CobbleDB: Modelling Levelled Storage by Composition

Emilie Ma
University of British Columbia
Vancouver, Canada
contact@emilie.ma

Ayush Pandey
Télécom SudParis
Palaiseau, France
ayush.pandey@telecom-sudparis.eu

Annette Bieniusa
RPTU University of Kaiserslautern-Landau
Kaiserslautern, Germany
bieniusa@cs.uni-kl.de

Marc Shapiro
Sorbonne-Université—LIP6 & Inria
Paris, France
marc.shapiro@acm.org

Abstract

We present a composition-based approach to building correct-by-construction database backing stores. In previous work, we specified the behaviour of several store variants and proved their correctness and equivalence. Here, we derive a Java implementation: the simplicity of the specification makes manual construction straightforward. We leverage spec-guaranteed store equivalence to compose performance features, then demonstrate practical value with CobbleDB, a reimplementation of RocksDB's levelled storage.

CCS Concepts

• **Information systems** → **Storage architectures**; • **General and reference** → *Verification*; *Design*; • **Theory of computation** → *Database theory*.

ACM Reference Format:
Emilie Ma, Ayush Pandey, Annette Bieniusa, and Marc Shapiro. 2026. CobbleDB: Modelling Levelled Storage by Composition. In *Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '26), April 27–30, 2026, Edinburgh, Scotland, UK*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3806077.3806696

1 Introduction

Rigorous development techniques, based on formal specifications, increase confidence that code is bug-free, particularly in critical applications like databases. However, verifying systems from scratch is daunting: RocksDB's core backing store is ≈300 KLoC [16], and Redis's core is ≈100 KLoC [15].

This paper builds upon our prior formal specification framework for database backing stores and a generic transactional protocol [17]. We proved that this specification avoids anomalies, and that store lookup returns the formally-specified value. Our first contribution in this paper is a Java implementation, derived by hand directly from this spec (§ 3). Our specification uses a common store API that encapsulates specific store variants. This enables us to compose basic stores into high-level features, such as write-ahead logging or data compaction. Thus, we can improve store performance while retaining the same correctness guarantees (§ 4).

Our second contribution in this paper is CobbleDB, a basic reimplementation of RocksDB's levelled storage, under this formal framework (§ 5). CobbleDB supports levelled persistent storage and compaction, guarantees Transactional Causal Consistency (TCC) or Snapshot Isolation (SI) [2], and achieves reasonable (for Java) performance. CobbleDB demonstrates how naturally our formal framework describes existing system models.

Our generic, uniform spec makes it simple to compose the optimizations that are essential for a practical database backend. Composition provides an approachable onramp to implement complex features with provable rigor.

2 Formal Background

This section summarises our formal model [17]. We abstract an update as an *effect* δ, a function from a key's pre-value to its post-value. There are assignment effects, like $\delta_{\text{assign } 5}$, as well as increment effects, such as $\delta_{\text{incr } 10}$. Using increment effects to perform updates blind (i.e. regardless of the original value) avoids the expensive read-modify-write cycle required to implement increments via assignments.

We specify both the sequential and the concurrent behaviour of effects. Effects combine sequentially with the associative operator $\odot$ (pronounced apply). For instance, $\delta_{\text{assign } 2} \odot \delta_{\text{incr } 1} = \delta_{\text{assign } 3}$. Concurrent effects are *merged*: $\mathsf{merge}(\delta_{\text{incr } 1}, \delta_{\text{incr } 3}) = \delta_{\text{incr } 4}$. To ensure that concurrent updates lead to a deterministic final state, a merge operation is associative, commutative, and idempotent, as in Conflict-Free Replicated Datatypes [18].

We formalise a transactional semantics, driven by client requests, and guaranteeing TCC [2]. A transaction begins; reads from a *snapshot timestamp* into a local buffer; applies effects, which it buffers locally; and terminates, either by aborting, or by making its buffered effects visible with a *commit timestamp*. Each action in the transaction semantics calls into the corresponding method of the *store* class, i.e., doBegin, doUpdate, doAbort, doCommit. The Lookup method loads the snapshot. A snapshot includes all effects with commit timestamps strictly less than the snapshot's timestamp. We capture this *visibility* relationship as a DAG, called a *trace*. Commit timestamps are unique, and a commit timestamp may not be less than any already-started snapshot (the noInversion constraint) [11].

The *valuation* function specifies the expected value of a key at some point in the trace. Starting from an empty state, the sequential (resp. concurrent) effects visible at that point are apply'ed (resp. merge'd). Importantly, any effect before the most recent assignment can be ignored.

