Rigorous design and implementation of full-featured database backends

SAALIK HATIA, Sorbonne-Université (LIP6), France

ANNETTE BIENIUSA, TU Kaiserslautern, Germany

GUSTAVO PETRI, ARM, United Kingdom

CARLA FERREIRA, Universidade NOVA de Lisboa, Portugal

MARC SHAPIRO, Sorbonne-Université (LIP6) and Inria, France

Conceptually, a database backend is just a memory for storing and retrieving data. However, the quest for performance and reliability adds significant complexity. To ensure correctness, we derive our implementation from a formal specification. We start from the formalisation of a basic transactional API. We implement this specification in a verbatim, unoptimised persistent *map-based store*. We also formalise a journal-based variant, prove equivalence with the map semantics, and implement an unoptimised crash-tolerant *journal-based store*. Finally, we study the correct *composition* of stores. We leverage compositionality to create variants of increasing complexity: Features such as caching, checkpointing, and write-ahead logging are each represented as an instance of a map or journal store, restricted to a suitable domain, which is composed with another, larger store. This paper reports on the formal specification, the approach to correctness of specifications and implementation, and a preliminary experimental evaluation. This is work in progress.

Contents

STORY FOR PL PAPER

Title "Rigorous design by composition: a formally-verified transactional database backend design."

- DB backends are inherently complex.
- Simpler model by composition of primitive backends. Clarify/justify common features/optimisations: caching, journaling, checkpointing, layered stores, etc.
- Operational semantics of transactions and backend components.
- Operational model of composition.
- Show correctness of widely-used optimisations.

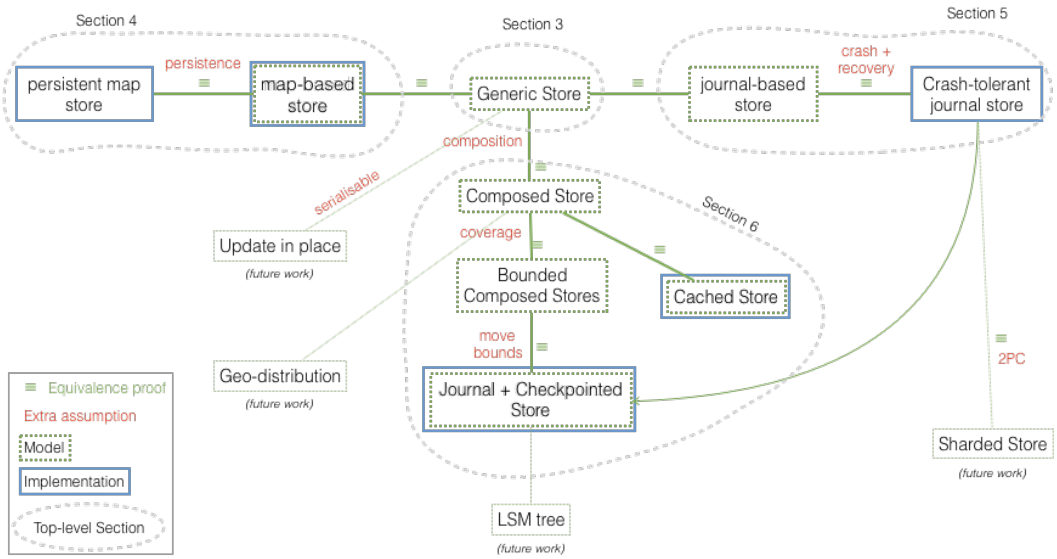

Fig. 1. Structure of the approach

1 INTRODUCTION

A database system manages a collection of digital data, to be organised, secured, updated, queried, and so on. An essential component is the *backend*, in charge of recording the lowest level of 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, reliability, and so on. For instance, the open-source RocksDB has *XXX* LOC, Redis is *XXX* LOC, and the AntidoteDB backend is 1300 LOC [**? ? ?**]. Any such complex software has bugs; database backend bugs are critical, possibly violating data integrity or security. ▹**(Marc:) Cite some episodes?**◃

To ensure correctness, formally-verified development is a powerful tool, but is often perceived as at odds with the performance requirements of backends. To our knowledge, formally-verified development of a serious backend has not been attempted before. One obstacle to the formal approach is the very complexity of a modern backend: fully specifying all the moving pieces is a daunting task.

