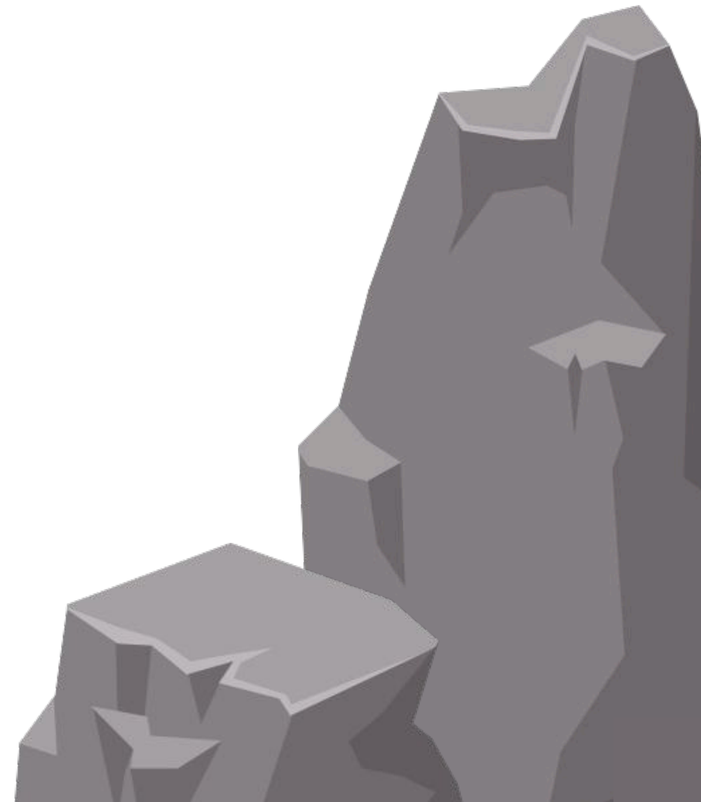
CobbleDB: Modelling Levelled Storage by *Composition*

Emilie Ma, Ayush Pandey, Annette Bieniusa, & Marc Shapiro ◦ PaPoC 2026



Motivation

Writing databases is hard.

AntidoteDB (“Planet-scale, highly available, replicated