2.1 Specifying Backing Stores

A backing store is the database component that reliably stores and retrieves data, by implementing the store methods outlined above. Our specification formalises the behaviour of classical *map-based* and *journal-based* backing stores. Both maintain a partially ordered set of effects.

This work is licensed under a Creative Commons Attribution 4.0 International License.
PaPoC '26, Edinburgh, Scotland, UK
© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2637-8/26/04
https://doi.org/10.1145/3806077.3806696

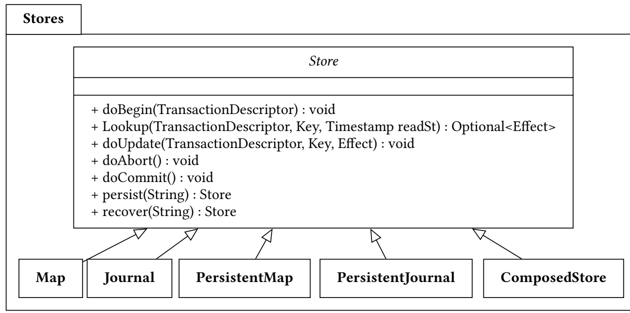

Figure 1: Basic stores class hierarchy diagram.

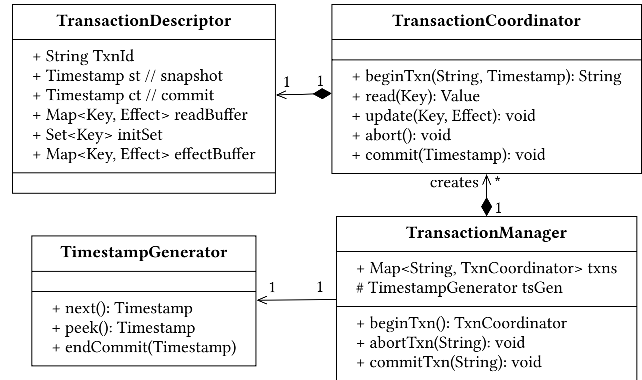

Figure 2: Transaction layer class diagram.

The *map store* variant is designed for read-heavy workloads. It maps a key to a list of effects, with metadata to distinguish concurrent effects. Uncommitted writes remain transaction-private, i.e., doUpdate is a no-op; doCommit pushes all its buffered effects in bulk. Map.Lookup searches for the most recent assignment(s) up to a specified snapshot timestamp, and applies/merges increments that follow.

The *journal store* variant is a sequential structure ordered by time; it is optimised for writes and is crash tolerant. A doX method (e.g., doUpdate) appends the corresponding event to its log. Journal.Lookup replays updates up to the specified timestamp; similarly for crash recovery.

We prove elsewhere [17] that map and journal stores' Lookup conform to the valuation. This implies that the map and journal specifications are correct; thus that they behave identically; and thus that they are composable. Formalising store composition enables representing the complex structure of modern databases, as discussed later.

3 Implementing Basic Stores

We implement the above spec in Java; the code is open-source [12]. We chose Java for its widely-used concurrency libraries and to leverage the Java-based YCSB benchmarks, discussed later. The implementation has no optimisations other than those mentioned herein.

Because the spec governs observable behaviour, implementation correctness reduces to conformance. For every proposition in the spec, the code has a matching assert. We also implemented extensive test suites, with 100% code coverage. Although the implementation is derived manually, correctness can thus be systematically validated.

Store logic. As shown in Figure 1, each concrete *basic store* (e.g. Map, Journal) extends the abstract class Store.

The in-memory Journal type is implemented as a thread-safe deque of records; each doX API method appends a corresponding record. Journal.Lookup extracts the committed effects, partially-orders them by visibility (based on begin and commit record timestamps), and applies or merges them as described earlier. The Map is implemented as a thread-safe hashmap of keys to partially-ordered deques of versions. Map.Lookup follows the journal algorithm, applied per-key.

We implement persistent variants of the in-memory stores. For example, the persistent journal appends records to a sequential file. To ensure crash-atomicity, doCommit flushes the file. On recovery, it checks integrity (using sentinel metadata in each record) and appends an abort record to any unterminated transaction, then proceeds normally. In contrast, a map is made read-only before being persisted, ensuring it can be recovered intact. In our usage (§ 5), a persistent store is rarely read and its size is bounded. Thus, its efficiency is not a main concern. We use standard Java serialisation.

