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.

Update and deletion. To acquire a LockUpdDel, ACC performs two steps:

1. 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.
   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.

2. If the record was not found in the indexlet, ACC acquires LockInsHashTbl for the record, 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.

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.

4.2.3 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 NEW-ORDER 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 (§4.2.1), due to keys that were inserted by the same thread in a prior transaction and are waiting to be merged into the range indexes.

4.3 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 (§5.4) and is 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 $y_2 = 36$, 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. Thus, after a crash, if T_2 read x_1 and is logged, then T_1 must be logged as well. 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, in fact 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 among them.

4.4 Correctness

Using ACC, ScaleDB guarantees serializability [28], with the additional guarantee that the equivalent serial order is one where transactions are ordered by their commit timestamps. ACC derives its correctness guarantees in part from the guarantees provided by the locks and data structures it builds upon, as well as its descendence from OCC, which guarantees serializability [73]. The key difference from OCC is that ACC must deal with ScaleDB's asynchronous updates to range indexes. Our proof of correctness [64] shows that ACC's atomic commit and phantom detection protocols provide serializability in this scenario.

5 Implementation

We implement ScaleDB by modifying the Peloton [18] 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.

5.1 Indexlet and Phantomlet Hash Table

Our indexlet and phantomlet implementations build on a simple open-addressing [41] hash table which uses linear probing for resolving collisions. We considered more sophisticated open-addressing schemes like Cuckoo hashing [2, 7] 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: thus we can avoid mechanism contention 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 [29, 36]. 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 (§4.2.2).

5.2 Lock-Free Reads

To avoid reader contention on the same indexlet or phantomlet entries, ScaleDB provides *LockFreeRdHashTbl* (based on seqlocks [3]). 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 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) [24] for implementing LockFreeRdHashTbl [73]. 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 (§4.2.2).

5.3 Concurrent Range Index

Our range index implementation is a B+ tree with optimistic latch coupling (OLC) [60], used in a recent study [77] 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 nodes need modification.

5.4 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 multicore processors, such as the timestamp counter (TSC) on recent Intel x86 processors, which runs at a constant rate. Intel has indicated [52] 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 [8, 9]. Recent work [34, 35] 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 [20, 26].

On architectures where a system-wide TSC is not available, it is possible to use a dedicated timing thread [71] for handing out timestamps. 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 [71], but requires a core.

6 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. What is the query scalability of ScaleDB when compared to Peloton (§6.1)? We use YCSB to answer this question.
2. How does ScaleDB scalability compare to Cicada when guaranteeing serializability for transactions (§6.2)? Is the transaction abort rate affected? We evaluate TPC-C.

3. Is the ScaleDB asynchronous architecture a scalable design (§6.3)? We evaluate the scalability of indexlets (phantomlets) and system-wide timestamps given that these mechanisms are necessary for a scalable, asynchronous database.

Testbed. All machines in the evaluation have 2×18-core Intel Xeon Gold 6154 CPUs with 36 cores. 192GB of memory is divided across two NUMA nodes. Each machine has a Mellanox ConnectX-5 NIC, 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 [13] as needed to saturate the database server. Accordingly, all experiments report peak throughput.

6.1 Asynchronous Index Update

We evaluate ScaleDB's scalability of asynchronous updates to a single range index and compare to Peloton. For this purpose, we use the Yahoo! Cloud Serving Benchmark (YCSB) [40] read-insert workload. To generate 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 [77] 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 for reads 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 7a shows terminal scalability 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 detail this effect, we examine the sensitivity of both systems to increasing write intensity by varying the fraction of inserts in the workload, fixing the number of terminals to 160. Figure 7b 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.