DB”) = 60K LoC, 6 PhDs, still has data loss bugs

In principle, formal methods eliminate bugs.

Specs help us articulate behaviour and uncover correctness issues early.

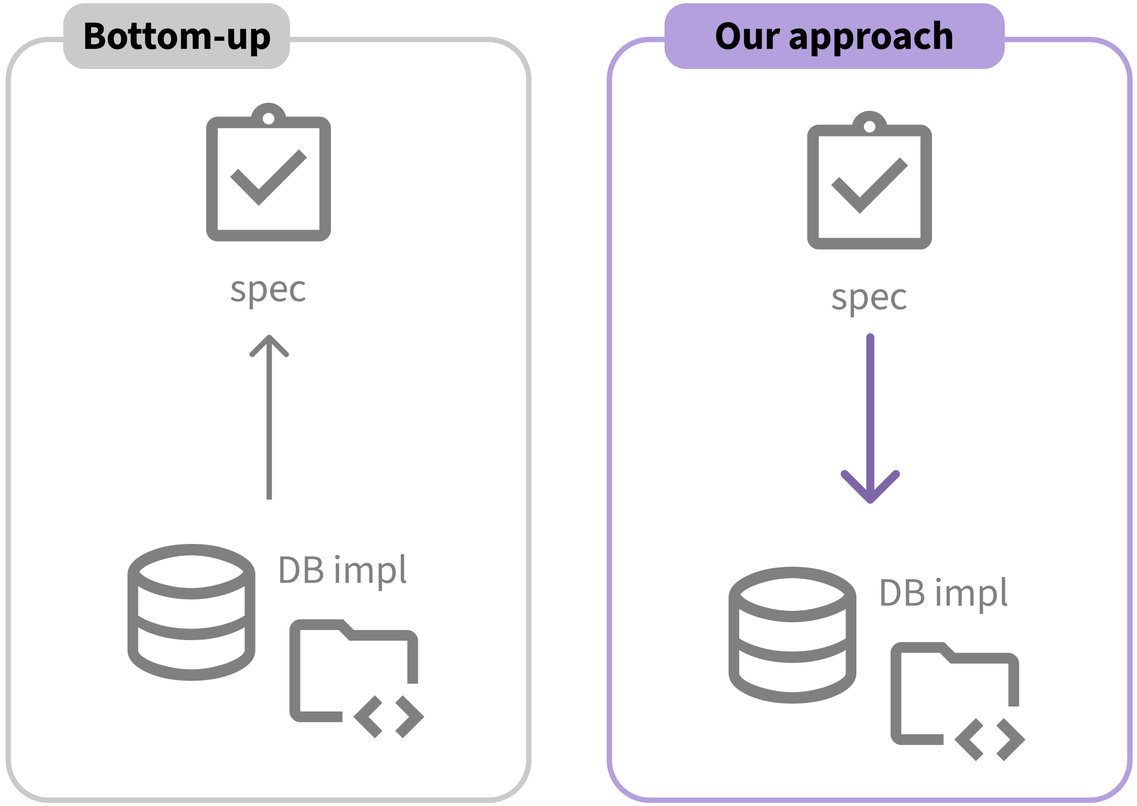
In practice, they seem infeasible.