Transaction protocol logic. Figure 2 depicts transaction-related types. A TransactionDescriptor tracks the state of a transaction, including its unique identifier, snapshot and commit timestamps, and read and write buffers. The TransactionCoordinator (TC) directly implements the transaction spec. The TransactionManager (TM) handles the lifecycle of allocating/deallocating a TC per client. This exemplifies the ease of development: given that the basic stores implement the same APIs and satisfy the same properties, the Java translation of the transaction protocol "falls out" of its spec.

A central TimestampGenerator ensures the timestamp constraints (see § 2). In particular, for noInversion, it maintains a *minimum allowed commit timestamp* always greater than the *maximum allowed snapshot timestamp.* Our logic in Algorithm 1 is inspired by the design considerations of Hatia [10]. next issues a unique commit timestamp, while endCommitNotify lazily advances the snapshot timestamp after earlier timestamp requests are durably committed. The TC's doCommit is called in between these two methods: splitting up the calls keeps next non-blocking and ensures crash-atomicity by finalizing commits only after a durable flush. If a flush is delayed and transactions are pending when endCommitNotify is called, the max snapshot timestamp does not update until the "gap" is closed, yielding at most older snapshots but never a partially-flushed timestamp.

Following the formal model, CobbleDB guarantees TCC. By strengthening the checks in commitTxn, it also supports SI. Supporting PSI or SSER would be straightforward [2].

Algorithm 1 `TimestampGenerator` pseudocode

```
minAllowedCT ← AtomicInt(0)                         ▷ CT: commit timestamp
maxAllowedST ← AtomicInt(0)                         ▷ ST: snapshot timestamp
running ← ∅                               ▷ CT's of pending transactions
▷ beginTxn sets snapshot to maxAllowedST
▷ commitTxn leases commit timestamp with next, then finalizes
  with endCommitNotify
procedure NEXT
    ts ← minAllowedCT.getAndIncrement()
    running.add(ts)
    return ts
synchronized procedure endCommitNotify(CT)
    running.remove(ts ∈ running | ts ≤ CT ∧ ts committed)
    if any ts removed then maxAllowedST ← min(running)
```

Algorithm 2 CobbleDB's levelled lookup algorithm

```
procedure LOOKUP(key, readSt)
    σ ← CobbleDB Store
    δ[] ← []                                          ▷ list of effects
    level ← −1                                   ▷ start at the live level
    while no assignment in δ[] and level < MAX do
        σ_level := σ[level]                  ▷ list of ministores at level
        for n := len(σ_level) − 1; n ≥ 0; n − − do
            δ_r ← σ_level[n].Lookup(key, readSt)
            if δ_r ≠ ⊥                        ▷ if an effect is found then
                δ[].push(δ_r)
                if δ_r.isAssignment() then break
        level ← level + 1
    return δ[n] ⊙ . . . ⊙ δ[0]
```

4 Composing Stores

Recall from § 2 that basic stores are provably equivalent: passing a Map or a Journal to a TransactionManager yields the same behaviour. Hence, we can compose stores to improve performance while observing identical results. For instance, we might serve reads from an in-memory map while persisting writes in a journal; this implements a *write-ahead log* that performs efficient sequential writes and avoids disk I/O on read. Composition helps specify high-level features; low-level implementation details are out of scope.

A composed store follows the same Store API as a basic store. We call its component stores "ministores." A ministore tracks a slice of the database history; we call *window* the closed-open interval of timestamps for which the ministore contains all effects. A composed store's window is the union of its ministore windows. The top-level window must cover the entire history: gaps would render Lookup unsafe. A ministore is significant only in its window, and equivalence holds where windows overlap.

Ministores follow the "write all, read one" rule. Thus, a doX method on a composed store recursively calls the same method on *all* its ministores in the window. This maintains equivalence between ministores within overlaps. Because the ministore windows collectively cover the whole top-level history, a composed store is thus always observationally equivalent under Lookup to a basic store. Conversely, ministores with the same window are interchangeable for Lookup; we prefer to read from the fastest one. By the equivalence of store Lookups, this optimization does not affect correctness.

5 Emulating RocksDB's Levelled Storage

To demonstrate composition, we implement CobbleDB, which emulates RocksDB by composing basic stores. RocksDB is a persistent KV store offering SI [16].[1] Its backing store is based on a Log-Structured Merge-Tree. Its *levels* contain updates from most recent (at the top) to oldest (at the bottom). Each level contains a sequence of files. The top-most *live level* receives events in its rightmost file; the remaining live level files and all lower levels (numbered from 0) are immutable except via compaction. RocksDB uses levelled compaction: from the live level into L0, from L0 into L1, and so on [5]. Compaction is instrumental for RocksDB's performance.

