Models for Storage in Database Backends

A Rigorous Approach for Formally-Correct Designs

Edgard Schiebelbein
University of Kaiserslautern-Landau
Kaiserslautern, Germany
e_schiebel19@cs.uni-kl.de

Saalik Hatia
Sorbonne-Université (LIP6)
Paris, France
saalik.hatia@lip6.fr

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

Gustavo Petri*
Amazon Web Services
Cambridge, United Kingdom
gfpetri@amazon.co.uk

Carla Ferreira
Universidade NOVA de Lisboa
Lisbon, Portugal
carla.ferreira@fct.unl.pt

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

Abstract

This paper describes ongoing work on developing a formal specification of a database backend. We present the formalisation of the expected behaviour of a basic transactional system that calls into a simple store API, and instantiate in two semantic models. The first is a map-based, classical versioned key-value store; the second, journal-based, appends individual transaction effects to a journal. We formalise a significant part of the specification in the Coq proof assistant. This work will be the basis for formalising a full-fledged backend store with features such as caching or write-ahead logging as variations on maps and journals.

CCS Concepts: • **Theory of computation** → *Operational semantics*; • **Information systems** → **Information storage systems**.

Keywords: formal methods, verification, key-value store

ACM Reference Format:

Edgard Schiebelbein, Saalik Hatia, Annette Bieniusa, Gustavo Petri, Carla Ferreira, and Marc Shapiro. 2024. Models for Storage in Database Backends: A Rigorous Approach for Formally-Correct Designs. In *The 11th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3642976.3653036

*This work was conducted while Gustavo Petri was at ARM Ltd.

PaPoC '24, April 22, 2024, Athens, Greece

ACM ISBN 979-8-4007-0544-1/24/04
https://doi.org/10.1145/3642976.3653036

1 Introduction

A database system manages a collection of digital data. An essential component is the *backend*, which is in charge of recording the data into some memory or *store*. Although conceptually simple at a high level, actual backends are complex, due to the demands for fast response, high volume, limited footprint, concurrency, distribution, and reliability. For instance, the open-source RocksDB comprises 350+ kLOC and Redis is approximately 200 kLOC [5, 11]. Any such complex software has bugs; and database backend bugs are critical, possibly violating data integrity or security [7, 8].

Formal methods have the potential to avoid such bugs, but, given the complexity of a modern backend, fully specifying all the moving pieces is a daunting task.

This paper reports on an incremental approach to the rigorous and modular development of such a backend towards an implementation. To this end, we formalise the semantics of atomic transactions above a versioned key-value store; this high-level specification helps to reason about correctness, both informally and formally with the Coq proof tool. Although this paper focuses on a highly-available transaction model (convergent causal consistency or TCC+, a variant of PSI [13]), our results generalise to stronger models such as SI or strong serialisability. The transaction model appeals to a store's specialised book-keeping operations (called doBegin doUpdate, doCommit, and lookup), implicity assuming infinite memory and no failures.

Next, we instantiate these semantics with two models of the store. The first variant is a classical map-based, versioned key-value store. As a transaction executes, it *eagerly* computes new versions, which it copies into a map upon commit, labelled with the transaction's commit timestamp; reading

$$
\begin{array}{rcl}
\text{empty} & : & \Sigma \\
\text{doBegin} & : & \Sigma \times \mathit{TxnID} \times \mathit{Timestamp} \to \Sigma \\
\text{lookup} & : & \Sigma \times \mathit{Key} \times \mathit{Timestamp} \to \mathit{Effect}_{\perp} \\
\text{doUpdate} & : & \Sigma \times \mathit{TxnID} \times \mathit{Key} \times \mathit{Effect} \to \Sigma \\
\text{doCommit} & : & \Sigma \times \mathit{TxnID} \times \mathit{Timestamp} \times \mathcal{P}(\mathit{Key}) \\
 & & \times \mathcal{P}(\mathit{Key}) \times \mathit{EffectBuf} \times \mathit{Timestamp} \to \Sigma
\end{array}
$$

Figure 1. Store interface.

a key searches the map for the most recent corresponding version. We plan to mechanise the proof that the map-based model satisfies the transactional specification, i.e., that in any reachable state, a call to read returns the value expected by the semantics for any key and timestamp pair.

Our second variant uses a journal (or log). A transaction appends individual effects to the journal, tagged with the transaction identifier; committing appends a commit record, sealing it with its commit timestamp. Reading from the journal applies all the relevant effects previously recorded in the journal. Again, we plan to prove mechanically that the journal-based store satisfies the abstract specification.

In this paper, we summarize our work-in-progress on the formal models for the journal- and map-based store. We define a common interface and show how these stores can be employed in a transactional storage system. Further, we sketch their implementation and reasoning about their correctness in Coq.

The models presented here lack fault-tolerance and essential features such as sharding, caching, write-ahead logging, etc., which are required for state-of-the-art performance. Our hypothesis for future work is that such features can be described and implemented by *composing* instances of these basic variants.

2 System Model and Terminology

Next, we present an informal, high-level overview of the system model and terminology. Table 1 overviews our notation.

Stores and transactions. At the core of the model is an abstract mutable shared memory, called a *store*, $\sigma \in \Sigma$. A store follows the common API shown in Figure 1. Method lookup returns the value that store σ associates with key k at time t; an absent mapping returns $\perp$ (i.e., initially, every key maps to $\perp$). Update method doUpdate applies a new *effect* δ to that key's entry in the store, in order to update the value (see below). Successfully invoking doCommit makes the updates of the current transaction visible with a commit timestamp noted `ct`.