Redis ≈ 100K LoC in C
RocksDB ≈ 300 KLoC in C++
Looks impossible…



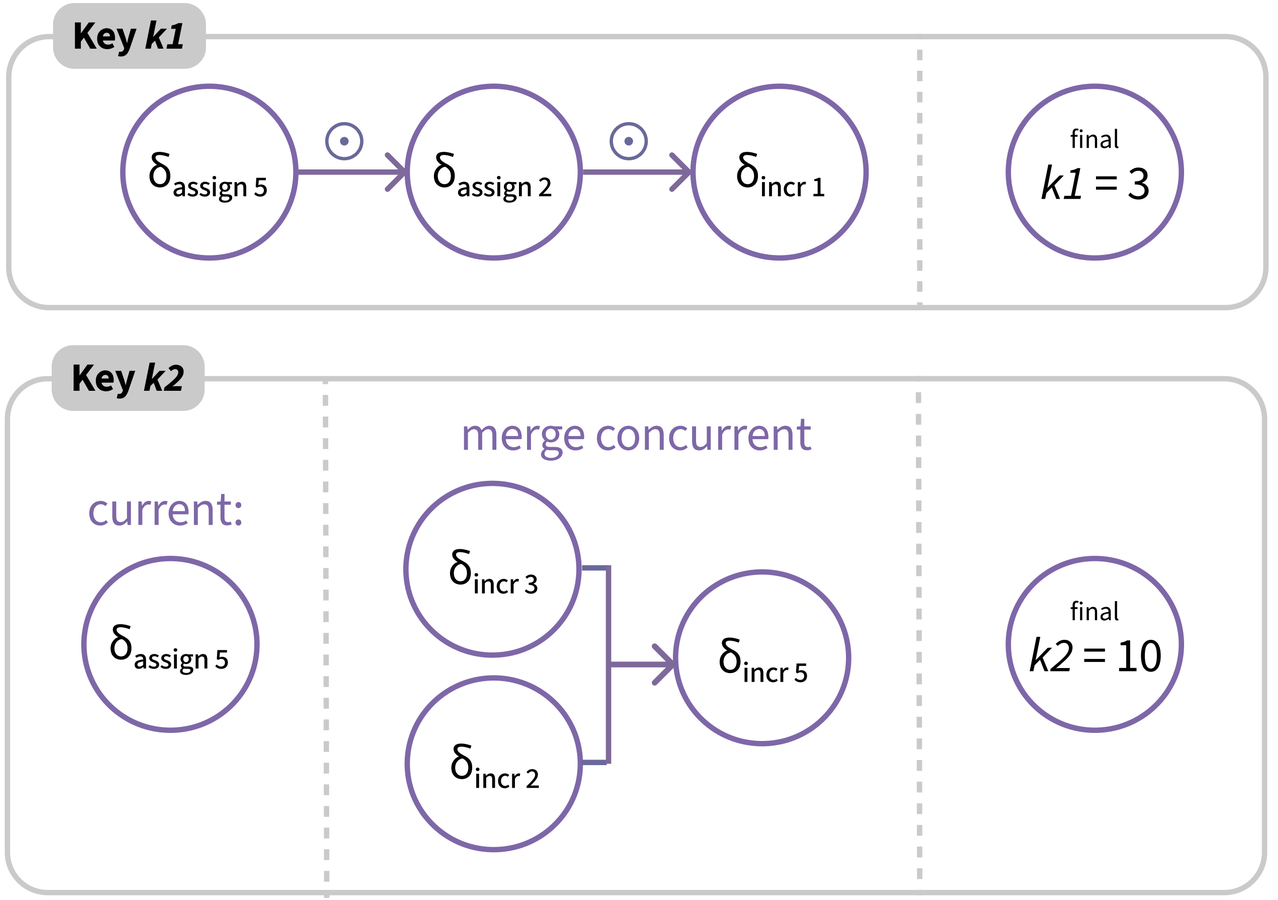
Let's work the other way!