CobbleDB implements levelled storage by composing stores. This is justified by the equivalence proof and made easy by the uniform API. CobbleDB does not implement RocksDB's other optimisations (e.g., indexing, Bloom filters).

Implementation. CobbleDB relies on two additional store types: a Checkpoint and a WALMemtablePair. A Checkpoint is a special map with a single version per key, recording its value at checkpoint time. A basic store can be compacted into a Checkpoint via its checkpoint() method, which returns Lookup on each key at the high timestamp of its window.

A WALMemtablePair (WMP) is an example of composition, combining a persistent journal with an in-memory map.[2] WMPs make up the topmost, *live level* of the backing store. A doUpdate appends the update to the journal only (recall that Map.doUpdate is a no-op), whereas doCommit also pushes all the transaction's updates to the map. On the other hand, Lookup is routed to the map for fast reads.

See Algorithm 2 for CobbleDB's lookup pseudocode.[3] Lookup proceeds from most to least recent update. It reads the WMPs in the live level, accumulating effects and stopping at the first assignment. If no assignment is reached in the live level, it proceeds similarly through L0, then to L1, and so on. The store finally applies the effects in sequence.

Our implementation restricts concurrency to ensure that every transaction starts and finishes within the same ministore. Then, there may be concurrent transactions to be merge'd within the same ministore only, and no concurrency between ministores or levels. As a result, we can consolidate effects per ministore and simply apply these in sequence.

Compaction. RocksDB's levelled compaction reduces read amplification and storage requirements. Each level has a storage capacity, which triggers compaction when it is exceeded [5]. L0 contains newly flushed data from the live level, sharded by timestamp; other levels are sharded by key.

See Algorithm 3 for CobbleDB's compaction pseudocode. At the live level, compaction checkpoints the oldest stores until the

[1]The spec of § 2 supports TCC; CobbleDB strengthens this to SI for a fair comparison with RocksDB.

[2]WAL and Memtable are RocksDB's names for our Persistent Journal and In-Memory Map, respectively.

[3]The use of composition and the data model come from the formal specification. The specific algorithms are modelled after those of RocksDB.

Algorithm 3 CobbleDB's compaction algorithm

procedure COMPACT
  $\sigma \leftarrow$ CobbleDB Store
  $\sigma_{-1}[\,] \leftarrow$ list of overflowing stores from start of live level
  $\sigma_{-1}[\,]$.map($s \rightarrow s$.checkpoint())
  $\sigma_0[\,]$.push($\ldots\sigma_{-1}[\,]$), remove $\sigma_{-1}[\,]$ from live level
  $level \leftarrow 0$ ▷ continue compaction at level 0
  while $level < MAX - 1$ **do**
    $\sigma_{level}[\,] \leftarrow$ list of overflowing stores from level $level$
    for each store $s \in \sigma_{level}[\,]$ **do**
      for each store $s_{low} \in \sigma_{level+1}[\,]$ **do**
        for each key k in s **do**
          if s_{low} contains k **then**
            $s_{low}[k] \leftarrow s_{low}[k] \odot s[k]$
      create s_{new} to store any keys not yet compacted
      $\sigma_{level+1}[\,]$.push(s_{new})
      remove $\sigma_{level}[\,]$ from level $level$
    $level \leftarrow level + 1$

remaining stores fit within the live level's capacity. The checkpoints are pushed to L0 and the source stores are removed. If this causes an overflow at L0, compaction continues into lower levels until the stores at each level fit within capacity. For each checkpoint on level N, compaction finds the corresponding checkpoint on level $N + 1$ and applies updates in sequence. No concurrent transactions exist across stores or levels, so compaction does not require effect merges.

Recovery. RocksDB records its internal file structure in an authoritative *MANIFEST log* [6]. Similarly, CobbleDB's MANIFEST, a persistent journal, records ministore locations in a crash-tolerant manner. On crash recovery, the MANIFEST journal retrieves each level's file paths. Each store is recovered, and the live level is checkpointed directly to L0, matching RocksDB's behaviour. Recovery requires updating the MANIFEST with the new L0 checkpoints' paths, as well as clearing the live level's old paths. The snapshot and commit timestamp of this MANIFEST transaction are equal to the highest timestamp encountered during recovery. The Transaction-Manager then sets its timestamp generator to only pick timestamps after this timestamp.