6.2 Serializability

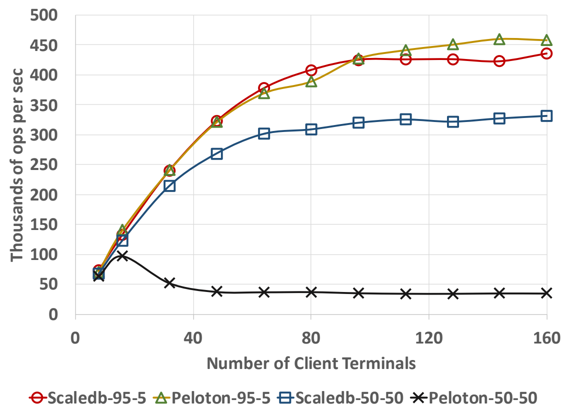
We now evaluate asynchronous scalability with serializable transactions on the TPC-C benchmark, which has multiple tables and several primary and secondary range indexes. We compare with the Cicada [62] database. Cicada's prototype does not have a network layer and it uses a TPC-C implementation linked with the database binary, calling directly into the Cicada function call API as opposed to sending SQL calls across the network. For a fairer comparison, we do the same for ScaleDB. Cicada's prototype also pre-allocates all of its memory using huge pages. Recent work from Huang et al. [51] has recommended avoiding this strategy since it "changes system dynamics significantly—for instance, pre-allocated 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 as well. Therefore, we modify Cicada to instead use jemalloc [6], which is what ScaleDB uses for memory allocation. Finally, Cicada simplifies multi-column keys by reducing them to 64-bit integers (using assumptions about the maximum range of each column). Thus, all key comparisons in Cicada are between single 64-bit integers, while ScaleDB stores and compares multi-column keys (with possibly varying column types). Hence, the baseline performance in this evaluation is biased against ScaleDB. We report self-normalized scalability, in addition to raw transactional throughput (Figure 8), for a more complete picture.

TPC-C does not run range scans on the WAREHOUSE, DISTRICT and ITEM tables. Cicada uses hash indexes for these tables and we do the same for ScaleDB. For the other tables, ScaleDB's maximum per-thread batch sizes are calculated using the method outlined in §4.1. The New-Order and Delivery transactions exert a very small W-to-RS latency on the NEW-ORDER table, requiring this table's maximum batch size to be set to 0 (*i.e.,* synchronous merging into the range index). The remaining tables have a batch size of 2,048.

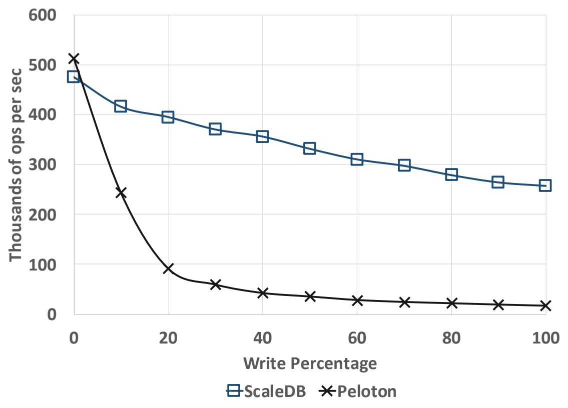
Figure 8 shows the TPC-C evaluation. The setup for these experiments is the same as that of Figure 2. ScaleDB does not have a partitioned index configuration, so all results for ScaleDB use shared indexes. On the canonical TPC-C benchmark (Figure 8a), ScaleDB scales 22.3× on 36 cores (relative to its single core throughput), which is significantly better than Cicada's scalability (with shared indexes) of 6.4×. At scale, ScaleDB's raw throughput is 1.8× higher than Cicada.

Partitioned indexes show the upper bound for Cicada's scalability. ScaleDB, with shared indexes, achieves better self-normalized scalability than Cicada with partitioned indexes (Cicada scales only 20× over 36 cores). At scale, ScaleDB's raw throughput is 60% of Cicada. Of course, Cicada's partitioned indexes do not generalize to skewed workloads.

We also evaluate a workload (NewOrd-Deliv, Figure 8b) consisting of TPC-C transactions New-Order and Delivery in equal proportions. On this more index-contended benchmark,

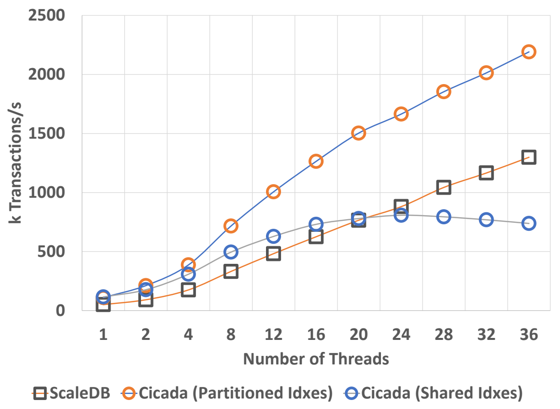


(a) Throughput.