Updating a key k under timestamp t creates a new version mapped at index (k, t). A mapping is write-once, and remains valid until the next mapping, if any. For example, suppose store σ updates a version of key k at time $t = 100$ with an assignment of 27.[1] Then, $\text{lookup}(\sigma, k, 101)$ should return 27. If there are no other versions between 100 and 110, $\text{lookup}(\sigma, k, 111)$ should also return 27. If the next mapping is at timestamp 120, to incr10, then $\text{lookup}(\sigma, k, 121)$ should return 37.

A client (left unspecified by our model) accesses the store in the context of a *transaction*, a sequence of begin, lookup, update and commit/abort actions. A transaction reads from a consistent snapshot of the store, and makes its effects visible in the store by committing the transaction atomically (all-or-nothing). We defer a more detailed discussion of transactions to Section 3.

Keys, values and timestamps. Keys, values, and timestamps are opaque types. Keys compare for equality only. Timestamps are partially ordered by $\leq$; we say timestamps are *concurrent*, if they are not ordered, i.e., $t_1 \parallel t_2 \stackrel{\text{def}}{=} t_1 \not\leq t_2 \wedge t_2 \not\leq t_1$. Note that we do not assume a global clock.

An *ordered timestamp pair* (OTSP) is of the form $(d, v) \in \mathit{Timestamp} \times \mathit{Timestamp}$, where $d \leq v$, called dependence and version respectively. We define a strict partial order relation $\prec_{OTSP}$ over OTSPs, as follows: $(d_1, v_1) \prec_{OTSP} (d_2, v_2) \stackrel{\text{def}}{=} v_1 < d_2$. Two OTSPs are concurrent if they cannot be ordered by $\prec_{OTSP}$:

$$
\begin{aligned}
&(d_1, v_1) \parallel_{OTSP} (d_2, v_2) \\
&\qquad \stackrel{\text{def}}{=} (d_1, v_1) \not\prec_{OTSP} (d_2, v_2) \wedge (d_2, v_2) \not\prec_{OTSP} (d_1, v_1) \\
&\qquad \equiv (d_2 \leq v_1 \vee d_2 \parallel v_1) \wedge (d_1 \leq v_2 \vee d_1 \parallel v_2)
\end{aligned}
$$

Effects. Classically, an update simply assigns a new value to the key, as in $k := 27$. Such an assignment creates a new version of k with value 27.

Many recent stores [2, 3, 12] support a more general concept of update, which we call *effect*. Applying effect δ to a current value v computes a new value $\delta(v)$. For instance, δ_{incr10} is the effect that adds 10 to some value. An assignment is thus a constant effect; for example, $\delta_{\text{assign}_{27}}(v)$ yields 27 whatever the current value v.

If a sequence of updates with effects $\delta, \delta', \ldots, \delta''$ has been applied to key k, then a store is expected to return value $(\delta \odot \delta' \odot \ldots \odot \delta'')(\perp)$ when queried.[2] We say a sequence of effects is *proper* if it starts with an assignment and therefore evaluates to a value, or $\perp$, when applied to $\perp$; i.e., it does not depend on any preceding effects. We assume that every history forms a proper sequence.

An assignment masks any previous effects to the same key: $\forall \delta, (\delta \odot \delta_{\text{assign}}) = \delta_{\text{assign}}$; therefore effects that precede

[1] Assuming integer timestamps, for the sake of example.

[2] Operator $\odot$ is pronounced "apply;" it is equivalent to classical functional composition $\circ$, but associates left to right for convenience: $\delta \odot \delta' = \delta' \circ \delta$. Note that $\odot$ is associative.

k, t, v	$\in Key, \in Timestamp, \in Value$	key, timestamp, value
δ	$\in Effect = Value_\perp \to Value$	effect
$\perp$	$\notin Effect$	absence of effect
$\odot$	$\in Effect_\perp \times Effect_\perp \to Effect_\perp$	effect composition
x	$\in X$	Transaction
τ	$\in TxnID$	transaction identifier
$\mathtt{st}$	$\in Timestamp$	snapshot timestamp (of transaction)
$\mathcal{R}$	$\in \mathcal{P}(Key)$	read set, keys read in transaction
$\mathcal{W}$	$\in \mathcal{P}(Key)$	write/dirty set, keys modified in txn
$\mathcal{B}$	$\in EffectBuf = Key \to Effect_\perp$	effect buffer (of transaction)
$\mathtt{ct}$	$\in Timestamp$	commit timestamp (of transaction)
$(\tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct})$	$\in TxnDesc$	transaction descriptor
$\mathcal{X}_\mathrm{a}, \mathcal{X}_\mathrm{c}, \mathcal{X}_\mathrm{r}$	$\subseteq TxnDesc$	aborted, committed, running transactions

Table 1. Overview of notation.

$$(\perp \odot \delta) = (\delta \odot \perp) = \delta \quad \text{(non-effect)}$$
$$(\delta \odot \delta') \odot \delta'' = \delta \odot (\delta' \odot \delta'') \quad \text{(associativity)}$$
$$(\lambda v.c) \odot \delta = \lambda v.\delta(c) \quad \text{(compacting a proper sequence)}$$

Figure 2. Effect composition.

the last assignment in a sequence can be safely ignored. Conversely, any proper sequence is equivalent to a single assignment. For instance, $\delta_{\mathrm{assign}_{27}} \odot \delta_{\mathrm{incr10}} = \delta_{\mathrm{assign}_{37}}$. This justifies checkpointing a proper sequence into a single assignment. Figure 2 summarizes the rules of effect composition.