Our recovery implementation is idempotent, thus tolerating repeated crash-recovery failures. For instance, if serialization fails or the system crashes while writing a ministore file, the partially written file is not referenced by a committed MANIFEST transaction and is ignored. Similarly, if disk writes fail while checkpointing the live level; the new L0 checkpoints will be re-created on future recovery attempts. Our tests for persistence and recovery include serialization and disk I/O failure simulations.

6 Results

Our uniform, composable specification allows us to naturally express RocksDB's levelled storage structure. CobbleDB comprises 3 204 LoC in Java[4]. We apply the most straightforward data structures without concern for performance or optimization; for instance, a timestamp is represented as a map. Our tests have 100% code coverage, exercising both sequential behaviour and concurrent execution with up to ten workers. This includes scenarios with concurrent reads, writes, and commits and stress tests for composition consistency. While computing concurrent behaviour coverage is infeasible, our test suite covers the actions most critical for correctness.

The following sections present performance evaluations:

- **RQ1**: How does CobbleDB compare to basic stores?
- **RQ2**: How does CobbleDB compare to RocksDB?
- **RQ3**: How do CobbleDB's configurations compare?

RQ1: CobbleDB vs. basic backing stores. We benchmark stores by extending the YCSB framework [8] with custom transactional workloads. We run each workload with 1–20 YCSB threads, each communicating with its own database client over HTTP (`TCP_NODELAY` enabled). A transaction contains five lookup or update operations. The key distribution is zipfian. Benchmarks are run on machines with two Intel Xeon E5-2620 v4 CPUs and 64 GB RAM on a 10 Gbps network. YCSB and the store run on separate machines to avoid noise; given network latency and older hardware, our results will differ from published benchmarks using a single machine. Each experiment starts with an empty database (no warmup) and runs for 60 s. We define three workloads:

- txn: 50/50 split between lookups and assignments.
- old_reads: same, with a 50% chance that the snapshot timestamp is strictly before the last commit.
- txn_increments: 50/30/20 split between lookups, assignments, and increments respectively.

Figure 3 shows latency and throughput for CobbleDB and the basic stores, under the txn workload.[5] Each point represents a separate run; the number of YCSB client threads varies from 1 to 20. The journal, map, and CobbleDB have similar performance trends, with a latency-throughput knee at 2–4 threads. The map has lower latency than the journal: its key-sharded structure leads to smaller effect lists to process. Otherwise, the two have comparable max throughputs of 5 088 ops/s (journal) and 5 098 ops/s (map). CobbleDB's lookup latency is lower than the journal's: checkpoint compaction decreases the reads required per lookup. Compaction overhead also leads to lower throughput.

The persistent journal's performance is orders of magnitude worse. Lookup latency increases with additional threads due to extra log records, and flush-on-commit caps throughput. The persistent journal also has worse update latency than CobbleDB. This is surprising, since CobbleDB first writes to the persistent journal component of WMP. However, when increasing writes to 100% (experiment not plotted), update latency is nearly identical. We surmise that the JVM serialises file access. While CobbleDB serves reads from WMP map separately, in the persistent journal writes

[4] In comparison, RocksDB's core is ≈300 000 LoC of C++, even excluding, to the best of our ability, features that CobbleDB does not implement (e.g., compression). While this is not an apples-to-apples comparison, this still demonstrates our spec-driven approach's brevity. We exclude tests and libraries from LoC in both cases.
[5] We omit the persistent map, as it is not used in the CobbleDB composition. We also omit ranges without data points via axis breaks for visual clarity.