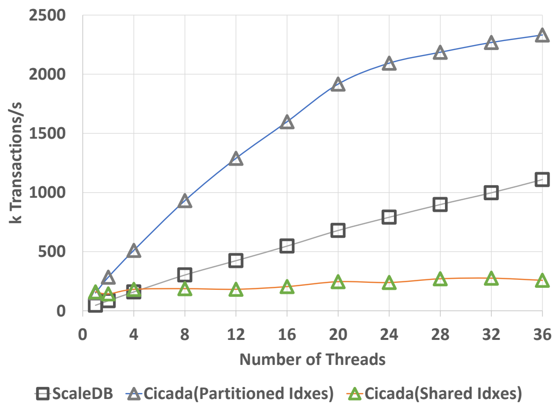


(b) Write sensitivity.

Figure 7. YCSB read-insert workload. 95-5 is 95% reads and 5% inserts. 50-50 is 50% reads and 50% inserts.

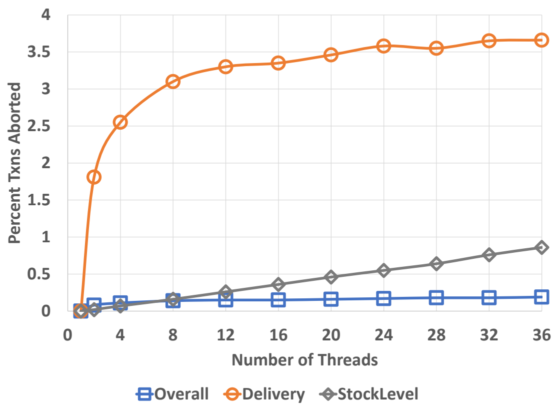


(a) TPC-C.

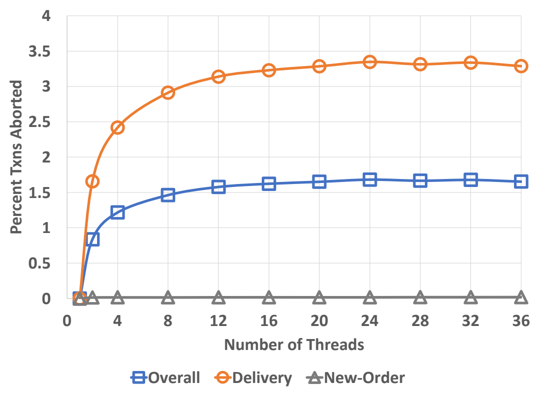


(b) NewOrd-Deliv.

Figure 8. ScaleDB vs Cicada goodput scalability on the TPC-C benchmark. Goodput counts only committed transactions.

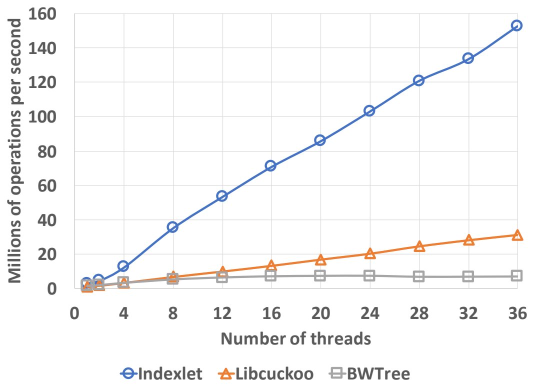


(a) TPC-C.

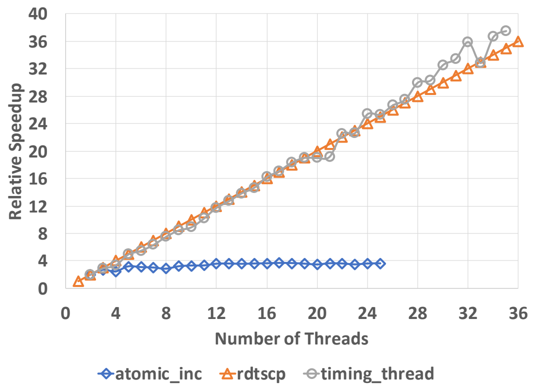


(b) NewOrd-Deliv.