- Instead of specifying a DB bottom-up, verify it top-down
- Schiebelbein et al. (2024) constructed a formal spec of a basic transactional system
- This paper: implement it to build a correct-by-construction DB!



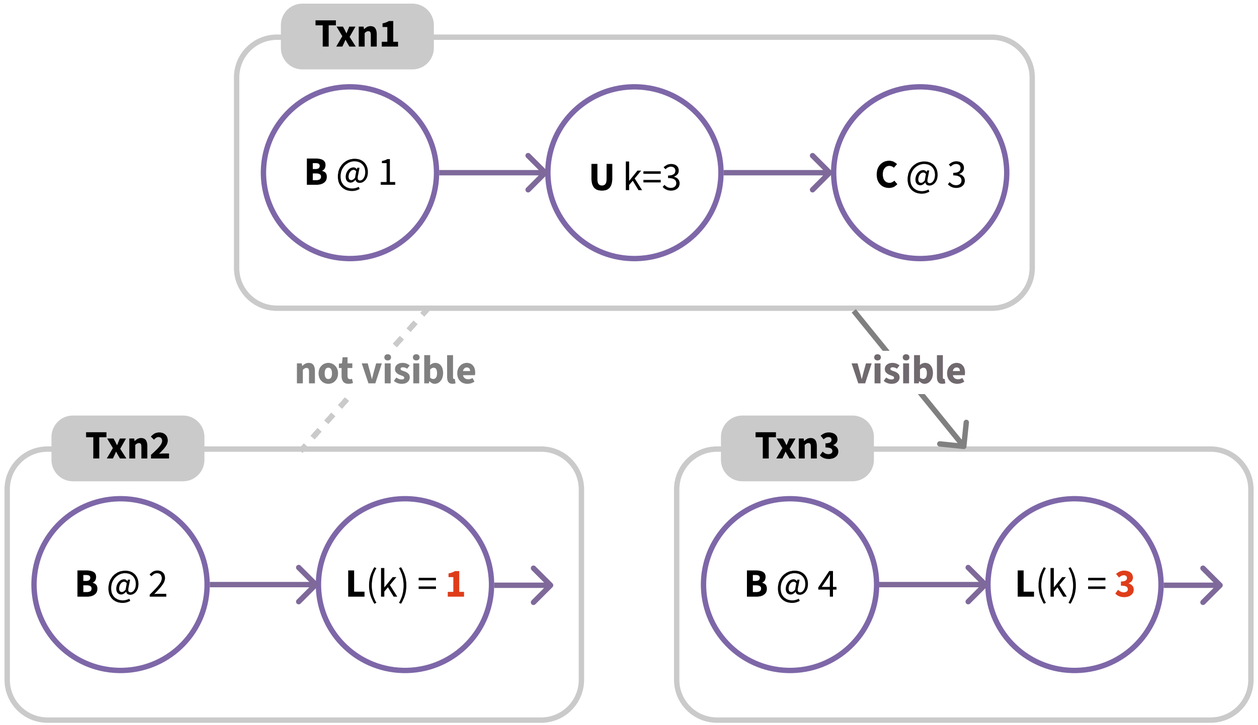
Formalizing Updates

- Effects (δ): change a key's value
- Assignment effects: $\delta_{\text{assign 5}}$
- Increment effects: $\delta_{\text{incr 2}}$
- Apply (⊙, in sequence) or merge (concurrent operations)
- Like CRDTs (Shapiro et al., 2011)!



Transaction Semantics

- Begin a txn. at some *snapshot timestamp*
- Lookup or update some data
- Commit at a *commit timestamp* or abort
- Sees updates from other txns. after other txns. committed
- Guarantees Transactional Causal Consistency (Cerone et al., 2015)



Backing Stores

- Reliably stores and retrieves data
- Common Store interface:
  - doBegin()
  - Lookup()
  - doUpdate()
  - doAbort()
  - doCommit()
  - persist()
  - recover()
- Implemented map and journal variants, **proven equivalent** under txn. semantics

Map

k1	$\delta_{assign\ 2}, \delta_{incr\ 5}$
k2	$\delta_{assign\ 1}, \delta_{assign\ 2}$
k3	$\delta_{assign\ 2}, \delta_{incr\ -1}$

Uncommitted:

$k3 = \delta_{assign\ 5}$

$k4 = \delta_{assign\ 3}$

Journal

Begin Txn1 @ 00:01

Update k1, $\delta_{assign\ 2}$

Update k2, $\delta_{assign\ 1}$

Commit Txn1 @ 00:04

Begin Txn2 @ 00:02

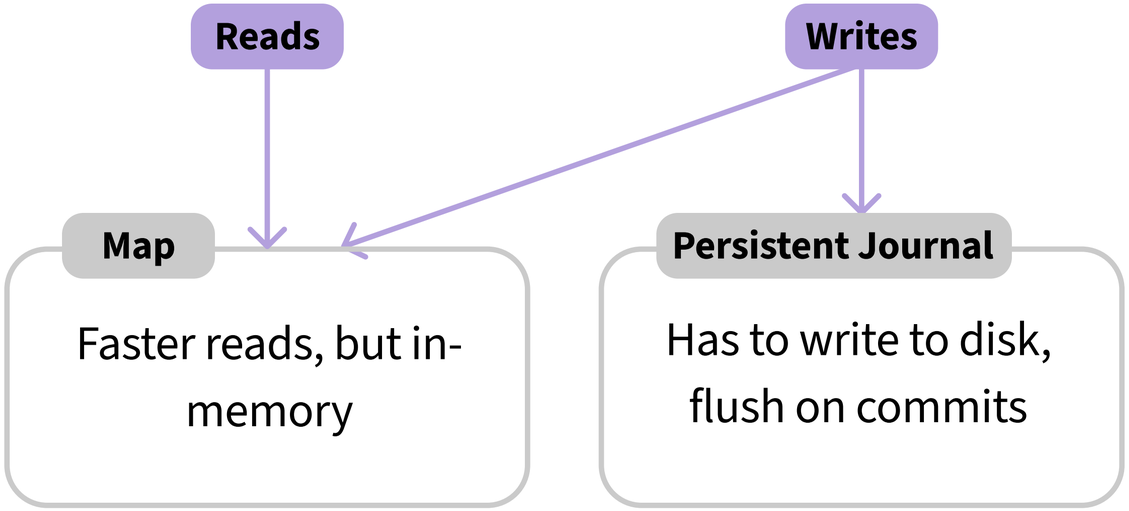
Update k1, $\delta_{incr\ 5}$

Commit Txn2 @ 00:05



✶ Store Composition

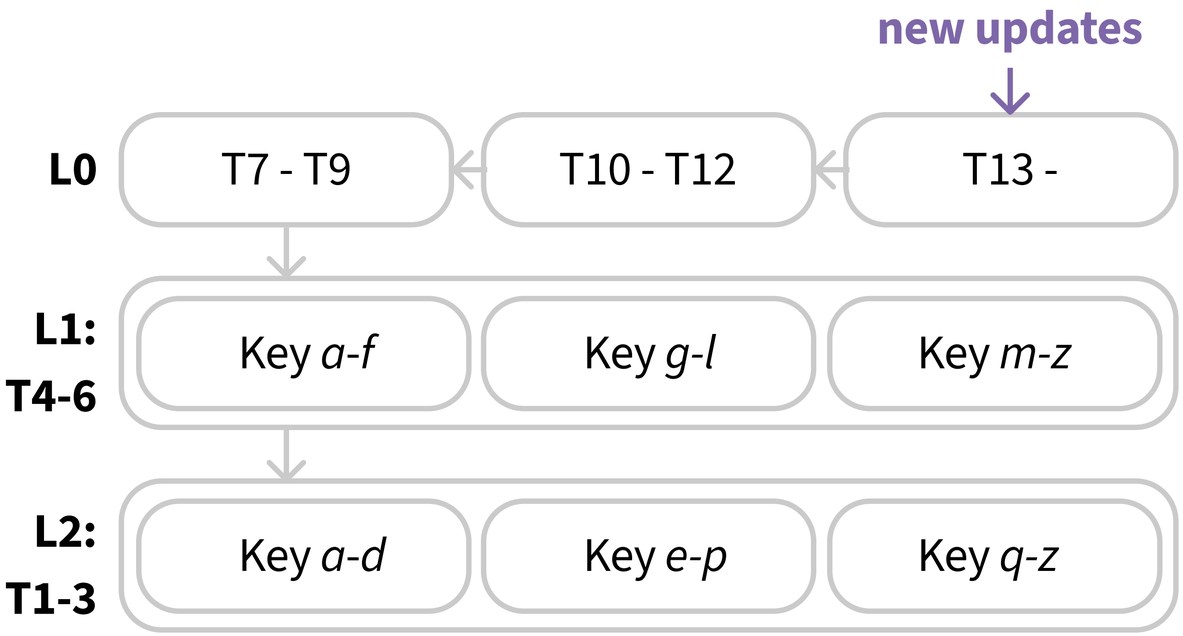
- Since we proved stores are equivalent, compose them for performance!
- Multiple ministores kept in sync, still guaranteed to be TCC
- Write all stores (for correctness), read one (for performance)
- Per-ministore key and time domains
- **Takeaway: composition boosts performance, keeps same guarantees**