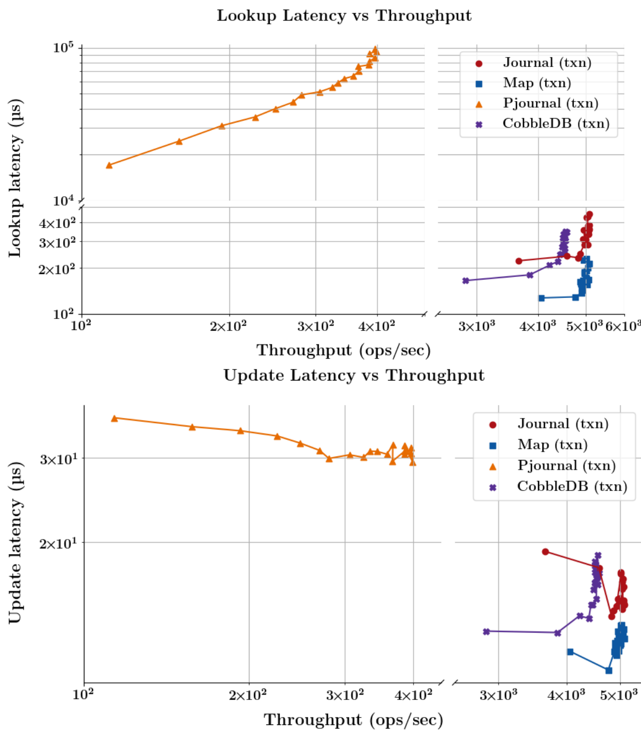

Figure 3: Latency vs. throughput for CobbleDB and basic stores.

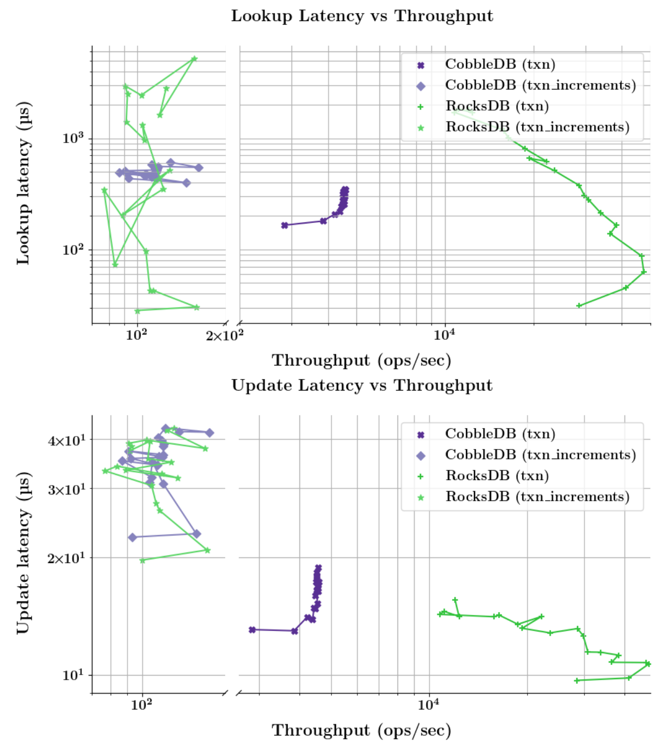

Figure 4: Latency vs. throughput for CobbleDB and RocksDB.

must wait for reads. CobbleDB's max throughput is 4 581 ops/s whereas the persistent journal's is 398 ops/s, providing 11.5× more throughput with the same persistence guarantees.

RQ2: CobbleDB vs. RocksDB. We ran the same YCSB workloads to compare CobbleDB against RocksDB in its default configuration. We note that the results are not directly comparable. For one, RocksDB is implemented in optimized C++ instead of Java. The latest Computer Language Benchmark Game (CLBG) finds the most optimized Java programs to be ≈4.5× slower than C++ for high I/O workloads (up to 12x for lower I/O workloads) [3]. As well, CobbleDB lacks many of RocksDB's performance features and was deliberately designed to reflect our spec without optimizations.

Figure 4 compares CobbleDB and RocksDB on the txn and txn_increments workloads.[6] Again, each point represents a separate run, varying the number of threads from 1 to 20.

On the txn workload, CobbleDB's throughput is one order of magnitude less than RocksDB's (RocksDB max throughput 47 498 ops/s vs. CobbleDB 4 581 ops/s). However, RocksDB's throughput decreases as thread count increases, likely due to its write backpressure [7]. RocksDB's lookup latencies grow faster than CobbleDB's, ballooning 59× from 1 to 20 threads, compared to CobbleDB's 2× increase; RocksDB's update latency grew 1.6× compared to CobbleDB's 1.3×.

On the txn_increments workload, throughput is much lower. As shown by the spikes, latency is highly variable: merging increments accumulates more state. The zipfian distribution used to select keys also introduces significant spread in how far back the search must progress. Yet, CobbleDB handles increments more efficiently than RocksDB: CobbleDB's lookup latency averages 499 μs, compared to RocksDB's 1 174 μs, and update latency matches RocksDB's. CobbleDB's max throughput on this workload also slightly exceeds RocksDB's: 161 ops/s v.s. RocksDB's 156 ops/s.

In summary, RocksDB betters CobbleDB on assignment-only workloads, but they are comparable when increments are included. On assignment workloads, the performance ratio is of the order observed in the CLBG, despite CobbleDB's spec-driven, unoptimised implementation.