Figure 9. ScaleDB abort rate.



(a) Indexlet.



(b) System-wide timestamps.

Figure 10. ScaleDB mechanism scalability.

ScaleDB maintains its scalability, while Cicada's scalability is severely impacted. ScaleDB scales 24× over 36 cores, compared to only 1.6× for Cicada with shared indexes. ScaleDB's throughput is 4.3× higher than Cicada. With partitioned indexes, Cicada scales 15.5×, still worse than ScaleDB using shared indexes. At scale, ScaleDB (with shared indexes) provides 48% of Cicada's throughput (with partitioned indexes).

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 an increasing number of threads. Figures 9a and 9b show this evaluation. On the canonical TPC-C benchmark, only the Delivery and StockLevel transactions have a non-negligible abort rate. The Delivery abort rate stabilizes at 3.5% around 20 cores (for both workloads), which implies that ScaleDB continues scaling even beyond 36 cores. The Stock-Level abort rate stays under 1%, even for 36 cores.

We also evaluated sensitivity of the abort rate to batch size, but found that our workloads were not very sensitive to even significant variations around the initial batch size—calculated according to the expected write rate for the corresponding table (§4.1). Accordingly, we omit those results for brevity.

6.3 ScaleDB Mechanisms

Indexlets. We evaluate the scalability of indexlets against libcuckoo [7], an optimized concurrent hash table, and the BwTree [61, 77], 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 10a shows, indexlets achieve nearly 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.

Memory Overhead. Indexlets have low memory overhead. Each indexlet entry only contains the primary key, a reference to the actual database row, and a small amount of metadata (e.g., a spinlock). For primary keys composed of integer columns, such as those in TPC-C tables, an indexlet entry can fit within a cache line (i.e. 64 bytes). As a result, the maximum size of an indexlet in our benchmarks was ~60MB, even for tables (e.g. the TPC-C Orderline table) which absorbed millions of record inserts per second at peak. For phantomlets, the memory overhead is even more modest, as their entry count is sized according to the expected number of leaf index nodes used for inserts per epoch. Accordingly, the maximum size of phantomlets in our benchmarks was lower than 1MB.

System-wide timestamps. We evaluate the TSC and timing thread approach (§5) and compare with an atomic increment as a global timestamp. As Figure 10b shows, both timing thread and TSC approaches scale linearly to 36 cores, while the atomic increment does not scale beyond 4 cores.

7 Conclusion

ScaleDB is an asynchronous in-memory database that provides scalability and serializability for ACID transactions. ScaleDB asynchronously updates range indexes by temporarily holding writes in indexlets that are merged periodically into range indexes. ScaleDB uses asynchronous consistency control (ACC) to provide transaction serializability. ACC extends OCC with asynchronous phantom detection via phantomlets and atomic transcation commit using locks in indexlets, rather than range indexes. For durability, ScaleDB uses system-wide time stamp counters for scalable redo logging. ScaleDB achieves 9.5× better query throughput than Peloton on the YCSB benchmark and 1.8× better transaction throughput than Cicada on the TPC-C benchmark.

Acknowledgments. We thank the anonymous reviewers and our shepherd, Murat Demirbas, for their feedback. This work was supported by NSF grant 2227066.

References

[1] Azure SQL database: Managed, intelligent SQL in the cloud. https://azure.microsoft.com/en-us/services/sql-database/.

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

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

[4] Google Cloud Spanner. https://cloud.google.com/spanner/.

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

[6] jemalloc. https://jemalloc.net/.

[7] libcuckoo. https://github.com/efficient/libcuckoo.

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

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

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

[11] MyRocks: A space- and write-optimized MySQL database. https://engineering.fb.com/core-data/myrocks-a-space-and-write-optimized-mysql-database/.

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

[13] OLTP-Bench. https://github.com/oltpbenchmark/oltpbench.