Kept in sync, proved equivalent

Emulating RocksDB

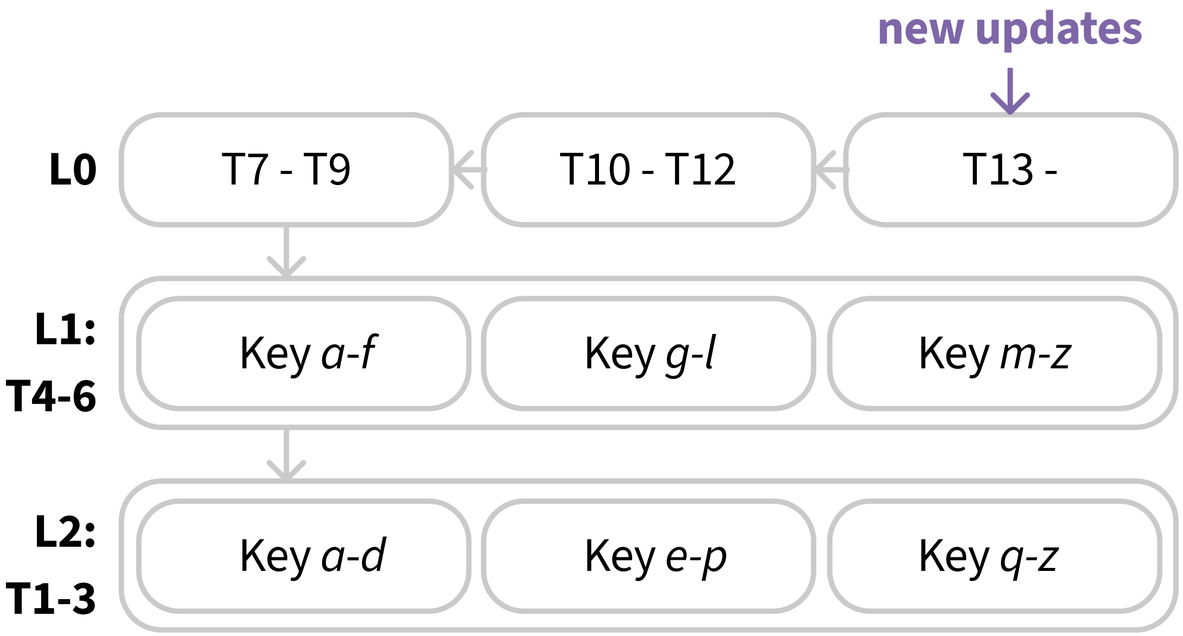
- Production persistent KV store
- Based on Log-Structured Merge Tree: L0 sharded by time, L1+ by key
- New updates added to end of L0
- Lookup goes through L1+ until assn. found, applying/merging effects
- Compaction: when L0 overflows, a new store is added and pushes older stores into lower levels



simplified view

RocksDB as a Composition

- How can we express these concepts as a composition?
- L0: composition of in-memory map and persistent journal
- L1+: composition of maps with different key domains
- Compaction: moving ministores around according to formal spec



simplified view

✶ CobbleDB

- Implemented spec in unoptimized Java, using built-in libraries
- Manual transcription of spec
- Only 3K LoC
- Supports compaction and recovery, using composition + formal spec to ensure correctness
- 100% test coverage, exercising both single- and multi-threaded actions

Spec

$$\frac{\forall x \in \mathcal{X}_a \cup \mathcal{X}_c \cup \mathcal{X}_r, x.\tau \neq \tau \qquad \sigma' = \text{doBegin}(\sigma, \tau, \text{st}) \qquad \mathcal{X}_r' = \mathcal{X}_r \cup \{(\tau, \text{st}, \emptyset, \emptyset, \emptyset, _)\}}{(\sigma, \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r) \xrightarrow[\tau]{\text{beginTxn(st)}} (\sigma', \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r')} \text{[BeginTxn]}$$

X_r' = append new running txn to X_r

CobbleDB

```
public void beginTxn(String txnId, Timestamp st) {
    TransactionDescriptor td = new TD(txnId);
    td.st = st;
    store.doBegin(td);
}
```



✶ CobbleDB

Spec

$$\frac{\mathtt{st} \le \mathtt{ct} \quad \mathit{NoInversion}(\mathtt{ct}) \quad \sigma' = \mathrm{doCommit}(\sigma, \tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct}) \quad \mathcal{X}_r = \mathcal{X}_r' \uplus \{(\tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, _)\} \quad \forall x \in \mathcal{X}_c, x.\mathtt{ct} \neq \mathtt{ct} \quad \mathcal{X}_c' = \mathcal{X}_c \uplus \{(\tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct})\}}{}\ [\textsc{Commit}]$$