RQ3: CobbleDB configurations. Figure 5 compares default CobbleDB (ensuring SI) against the txn and old_reads workloads.[7] Thread count again varies from 1 to 20 for each workload run. We also run CobbleDB under TCC against the txn workload, to exercise the concurrent updates' merge.

CobbleDB achieves a max throughput of 4 592 ops/s on the old_reads workload, relative to 4 581 ops/s on the txn workload. Despite the old_reads workload requiring deeper reads, the performance trends are similar, highlighting how composition via the checkpoint stores aids performance.

Under TCC, CobbleDB exhibits similar latency-throughput curves as under SI, with performance improvements due to an increase in successful concurrent transactions. CobbleDB's throughput peaked

[6] old_reads behaves similarly to txn and is discussed later.

[7] We omit the txn_increments workload for visual clarity given significantly lower throughput.

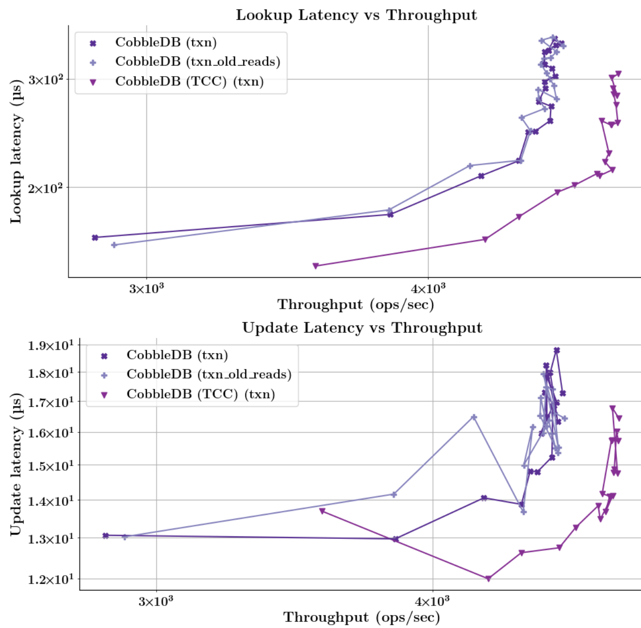

Figure 5: Latency vs. throughput for different CobbleDB configurations and workloads.

at 4 859 ops/s. CobbleDB's update latency with TCC averages 14.4 μs compared to 20.8 μs with SI, as TCC does not check nor abort conflicting updates. Lookup latency with TCC also decreased (236 μs vs. 328 μs on SI): increased updates make it more likely that lookup terminates earlier in the live, or more recent, levels.

7 Related Work

Prior research has also verified components of new database backing stores. For example, Malecha et al. develop a Rocq-verified relational database [13], and Patel et al. present a framework to verify the linearizability of concurrent key-value stores [14]. However, these works do not address crash recovery or transactional guarantees, making them inadequate for modern production use. The VerIso project addresses the transactional specification gap, proposing a Isabelle/HOL framework for concurrency control protocols [9]. It also leaves durability to future work; in contrast, CobbleDB and our spec deliberately handle these concerns.

8 Conclusion

In this paper, we show that composition is a promising way to improve the performance of formally-specified stores. Because our spec guarantees the equivalence of stores in their overlapping window, composition unlocks improvements like serving reads from faster stores while retaining the same correctness assurances. As an example, we describe CobbleDB, an emulation of RocksDB's levelled storage structure. Our test suite, based on the spec propositions and with full code coverage, lends confidence in this implementation. We found that CobbleDB achieved 9.6% of RocksDB's performance, with the discrepancy likely attributable to implementation language differences. Our implementation highlights how cleanly composition maps onto production database components, like RocksDB's levelled compaction.

Our approach is not intended to replace database development techniques wholesale, but to integrate with existing ones. Developers can continue to hand-write systems and apply conventional testing techniques, but ground their high-level architecture in a formal spec and incrementally refactor. For instance, they could start by refactoring to our WAL and recovery system to gain consistency guarantees, while keeping code paths for other features separate.

Future work includes emulating additional databases to further validate our approach, particularly distributed databases like AntidoteDB [1, 4]. As well, we plan to continue performance tuning for CobbleDB.

By starting from a generic specification and composing basic backing stores, we can cleanly derive advanced performance features. Our approach supports the practical application of formal methods, using composition to make rigorous guarantees approachable in complex storage architectures.