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

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

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

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

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

[19] Time-series data: Why (and how) to use a relational database instead of NoSQL. https://www.timescale.com/blog/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c/.

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

[21] TimesTen: Fastest OLTP database, ultra high availability, elastic scalability. https://www.oracle.com/database/technologies/related/timesten.html.

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

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

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

[25] Why Uber Engineering Switched from Postgres to MySQL. https://www.uber.com/blog/postgres-to-mysql-migration/.

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

[27] Firas Abuzaid, Peter Bailis, Jialin Ding, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. Macrobase: Prioritizing attention in fast data. *ACM Trans. Database Syst.*, 43(4):15:1–15:45, December 2018.

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

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

[30] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.

[31] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. Spanner: Becoming a SQL system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 331–343, 2017.

[32] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

[33] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983.

[34] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 69–86, 2017.

[35] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR-2014-019, MIT, September 2014.

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

[37] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, 2010.

[38] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 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*, VLDB '01, pages 181–190, 2001.

[39] Biswapesh Chattopadhyay, Sagar Mittal, Roee Ebenstein, Nikita Mikhaylin, Hung-ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Priyam Dutta, Selcuk Aya, Vera Lychagina, Brett Elliott, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, and David Lomax. Procella: unifying serving and analytical data at YouTube. *Proceedings of the VLDB Endowment*, 12:2022–2034, August 2019.

[40] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, 2010.

[41] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 11. The MIT Press, 3rd edition, 2009.

[42] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, 2013.

[43] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, December 2013.

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

[45] Jose M. Faleiro and Daniel J. Abadi. Latch-free synchronization in database systems: Silver bullet or fool's gold? In *8th Biennial Conference on Innovative Data Systems Research*, CIDR '17, pages 9–21, 2017.

[46] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 50–59, 2004.

[47] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccNUMA: Exploiting skew with strongly consistent caching. In *Proceedings of the 13th EuroSys*

Conference, EuroSys '18, 2018.

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

[49] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity*, SIROCCO'07, pages 124–138, 2007.

[50] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 1–7, 2014.

[51] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *Proc. VLDB Endow.*, 13(5):629–642, January 2020.

[52] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 3B, chapter 17, pages 17–41. November 2018.

[53] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module. http://arxiv.org/abs/1903.05714, 2019.

[54] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, August 2008.

[55] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1675–1687, 2016.

[56] Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 691–706, 2015.

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

[58] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, December 2011.

[59] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, December 1981.

[60] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, pages 3:1–3:8, 2016.

[61] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering*, ICDE '13, pages 302–313, 2013.

[62] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 21–35, 2017.

[63] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, 2012.

[64] Syed Akbar Mehdi. *Scalability through Asynchrony in Transactional Storage Systems*. PhD thesis, The University of Texas at Austin, 2022. Appendix 2.

[65] C. Mohan and Frank Levine. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, pages 371–380, 1992.

[66] Shuai Mu, Sebastian Angel, and Dennis Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of the 14th EuroSys Conference 2019*, EuroSys '19, 2019.

[67] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 677–689, 2015.

[68] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. An analysis of load imbalance in scale-out data serving. *SIGMETRICS Perform. Eval. Rev.*, 44(1):367–368, June 2016.

[69] Andrew Pavlo. What are we doing with our lives? Nobody cares about our concurrency control research. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 3, 2017.

[70] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, 2012.

[71] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clementine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks, 2017. https://arxiv.org/abs/1702.08719.

[72] Debendra Das Sharma. Compute Express Link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects*, HOTI '22, pages 5–12, 2022.

[73] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, 2013.

[74] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, and Xiaofeng Bao. Amazon Aurora: On avoiding distributed consensus for I/Os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 789–796, 2018.

[75] Stavros Volos, Djordje Jevdjic, Babak Falsafi, and Boris Grot. Fat caches for scale-out servers. *IEEE Micro*, 37(2):90–103, March 2017.

[76] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1643–1658, 2016.

[77] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a Bw-Tree takes more than just buzz words. In *Proceedings of the 2018 ACM International Conference on Management of Data*, SIGMOD '18, pages 473–488, 2018.

[78] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014.

[79] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642, 2016.