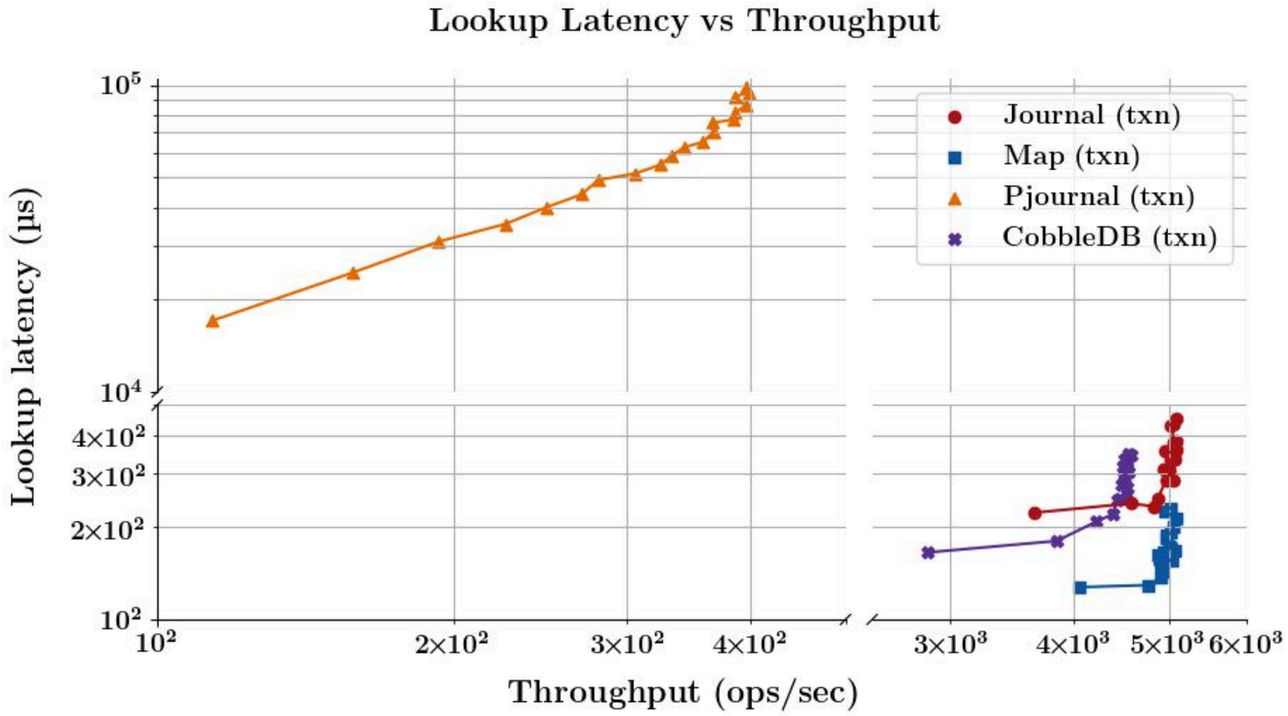
CobbleDB

```
synchronized {
    thisTxn = TC.getCurrentTxn();
    ct = TimestampGenerator.next(thisTxn);
    assert (ct > last_ct);
    assert (thisTxn.st ≤ ct);
    assert (noInversion(ct));
    store.doCommit(thisTxn.txnDescriptor);
    TC.endCommitNotify(ct);
    thisTxn.setCommitTS (ct);
}
```