Acknowledgments

Emilie Ma conducted this work as a visiting intern at Sorbonne Université, based on the previous contributions of Saalik Hatia's PhD and of Ayush Pandey. We thank Jaurel Fosset for his contributions to optimizations and benchmarking. We also thank Carla Ferreira for her feedback. This work was carried out thanks to a generous Amazon Research Award, and was supported in part by ANR under the Centeanes grant ANR-24-CE25-5598 (https://anr.fr/Project-ANR-24-CE25-5598). Our experiments were carried out on the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several universities as well as other organizations (see grid5000.fr).

References

[1] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong semantics meets high availability and low latency. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*. IEEE Comp. Society, Nara, Japan, 405–414. doi:10.1109/ICDCS.2016.98

[2] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *Int. Conf. on Concurrency Theory (CONCUR) (Leibniz Int. Proc. in Informatics (LIPIcs), Vol. 42)*, Luca Aceto and David de Frutos Escrig (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Madrid, Spain, 58–71. doi:10.4230/LIPIcs.CONCUR.2015.58

[3] CLBG Contributors. [n. d.]. Measured: Which programming language is fastest? Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html Consulted 17 January 2026.

[4] AntidoteDB Contributors. 2026. https://github.com/AntidoteDB/antidote

[5] RocksDB Contributors. 2026. Compaction. https://github.com/facebook/rocksdb/wiki/Leveled-Compaction

[6] RocksDB Contributors. 2026. MANIFEST. https://github.com/facebook/rocksdb/wiki/MANIFEST

[7] RocksDB Contributors. 2026. Write Stalls. https://github.com/facebook/rocksdb/wiki/Write-Stalls

[8] Brian Cooper and YCSB Contributors. 2026. brianfrankcooper/YCSB. https://github.com/brianfrankcooper/YCSB

[9] Shabnam Ghasemirad, Si Liu, Christoph Sprenger, Luca Multazzu, and David A. Basin. 2024. VerIso: Verifiable Isolation Guarantees for Database Transactions. *Proc. VLDB Endownment* 18, 5 (Oct. 2024), 1362–1375. https://www.vldb.org/pvldb/vol18/p1362-ghasemirad.pdf

[10] Saalik Hatia. 2023. *Leveraging formal specification to implement a database backend.* Ph. D. Dissertation. Sorbonne Université. https://theses.hal.science/tel-04291337

[11] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. 2023. NCC: Natural Concurrency Control for Strictly Serializable Datastores by Avoiding the Timestamp-Inversion Pitfall. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, Roxana Geambasu and Ed Nightingale (Eds.). Usenix, Boston, MA, USA, 305–323. https://www.usenix.org/conference/osdi23/presentation/lu

[12] Emilie Ma, Ayush Pandey, Jaurel Fosset, and Saalik Hatia. [n. d.]. CobbleDB source code. https://gitlab.inria.fr/shapiro/CobbleDB. Commit 77ab1b9c..

[13] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2010. Toward a Verified Relational Database Management System. In *Symp. on Principles of Prog. Lang. (POPL) (POPL '10)*. Assoc. for Computing Machinery, Madrid, Spain, 237–248. doi:10.1145/1706299.1706329

[14] Nisarg Patel, Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2021. Verifying concurrent multicopy search structures. In *Conf. on Object-Oriented Prog. Sys., Lang. and Applications (OOPSLA) (Proc. ACM Program. Lang., Vol. 5)*. Assoc. for Computing Machinery Special Interest Group on Pg. Lang. (SIGPLAN), Assoc. for Computing Machinery, Chicago, Ill., USA, 32 pages. doi:10.1145/3485490

[15] Redis. 2026. Redis. https://github.com/redis/redis

[16] RocksDB Contributors. 2026. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. https://github.com/facebook/rocksdb

[17] Edgard Schiebelbein, Saalik Hatia, Annette Bieniusa, Gustavo Petri, Carla Ferreira, and Marc Shapiro. 2024. Models for Storage in Database Backends. In *W. on Principles and Practice of Consistency for Distr. Data (PaPoC)*, Assoc. for Computing Machinery (Ed.). Euro. Conf. on Comp. Sys. (EuroSys), Assoc. for Computing Machinery, Athens, Greece, 58–66. doi:10.1145/3642976.3653036

[18] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS) (Lecture Notes in Comp. Sc. (LNCS), Vol. 6976)*, Xavier Défago, Franck Petit, and V. Villain (Eds.). Springer-Verlag, Grenoble, France, 386–400. doi:10.1007/978-3-642-24550-3_29