Visibility and concurrent effects. Effects are ordered by the *visibility* relation $\delta \prec \delta'$ (read "δ is visible to δ' "), defined as follows:

- $\delta \prec \delta'$ if both belong to the same transaction, and δ is before δ'.
- $\delta \prec \delta'$ if they belong to different transactions, x and x' respectively, where x has committed, and x is before x' in OTSP order, i.e., $x.\mathtt{ct} < x'.\mathtt{st}$.

Visibility is a strict partial order. Two effects are concurrent if they are not mutually ordered by visibility.

Some data types support concurrent effects thanks to a merge operator on effects. To ensure convergence, the merge operator is required to be commutative, associative, and idempotent (CAI) [12].

In the presence of concurrent effects, the value expected of key k is results from applying, from the initial $\perp$, the visible effects related to k, in visibility order, while mergeing concurrent effects.

Concurrent data types. As an aside, note that classical sequential data types generally disallow concurrent updates, leaving merge undefined. These data types require a strong consistency model, where updates occur in some serial order.

Data types that merge concurrent effects do exist [12]. There are also data types designed with non-assignment effects [1].

To provide the CAI properties, the implementation of an effect typically needs to carry metadata, e.g., to provide idempotence or determine causal relationships between updates. For example, the classical last-writer-wins approach supports concurrency by merging concurrent assignments under some deterministic total order (e.g., timestamp order), and retaining only the one with the highest timestamp. Another example is a counter supporting concurrent increment and decrement effects, which uses a vector of sets of effects, with one entry per (concurrent) client [12]. This representation ensures that a given increment or decrement is applied only once.

3 Semantics of Transactions

A transaction $x \in X$ is a sequence of effects. We associate to a transaction its *transaction descriptor* $(\tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct})$. It reads from a snapshot timestamped by *snapshot timestamp* $\mathtt{st}$. Its *write buffer* $\mathcal{W}$ lists the keys that it *dirtied*, i.e., modified. It may commit with a *commit timestamp* noted $\mathtt{ct}$.

However, the effects of a running transaction are not visible from outside (*isolation*). Within a transaction, an effect *visible* to another one that executes after it (the "read-your-own-writes" property [15]). The semantics formalise this by staging effects to an *effect buffer* $\mathcal{B}$. A transaction terminates in an all-or-nothing manner, by either an *abort* that discards its effect buffer, or by a *commit* that makes all its effects visible to later transactions at once. *Atomicity* is formalised by assigning the same, unique, *commit timestamp* to all its effects. The commit timestamp of a running or aborted transaction is irrelevant and can be arbitrary, marked by _.

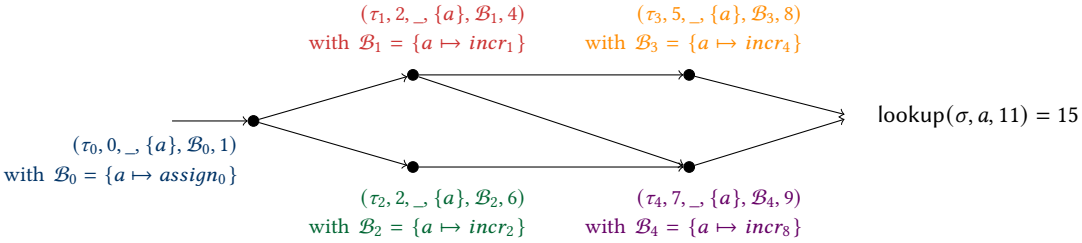
Every first read of some key comes from a same *snapshot*, identified by its *snapshot timestamp* noted $\mathtt{st}$. Transactions that are *visible* in the snapshot are those that committed strictly before its snapshot timestamp.

For interested readers, we provide a small-step operation semantics of transactions in the Appendix in Figure 6.

(a) History.

$$\tau_0 : \text{beginTxn}(0) \cdot \tau_0 : \text{update}(a, assign_0) \cdot \tau_0 : \text{commit}(\tau_0, 1) \cdot \tau_1 : \text{beginTxn}(2) \cdot \tau_1 : \text{update}(a, incr_1) \cdot \tau_2 : \text{beginTxn}(2) \cdot \tau_2 : \text{update}(a, incr_2) \cdot \tau_1 : \text{commit}(\tau_1, 4) \cdot \tau_3 : \text{beginTxn}(5) \cdot \tau_2 : \text{commit}(\tau_2, 5) \cdot \tau_3 : \text{update}(a, incr_4) \cdot \tau_4 : \text{beginTxn}(7) \cdot \tau_3 : \text{commit}(\tau_3, 8) \cdot \tau_4 : \text{update}(a, incr_8) \cdot \tau_4 : \text{commit}(\tau_4, 9)$$

(b) Dependency graph.