This article reports an incremental approach to the rigorous and modular development of a full-featured backend. We first formalise the simplest specification of a store, a versioned key-value interface; we implement it directly, without any optimisation. As this specification makes versioning explicit, the state is a deterministic function; we show that the precise order of execution does not matter, and that multiple instances of store are mutually equivalent. We next consider a journal-based variant, again in its simplest form; we formally prove that it is equivalent with the base variant; and we implement it directly without optimisation. To support common optimisations such as caching, garbage collection, checkpointing and write-ahead logging, we proceed by *composition*, using the insight that each such optimisation can be represented as one of the basic stores, restricted to a suitable domain.

Our work combines a formal operational semantics (on paper), mechanised formal proofs (in Coq), and implementation (in Java).

In future work, we believe our same approach can justify sharding, geo-distribution, layered checkpoints in the style of LSM-Trees, and update-in-place under serialisability.

Our operational semantics specifies the rules for transactions under convergent causal consistency (TCC+), a variant of PSI [18]. In the style of MVCC (multi-value concurrency control), a transaction reads from a causally-consistent snapshot. It terminates by either committing atomically, or by aborting without effect. Our results generalise to stronger models such as SI or strict serialisability (which differ only by the conditions on transaction begin and commit).

The semantics specify the behaviour of a store's specialised book-keeping operations (called doUpdate, doCommit and lookup), implicity assuming infinite memory and no failures. To bound a store's memory footprint, we restrict its domain, and show equivalence over the restricted domain. Furthermore we exhibit a (classical) crash-tolerant journal implementation, which satisfies the failure-free semantics by masking crash-recovery events.

Tools exist to compile a Coq specification to executable code [**?**]. We did not take that path for pragmatic reasons, because it is too far from the ordinary programmer's experience. Instead, we manually transcribe the specification to Java, verbatim, resisting the temptation to optimise. We check through testing that it behaves like the specification, and that variants that were proved equivalent do have the same runtime behaviour. We report on the implementation throughout the paper, and an experimental evaluation in Section 10.

This paper focuses on safety properties, and does not consider security issues.

Our contributions can be summarised as follows:

- A formal model for backends; proof of equivalence between concurrent and sequential execution; correctness conditions for store composition.
- Specification, verification and implementation of variants supporting state-of-the-art features such as journaling, caching, crash-tolerant write-ahead logging and journal truncation.
- Experimental evaluation, showing that the rigorous approach does not preclude state-of-the-art performance, and justifying retrospectively the features.

▹**(TODO:) Update me.**◃ This paper progresses as illustrated in Figure 1. After this introduction, Section 3 formalises the system model and semantics. Section 2 explains our approach, and establishes some key lemmas. As a warm-up, we produce a correct map-based persistent store in Section **??**. Then, Section **??** addresses correct journal-based semantics and its implementation. In Section **??** we apply store composition to achieve caching, checkpointing, and truncating the journal. Our experimental evaluation is in Section 10. Section 11 discusses related work, and we discuss lessons learned in Section 12.

2 APPROACH

This work proceeds as follows. We first model two basic variants. The first variant is a classical map-based, versioned key-value store. As a transaction executes, it *eagerly* computes new key versions, which it copies into a map upon commit, labelled with the transaction's commit timestamp. Reading a key at some timestamp searches the map for the most recent corresponding version. We mechanise the proof that the map-based model satisfies the specification, i.e., that for any given trace Θ, a call to $\mathtt{read}(k, \mathtt{t})$ returns the expected value noted $\mathtt{safe}(\Theta,k,\mathtt{t})$ for any key and timestamp pair. We translate the formal specification verbatim into Java, forsaking optimisations, and test empirically that this implementation satisfies the specification.