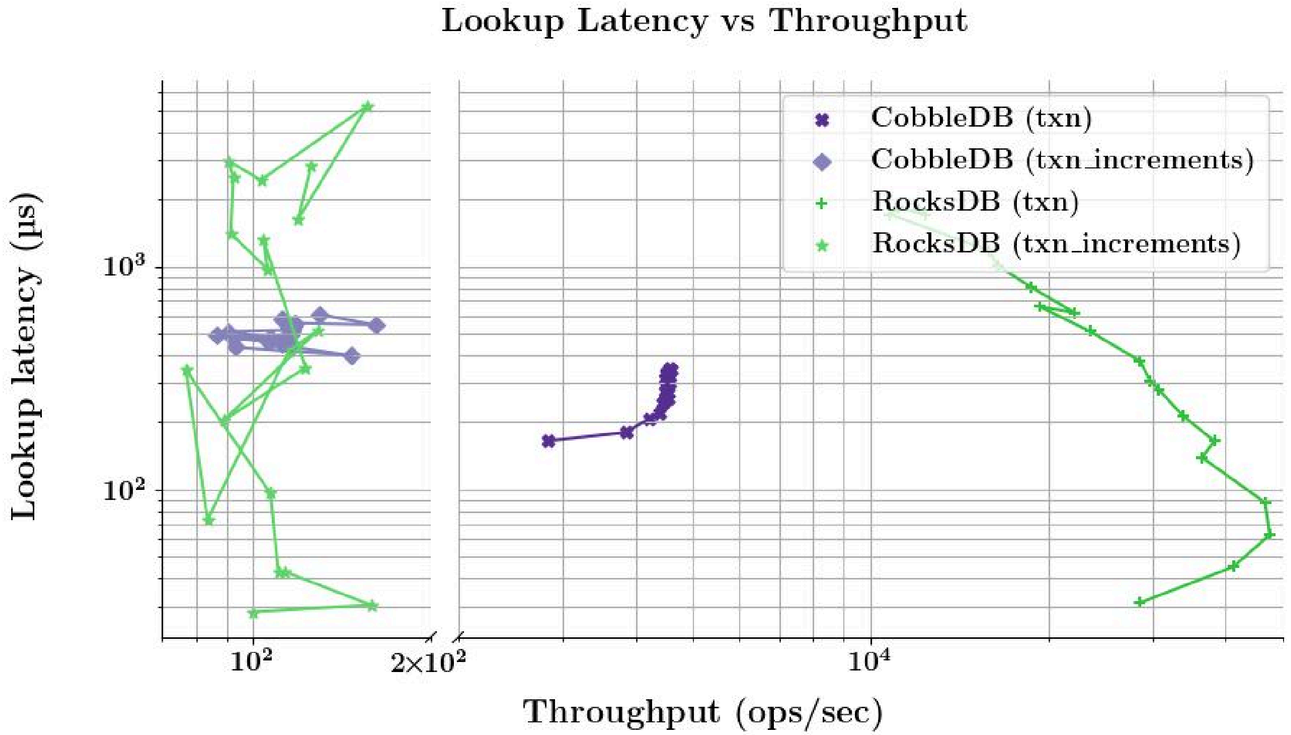
Composition & Performance

- Used YCSB benchmarking framework (1-20 threads, 50/50 lookups and assignments)
- CobbleDB similar perf. trends as in-memory stores
- PJournal orders of magnitude worse
- **CobbleDB/composition provides the same persistence guarantees with better scaling**



What about RocksDB?

- CLBG: Java 4.5x-12x slower than C++
- CobbleDB throughput is an order of magnitude lower (~CLBG ratio)
- CobbleDB latencies grow more slowly with addl. threads
- CobbleDB lower average lookup latency with increments
- **CobbleDB handles increased concurrency more efficiently**



Contributions

- **CobbleDB**: correct-by-construction emulation of RocksDB's levelled storage, directly translated from spec in 3K LoC
- Performance comparable to RocksDB when considering language differences
- **Takeaway: composition supports incremental formal methods in practice**
- Future work: perf tuning, emulating more DBs (AntidoteDB)

https://gitlab.inria.fr/shapiro/cobbledb