(c) Map store for key a at $t = 11$.

$$\sigma[a, t] = \begin{cases} (0, assign_0) & \text{for } t = 1 \\ (2, assign_1) & \text{for } t = 4 \\ (2, assign_2) & \text{for } t = 6 \\ (5, assign_5) & \text{for } t = 8 \\ (7, assign_{11}) & \text{for } t = 9 \\ \bot & \text{otherwise} \end{cases}$$

(d) Journal store at $t = 11$.

$$\begin{aligned} \sigma = {} & (\text{beginTxn}, \tau_0, 0) \triangleright (\text{update}, \tau_0, a, assign_0) \triangleright (\text{commit}, \tau_0, 1) \\ & \triangleright (\text{beginTxn}, \tau_1, 2) \triangleright (\text{update}, \tau_1, a, incr_1) \triangleright (\text{beginTxn}, \tau_2, 2) \\ & \triangleright (\text{update}, \tau_2, a, incr_2) \triangleright (\text{commit}, \tau_1, 4) \\ & \triangleright (\text{beginTxn}, \tau_3, 5) \triangleright (\text{commit}, \tau_2, 5) \triangleright (\text{update}, \tau_3, a, incr_4) \\ & \triangleright (\text{beginTxn}, \tau_4, 9) \triangleright (\text{commit}, \tau_3, 8) \triangleright (\text{update}, \tau_4, a, incr_8) \\ & \triangleright (\text{commit}, \tau_4, 9) \end{aligned}$$

Figure 3. Example of execution trace. The history (a) shows the order in which the transactional operations are executed. The dependency graph (b) visualizes the partial order of the transactions. The map (c) and journal (d) show the different stores after the history executed.

Figure 3 illustrates these semantics with an example. The history (Figure 3a) shows a sequence of (atomic) transactional steps; the steps for concurrently executed transactions, like τ_1 and τ_2, are interleaved. For simplicity, the example updates a single key a of integer type. Figure 3b visualizes the transactions in a dependency graph.

4 Store Models

This section discusses two basic variants implementing the general store API. We aim to model their most essential, primitive properties, abstracting away as much complexity as possible.

4.1 Map-based store semantics

The *map-based store* models a classic versioned key-value store as a random-access map, located either in memory or on disk. It is restricted to contain only values, which (in our model) are represented as assignment effects. Versions of a key are distinguished by their version timestamps. Such a store maps a *(key, version timestamp)* pair, to an *(assignment effect, dependency timestamp)* pair.

Figure 4 summarises its semantics. A map store defers its updates to commit time, and committing atomically copies the transaction's *effect buffer* into a new version of the corresponding keys. lookup searches for the most recent assign effect directly from the map; both doBegin and doUpdate are no-ops. Figure 3c illustrates the contents of a map store, after the history in Figure 3a.

In more detail, mapping $\sigma[k, v] = (d, \delta)$ associates a *versioned key* (k, v) with a *dependent effect* (d, δ). Here, v is a version timestamp, δ is an assignment (a map store does not support non-assignment effects) associated with *metadata* d, called dependence timestamp, where $d \leq v$. Versions are ordered by their OTSPs (d, v), i.e.,

$$\sigma[k, v_1] \prec_{MS} \sigma[k, v_2] \stackrel{\text{def}}{=} v_1 < \sigma[k, v_2].d$$

When a transaction commits, method doCommit of a map store eagerly creates a new version for each key modified by the transaction. The version identifier is the transaction's commit timestamp `ct`, and it is associated with metadata `st`, the transaction's *snapshot timestamp*. It returns a store unchanged except for the new versions.[3]

The versions of k visible from the current transaction are the set $V = \{\sigma[k, v] \mid v < t\}$. Since a map contains only

[3] A simpler specification might copy all keys, not just the dirty ones. However, this is not well-defined when the space of keys is unbounded (e.g., if keys are arbitrary strings).

$$
\begin{aligned}
\sigma &\in (\mathit{Key} \times \mathit{Timestamp} \rightarrow \mathit{Timestamp} \times \mathit{Assign}) \\
\mathsf{doBegin}(\sigma, \_, \_) &= \sigma \\
\mathsf{lookup}(\sigma, k, t) &= \mathsf{merge}\left(\max_{\prec_{MS}}(\{\sigma[k, v] \mid v < t\})\right) \\
\mathsf{doUpdate}(\sigma, \_, \_, \_) &= \sigma \\
\mathsf{doCommit}(\sigma, \_, \mathtt{st}, \_, \mathcal{W}, \mathcal{B}, \mathtt{ct})[k, t] &= \begin{cases} (\mathtt{st}, \mathcal{B}[k]) & \text{if } k \in \mathcal{W} \wedge t = \mathtt{ct} \\ \sigma[k, t] & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4. Semantics of map-based store.

assignments, $\mathsf{lookup}(\sigma, k, t)$ can omit all but the most recent one in this set in visibility order, noted $\max_{\prec_{MS}}(V)$. To determine the returned value, any concurrent effects are merged (as explained in Section 2), and the resulting assignment effect is then applied to $\perp$ to obtain a value.

The map store defers updates to commit time; therefore doUpdate leaves the store unchanged.

In practice, many existing database backends contain an in-memory map store, for simplicity and fast reads. To persist a map store, it suffices to write it to disk periodically; however such a large write can be slow and is not natively crash-atomic.

4.2 Journal-based store semantics

An alternative store variant is the journal-based store, which logs its updates incrementally to a sequential file.[4] This design is optimised for fast disk writes, and has good crash-tolerance properties. It is also friendly to non-assignment effects. However, to lookup the value of a key can be slow, as its semantics is to read the journal and applies effects sequentially.

Figure 5 gives the formal semantics of a journal store. A journal store is a finite sequence $\sigma = [e_1, e_2, \dots]$ of records of type BeginTxnRec, update and commit, initially empty. Function doBegin appends a record with transaction identifier τ and snapshot timestamp `st`. doUpdate appends an update record that contains transaction identifier τ, key k, and effect δ. Similarly, doCommit appends a commit record containing the transaction identifier τ, snapshot timestamp `st` and commit timestamp `ct`.