Our second variant uses a journal. As a transaction executes, it appends individual effects to the journal, tagged with the transaction identifier. Committing appends a commit record to the

k	$\in Key$	Primitive Key
$\mathtt{t}$	$\in \mathsf{TS}$	Primitive Timestamp
v	$\in Value$	Primitive data-type values
σ	$\in \Sigma = Key \times \mathsf{TS} \rightarrow \mathcal{E}\!\mathit{ff}_\perp$	Store
$\mathcal{D}_\sigma$	$\subseteq Key \times \mathsf{TS}$	Domain of store σ
$\mathcal{S}$	$\{ \sigma_1, \sigma_2, \dots \}$	Composition of stores
δ	$\in \mathcal{E}\!\mathit{ff} = Value_\perp \rightarrow Value$	Effect
$\perp$	$\notin \mathcal{E}\!\mathit{ff}$	Absence of effect
$\delta_k^{\mathtt{t}}$	$\in \mathcal{TE}\!\mathit{ff} = \mathcal{E}\!\mathit{ff} \times Key \times \mathsf{TS}$	Effect to key k with commit timestamp $\mathtt{t}$
$\delta \prec \delta'$		Visibility
Θ	$\in (\mathcal{TE}\!\mathit{ff}, \prec)$	Trace
τ	$\in \mathcal{T}_{ID}$	Transaction identifier
$\mathtt{dt}$	$\in \mathsf{TS}$	Dependency timestamp (of transaction)
$\mathcal{E}$	$\in \mathcal{P}(Key)$	Dirty set = set of keys modified in transaction
$\mathcal{R}$	$\in Rbuf = Key \rightarrow Value_\perp$	Read buffer (of transaction)
$\mathtt{ct}$	$\in \mathsf{TS}$	Commit timestamp (of transaction)
$(\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}, \mathtt{ct})$	$\in \mathcal{T}_{DESC} = \mathcal{T}_{ID} \times \mathsf{TS} \times \mathcal{P}(Key) \times Rbuf \times \mathsf{TS}$	Transaction descriptor
$\mathcal{T}_\mathrm{a}$	$\subseteq \mathcal{T}_{DESC}$	Aborted transactions
$\mathcal{T}_\mathrm{c}$	$\subseteq \mathcal{T}_{DESC}$	Committed transactions
$\mathcal{T}_\mathrm{r}$	$\subseteq \mathcal{T}_{DESC}$	Running transactions
$\Pi_x \mathcal{T}$		Project set of tuples $\mathcal{T}$ along dimension x
$\sigma, \mathcal{T}_\mathrm{r}, \dots$		Before transition
$\sigma', \mathcal{T}_\mathrm{r}', \dots$		After transition

Table 1. Notation

journal, containing its commit timestamp. $\mathtt{read}(k, \mathtt{t})$ *lazily* applies the effects to k, recorded in the journal, that committed before $\mathtt{t}$.

Again, we prove mechanically that the journal-based store satisfies the abstract specification. We transcribe the specification to a Java crash-tolerant journal store, and show empirically that it satisfies the specification.

The above implementations are impractical, because they are slow, consume a lot of memory, and are not fault tolerant. Features such as caching, sharding, replication, checkpointing, journal truncation, etc., are known to improve performance. Our key insight is the realisation that each such feature can be viewed as is a composition of "little stores" that each implement the specification over some limited domain. For instance, we compose a truncated journal with a map-based checkpoint at the time of truncation.

We formulate rules for the composition of little stores. The domain of a composition is the union of the composed domains. Reading composes the results from all the little stores that are in domain, and writing updates all the stores in domain. Domains may overlap, as the little stores cannot contain conflicting information; furthermore, a useful composition must not have gaps, i.e., all composed reads must be safe.