The real action is in $\mathsf{lookup}(\sigma, k, t)$, which accumulates the effects to key k that committed strictly before t. To formalise the procedure is somewhat complex.

- Procedure $\mathit{poststate}_\sigma\ (r, k)$ computes the state of key k after a record r in journal σ takes effect. Records take effect in $\prec_{JS}$ order.
- In $\prec_{JS}$ order, a record of type beginTxn has any number of immediate predecessors; other types of records have a single one.
- The poststate of an update record with key k and effect δ is computed by taking the poststate of its immediately-preceding record, and applying δ.
- The poststate of a beginTxn is the merge of poststate of its immediate predecessors.
- Otherwise, the poststate is the same as that of the immediate predecessor; i.e., updates for other keys are ignored, as well as commit records.

Note that a poststate can be computed in a single left-to-right pass over the journal, because a commit record always appears before the beginTxn of a transaction that depends on it. Note also that records are single-assigned and that $\mathit{poststate}_\sigma$ is a function; therefore a practical implementation may use a cache.

Figure 3d illustrates the contents of a journal store after execution of the history in Figure 3a. To obtain the value for $\mathsf{lookup}(\sigma, x, 11)$, we calculate recursively the $\mathit{poststate}_\sigma$s for τ_3 and τ_4 and merge the results:

$$
\begin{aligned}
\mathsf{lookup}(\sigma, a, 11) &= \mathsf{assign}_0 \odot \mathsf{merge}\ (\ \{\ \{\mathsf{incr}_5, \mathsf{incr}_{11}\}\ \}\ ) \\
&= \mathsf{assign}_{15}
\end{aligned}
$$

As explained earlier (Section 2) the implementation of effects must ensure that the incr_1 from τ_1 executes once only in the merged value, even though the history contains two paths to lookup.

5 Formal model in Coq

So far, we have formalized a major part of the definitions presented in Section 2, 4 and 3 in the proof assistant Coq with the goal to formally verify their correctness. Our Coq codebase comprises currently around 2k LOC, without using any external libraries other than the standard library. Our goal will be to show that all store variants yield the expected values under the respective isolation model using bisimulation.

There are several reasons for choosing to use interactive theorem proving. A positive aspect about the Coq formalization is the high level of abstraction. We can define constructs with desired properties without the need to provide a specific implementation. This is in contrast to traditional programming languages, where interfaces or abstract classes can be defined, but it is usually not possible to restrict the behavior

[4] This describes a *redo log*; an "undo log" would store inverse effects instead.

$$\sigma \in \{[e_1, e_2, \ldots] \mid e_i \in (\{(\mathsf{beginTxn}, \tau, \mathtt{st}) \mid \tau \in \mathit{TxnID}, \mathtt{st} \in \mathit{Timestamp}\} \cup \{(\mathsf{update}, \tau, k, \delta) \mid \tau \in \mathit{TxnID}, k \in \mathit{Key}, \delta \in \mathit{Effect}\} \cup \{(\mathsf{commit}, \tau, \mathtt{ct}) \mid \tau \in \mathit{TxnID}, \mathtt{st}, \mathtt{ct} \in \mathit{Timestamp}\})\}$$

Note $\max_{\prec_{JS}}(r)$ the immediate predecessor(s) of record r in $\prec_{JS}$ order (beginTxn may have any number of mmediate predecessors; update and commit have exactly one immediate predecessor). The $\mathit{poststate}_\sigma$ function computes the state of a key k after record r takes effect, as follows:

$$\mathit{poststate}_\sigma\ (r, k) \quad = \quad \begin{cases} \mathsf{merge}\ (\ \{\ \mathit{poststate}_\sigma\ (r', k) \mid r' \in \max_{\prec_{JS}} r\ \}\ ) & \text{if } r = (\mathsf{beginTxn}, \_, \_) \\ \mathit{poststate}_\sigma\ (\max_{\prec_{JS}} r, k) & \text{if } r = (\mathsf{update}, \_, k', \_) \wedge k \neq k' \\ \mathit{poststate}_\sigma\ (\max_{\prec_{JS}} r, k) \odot \delta & \text{if } r = (\mathsf{update}, \_, k, \delta) \\ \mathit{poststate}_\sigma\ (\max_{\prec_{JS}} r, k) & \text{if } r = (\mathsf{commit}, \_, \_) \end{cases}$$

The journal operations are defined as follows (where $\triangleright$ represents the append operation):

$$\begin{aligned} \mathsf{doBegin}(\sigma, \tau, \mathtt{st}) &= \sigma \triangleright (\mathsf{beginTxn}, \tau, \mathtt{st}) \\ \mathsf{lookup}(\sigma, k, t) &= \mathit{poststate}_\sigma\ (r, k) \text{ where } r = (\mathsf{beginTxn}, \_, t) \\ \mathsf{doUpdate}(\sigma, \tau, k, \delta) &= \sigma \triangleright (\mathsf{update}, \tau, k, \delta) \\ \mathsf{doCommit}(\sigma, \tau, \_, \_, \_, \_, \mathtt{ct}) &= \sigma \triangleright (\mathsf{commit}, \tau, \mathtt{ct}) \end{aligned}$$

Figure 5. Semantics of journal-based store.

of implementations. As an example of this, in our formalization we defined timestamps to be some arbitrary data type that comes with an ordering, as described in section 2, and for which equality is decidable. No assumption about the implementation is made, and refining the specification to specific instances can be done independently.

However, the main benefit of reasoning about our system in a formal context is the required level of detail and precision, which is typically not attained by pen-and-paper proofs. It not only makes spotting mistakes easier and earlier, but it also forced us to pay attention to specific corner cases. This already proved to be useful during this initial formalization phase. For example, in earlier versions the journal-based store explicitly maintained both a dependency and commit timestamp, while the map-based store did this implicitly. This oversight lead to an inconsistency when merging concurrent effects, since the map-based store carries too little information. Only when trying to formalize the map-based store, we noticed this mismatch.

While it is arguably more effort to formalize everything in a proof assistant rather than using pen-and-paper definitions and proofs, we believe that the benefits of obtaining a verified design outweigh the costs.

6 Discussion and Outlook

Formal methods have been successfully employed for proving (distributed) systems correct [4, 6, 9, 16]. The focus of these approaches has been the verification of safety and liveness properties for different types of distributed systems and their implementations. The work presented in [4] is closest to our approach, as it proves the correctness of a transaction library. However, their work targets the verification of a specific library and its sophisticated optimisations, while we aim to take a compositional approach to proving a generic database backend.

Typically, developers need to provide a high-level specification that is then refined in one or more steps, while the corresponding proofs are correspondingly refined. For example, Verdi [16] is a framework to implement and specify systems under different network semantics in Coq; starting from an idealized system, proof obligations are then transformed for more and more complex fault models. Finally, an implementation can then be generated from the specification. Our approach differs in two major aspects. The focus of our work is the correct design and implementation of a central system component, not a distributed protocol. This component is typically simplified in the before mentioned verification frameworks. However, a correct and efficient implementation is essential in any (distributed) datastore. Further, instead of refinement, we propose a compositional approach to construct more and more complex implementations. This helps system designers to select and incrementally add features such as caching, write-ahead logging, or checkpointing. Reducing these features to their essence helps us extracting their actual (and not incidental) requirements and to re-purpose metadata in different contexts.

Tools exist to compile a Coq specification to executable code [10, 14, 16]. We are not taking this path for pragmatic reasons: it is too far from the ordinary programmer's experience and as of today still requires extensive manual intervention. Instead, we manually transcribe the specification to Java, verbatim, resisting the temptation to optimise. We will check through testing that the implementation behaves like the specification, and that variants that were proved equivalent do have the same runtime behaviour.

References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *Journal of Parallel and Dist. Comp.* 111 (Jan. 2018), 162–173. https://doi.org/10.1016/j.jpdc.2017.08.003

[2] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2017. *Pure Operation-Based Replicated Data Types.* ArXiv e-print 1710.04469. arXiv Computing Research Repository (CoRR). http://arxiv.org/abs/1710.04469

[3] Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. 2013. *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure.* Microsoft Patterns & Practices. https://www.microsoft.com/en-us/download/details.aspx?id=34774

[4] Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2023. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 871–886. https://www.usenix.org/conference/osdi23/presentation/chang

[5] Facebook. Year of Access. RocksDB. https://github.com/facebook/rocksdb.

[6] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael Lowell Roberts, Srinath T. V. Setty, and Brian Zill. 2017. IronFleet: proving safety and liveness of practical distributed systems. *Commun. ACM* 60, 7 (2017), 83–92. https://doi.org/10.1145/3068608

[7] Huachao Huang. 2017. How We Found a Data Corruption Bug in RocksDB. Blog post. https://www.pingcap.com/blog/how-to-find-a-rocksdb-bug/.

[8] Kyle Kingsbury and Kit Patella. 2023. Jepsen reports. Blog post. https://jepsen.io/analyses/.

[9] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (1994), 872–923. https://doi.org/10.1145/177492.177726

[10] Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5028)*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.). Springer, 359–369. https://doi.org/10.1007/978-3-540-69407-6_39

[11] Redis. 2024. Redis. https://github.com/redis/redis.

[12] 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. https://doi.org/10.1007/978-3-642-24550-3_29

[13] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*. Assoc. for Computing Machinery, Cascais, Portugal, 385–400. https://doi.org/10.1145/2043556.2043592

[14] Akira Tanaka, Reynald Affeldt, and Jacques Garrigue. 2018. Safe Low-level Code Generation in Coq Using Monomorphization and Monadification. *J. Inf. Process.* 26 (2018), 54–72. https://doi.org/10.2197/IPSJJIP.26.54

[15] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*. IEEE Computer Society, Austin, Texas, USA, 140–149.

[16] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 357–368. https://doi.org/10.1145/2737924.2737958

A Formal semantics of transactions

Figure 6 shows the transition rules for the small-step operational semantics of a transaction system build on a store. The specification is fully formal and unambiguous: we find it invaluable to reason about the system, and it is easily translated to the language of a proof tool such as Coq. Most interestingly, it can be read as pseudocode, as we explain now.

The semantics are written as a set of rules. Each rule represents an indivisible state transition; i.e., there are no intermediate states from a semantic perspective, and any intermediate states in the implementation must not be observable.

The system state is represented as a tuple $(\sigma, \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r)$ consisting of a store, its field, and the sets of aborted, committed, and running transactions' descriptors.

A rule consists of a set of *premises* above a long horizontal line, and a *conclusion* below. A premise is a logical predicate referring to state variables. A variable without a prime mark refers to before the state before the transition (pre-state); a primed variable refers to state after the transition (post-state). Thus a premise that uses only non-primed variables is a pre-condition on the pre-state; if it contains a primed variable, it is a post-condition that constrains the post-state.

If the premises are satisfied, the state-change transition described by the conclusion can take place. A label on the transition arrow under the line represents a client API call. Thus, a rule can be seen as terse pseudocode for the computation to be carried out by the API.

A.1 Example

To explain the syntax, consider for example rule BeginTxn. The conclusion describes the transition made by API command beginTxn(st) from pre-state $(\sigma, \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r)$ on the left of the arrow $\xrightarrow[\tau]{\texttt{beginTxn(st)}}$, to post-state $(\sigma', \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r')$ on the right. Note that in the right-hand side of this conclusion, only $\mathcal{X}_r$ is primed, indicating that the other elements of the state do not change.

A.2 Parameters

The rules describe a transaction system, which is a tuple $(\sigma, \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r)$ consisting of a store σ, its associated field $\mathcal{F}$, and sets of transaction descriptors $\mathcal{X}_a$, $\mathcal{X}_c$, and $\mathcal{X}_r$, which keep track of aborted, committed and running (ongoing) transactions respectively. A transaction descriptor is a tuple $(\tau, \texttt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \texttt{ct})$ of transaction identifier τ, its *snapshot*

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

$$\frac{\begin{array}{c}\mathcal{X}_r = \mathcal{X}_r'' \uplus \{(\tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \_)\} \\ k \notin \mathcal{R} \qquad \mathsf{lookup}(\sigma, k, \mathtt{st}) = \delta \qquad \mathcal{R}' = \mathcal{R} \uplus \{k\} \qquad \mathcal{B}' = \mathcal{B}[k \leftarrow \delta] \qquad \mathcal{X}_r' = \mathcal{X}_r'' \uplus \{(\tau, \mathtt{st}, \mathcal{R}', \mathcal{W}, \mathcal{B}', \_)\}\end{array}}{(\sigma, \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r) \underset{\tau}{\rightarrow} (\sigma, \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r')} \text{ [InitKey]}$$

$$\frac{\mathcal{X}_r = \mathcal{X}_r'' \uplus \{(\tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \_)\} \qquad k \in \mathcal{R} \qquad \mathcal{B}[k] = \delta \in \mathit{Assign} \qquad v = \delta(\bot)}{(\sigma, \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r) \xrightarrow[\tau]{\mathsf{read}(k) \rightarrow v} (\sigma, \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r)} \text{ [Read]}$$

$$\frac{\begin{array}{c}\mathcal{X}_r = \mathcal{X}_r'' \uplus \{(\tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \_)\} \qquad k \in \mathcal{R} \\ \sigma' = \mathsf{doUpdate}(\sigma, \tau, k, \delta) \qquad \mathcal{W}' = \mathcal{W} \cup \{k\} \qquad \mathcal{B}' = \mathcal{B}[k \leftarrow \mathcal{B}[k] \odot \delta] \qquad \mathcal{X}_r' = \mathcal{X}_r'' \uplus \{(\tau, \mathtt{st}, \mathcal{R}, \mathcal{W}', \mathcal{B}', \_)\}\end{array}}{(\sigma, \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r) \xrightarrow[\tau]{\mathsf{update}(k, \delta)} (\sigma', \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r')} \text{ [Update]}$$

$$\frac{\mathcal{X}_r = \mathcal{X}_r' \uplus \{(\tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \_)\} \qquad \mathcal{X}_a' = \mathcal{X}_a \uplus \{(\tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \_)\}}{(\sigma, \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r) \xrightarrow[\tau]{\mathsf{abort}()} (\sigma, \mathcal{X}_a', \mathcal{X}_c, \mathcal{X}_r')} \text{ [Abort]}$$

$$\frac{\begin{array}{c}\mathcal{X}_r = \mathcal{X}_r' \uplus \{(\tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \_)\} \qquad \forall x \in \mathcal{X}_c, x.\mathtt{ct} \neq \mathtt{ct} \\ \mathtt{st} \leq \mathtt{ct} \qquad \mathit{NoInversion}(\mathtt{ct}) \qquad \sigma' = \mathsf{doCommit}(\sigma, \tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct}) \qquad \mathcal{X}_c' = \mathcal{X}_c \uplus \{(\tau, \mathtt{st}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct})\}\end{array}}{(\sigma, \mathcal{X}_a, \mathcal{X}_c, \mathcal{X}_r) \xrightarrow[\tau]{\mathsf{commit}(\tau, \mathtt{ct})} (\sigma', \mathcal{X}_a, \mathcal{X}_c', \mathcal{X}_r')} \text{ [Commit]}$$

Figure 6. Operational semantics of transactions. $\uplus$ denotes disjoint set union.

timestamp st, its *read set* $\mathcal{R}$, its *write set* $\mathcal{W}$, its *effect buffer* $\mathcal{B}$, and its *commit timestamp* ct.[5]

The two timestamps define visibility between transactions, as defined previously (Section 2). Initially, after rule BeginTxn, the sets and the effect buffer are empty and the commit timestamp is invalid. For each key that is accessed, rule InitKey initialises the buffer, and rule Update updates it. Computation of the actual commit timestamp may be deferred to the Commit rule.

The semantic rules are parameterised by commands lookup, doUpdate, and doCommit, specified in Figure 1. These commands are specialised for each specific store variant: the map-based variant in Section 4.1, the journal-based variant in Section 4.2.

A.3 Transaction begin

We now consider each rule in turn.

BeginTxn describes how API beginTxn() begins a new transaction with snapshot timestamp st. The snapshot of the new transaction is timestamped by st, passed as an argument; remember that a snapshot includes all transactions that committed with a strictly lesser commit timestamp.

[5]To simplify notation, we may write τ.st or τ.ct, for instance, for the corresponding elements of the descriptor whose transaction identifier is τ.

The first premise chooses a fresh transaction identifier τ. The last premise ensures that the appropriate transaction descriptor is in the post-state set of running transactions.

As the transition is labeled by τ, multiple instances of BeginTxn are mutually independent and might execute in parallel, as long as each such transition appears atomic.

A.4 Reads and writes

Reading or updating operate on the transaction's effect buffer $\mathcal{B}$, which must contain the relevant key.

Rule InitKey specifies a *buffer miss*, which initialises the buffer for some key k. As it does not have an API label, it can be called arbitrarily. It modifies only the current transaction's descriptor. Its first premise takes the descriptor of the current transaction τ from the set of running transactions $\mathcal{X}_r$. The second one checks that k is not already in the read set, ensuring that the effect buffer is initialised once per key. The third reads the store by using lookup (specific to a store variant). Next, a premise updates the read set, and another initialises the effect buffer with the return value of lookup. The final premise puts the transaction descriptor, containing the updated read set and effect buffer, back into the descriptor set of running transactions.

In Rule Read, API read(k) returns a value from the effect buffer. It does not modify the store. The first premise is as above. The second one requires that the key is in the read

set, thus ensuring that InitKey has been applied. The next two premises extract k's mapping from the effect buffer and compute the corresponding return value.

Note the clause $\delta \in \mathit{Assign}$. It requires that, previously to reading, the application has initialised the store with an assignment to k (possibly followed by other effects; such a sequence resolves to an assignment, by associativity, as explained earlier). Otherwise, lookup would return either $\perp$ (if the key has not been initialised at all) or a non-assignment (if the application has stored only non-assignments). We leave the burden of initialisation to the application to simplify the semantics; logically, it's an axiom.

In Rule Update, API call $\mathtt{update}(k, \delta)$ applies effect δ to key k. It updates both the store and the transaction descriptor. The first two clauses are similar to Read, and similarly require a buffer miss if the key has not been used before (avoiding blind writes). It updates the effect buffer, ensuring that the transaction will read its own writes, and puts the key in the write set. It calls the variant-specific command doUpdate, discussed later in the context of each variant.

A.5 Transaction termination

A transaction terminates, either by aborting without changing the store, or by committing, which applies its effects atomically to the store.

Rule Abort moves the current transaction's descriptor from $\mathcal{X}_\mathrm{r}$ to $\mathcal{X}_\mathrm{a}$, marking it as aborted. It does not make any other change.

API call $\mathtt{commit}(\tau, \mathtt{ct})$ takes a commit timestamp argument. It is enabled by rule Commit, which modifies the store, the running set, and the committed set. The first premise is as usual. Commit timestamp $\mathtt{ct}$ must satisfy the constraints stated in the next three premises: it is unique (it does not appear in $\mathcal{X}_\mathrm{c}$); it is greater or equal to the snapshot; the $\mathit{NoInversion}(\mathtt{ct})$ premise ensures that no already-committed or running transaction may depend on this one.

The latter premise aims to protect against the case where another transaction has read a value that this transaction has yet to write, because the transactions commit in the wrong order. To understand, consider the following anomalous example: *(i)* Transaction τ_1 has commit timestamp 1; *(ii)* Transaction τ_2 starts with snapshot timestamp $2 > 1$; thus τ_2 reads the updates made by τ_1; *(iii)* however, τ_1 is slow and its committed effects reach the store only after the read by τ_2. Clearly, this would be incorrect. To avoid this issue, τ_2 must not start until τ_1 has finalised its transition to committed. This requires synchronisation between concurrent transactions.

To avoid this issue, no still-running or committed transaction may read from the current transaction, i.e.,

$$\nexists x \in \mathcal{X}_\mathrm{c} \cup \mathcal{X}_\mathrm{r} : \mathtt{ct} < x.\mathtt{st} \wedge \mathcal{W} \cap x.\mathcal{R} \neq \varnothing$$

This expression requires to keep track of the read-set of committed transactions; to avoid this, we could check running transactions only, using the slightly stronger expression:

$$\nexists x \in \mathcal{X}_\mathrm{c} \cup \mathcal{X}_\mathrm{r} : \mathtt{ct} < x.\mathtt{st} \wedge (x \in \mathcal{X}_\mathrm{r} \implies \mathcal{W} \cap x.\mathcal{R} \neq \varnothing)$$

For simplicity, we choose to use the even stronger premise

$$\mathit{NoInversion}(\mathtt{ct}) \stackrel{\text{def}}{=} \nexists x \in \mathcal{X}_\mathrm{c} \cup \mathcal{X}_\mathrm{r} : \mathtt{ct} < x.\mathtt{st}$$

at the cost of aborting transactions unnecessarily.

Operation doCommit (specific to a store variant) provides the new state of the store; it should ensure that the effects of the committed transaction become visible in the store, labelled with the commit timestamp. Finally, the transaction descriptor, now containing the commit timestamp, is moved to the set of committed transactions.