A first application is caching. Consider some slow persistent store, either map- or journal-based, with a large domain $K \times T$. We compose it with a cache, an in-memory map-based store, whose domain is some small, arbitrary subset $\{(k_1, \mathtt{t}_1), (k_2, \mathtt{t}_2), \dots\} \subset K \times T$. For any key-version that is in the cache, both the cache and the larger store would read the same, safe, value, but the cache responds more quickly. Serving a cache miss grows the cache domain; eviction shrinks it

arbitrarily. We reuse the unoptimised Java map-based store, equipped with an associated domain. Unsurprisingly, our experiments confirm that caching improves response time; our contribution is acheiving this through rigorous programming discipline.

Our second application is to split the store between an "old generation" over time domain $[0, max\_old)$ and a "new generation" over $[min\_new, now())$, where $min\_new \leq max\_old$. As above, any key-version in the overlap will be the same in both stores. The bounds of generations may change, as long as the invariant $min\_new \leq max\_old$ holds, for instance by first increasing max_old by some amount, before increasing min_new by the same amount. A practical use case is a persistent map-based store as the old generation and a journal-based store as the new one. Advancing max_old corresponds to checkpointing, and advancing min_new to truncating the journal. Again, we combine the corresponding unoptimised Java implementations, and test that it satisfies the specification. We leave it as an exercise for the reader to address multiple levels of checkpointing, and to compose the split store with a cache, for performance.

In future work, we believe that the same theory of composition can justify sharding, local replicas at multiple sites, update-in-place (under strict serialisability), and layering in the style of LSM-Trees.

3 SYSTEM MODEL

▹(Marc 2023-01-07:) Give concrete examples◃

Table 1 summarises our notations, which are explained hereafter.

This study envisages different kinds of backend stores under a single system model. They follow a common API, described later, consisting of query method lookup and update methods doUpdate and doCommit.

A store runs under a general model of transactions, with Multi-Version Concurrency Control (MVCC). The transaction model guarantees Transactional Causal Plus Consistency (TCC+) [1, 19], also called Non-Monotonic Snapshot Isolation (NMSI) [3, 15]. Our results remain true for stronger consistency/isolation models, e.g., Parallel Snapshot Isolation, Snapshot Isolation, or Strict Serialisability; however this is mostly out of the scope of this paper.

3.1 Store

We define an abstract model of a backend storage system (hereafter abbreviated to just *store*) with a versioned key-value interface. A store is denoted through the type Σ, and ranged over by the meta-variable σ, which is a function from key (k) and timestamp ($\mathtt{t}$) pairs to (a means to compute) a value (v). In the case where either the key or the timestamp is not defined in the store, $\bot \notin Value$ represents the absence of a value.

Its API lookup$(\sigma, k, \mathtt{t})$ returns the value that store σ associates with key k at time $\mathtt{t}$.

3.1.1 Keys. Keys are denoted by the set *Key*, for which we use meta-variable k. Keys are opaque, and are compared only for equality.

3.1.2 Values. The set of values is denoted by *Value* and ranged over with the meta-variable v. For the purpose of this paper, the type of values is a collection of data types, with some constraints on semantics that are detailed later.

3.1.3 Effects. Classically, a store records values. We take a more general approach, where the store records operations that compute a value, called *effects*. We use the meta-variable δ to denote an effect, and the domain of effects Eff.

We define an effect to be a function from value to value. An example is "inc_10(arg)," which returns its argument plus 10. *Assignment* is modelled by the constant function; for example "assign_27(_)" is the effect that always returns 27. Abusing notation, we sometimes identify an assignment, e.g., and note simply 27 rather than "assign_27(_)."

3.1.4 Timestamps and clocks. Timestamps serve to identify events and to distinguish the versions of a key. We assume the following properties about timestamps:

- *Partial Order*: the set of timestamps (TS) defines a Partial Order, denoted $\leq$.
- *Lattice*: for any two timestamps, there exists for them a unique least upper bound and a unique greatest lower bound.

In general, except when mentioned specifically, we do not assume that timestamps are totally ordered. Two timestamps that are not mutually ordered, i.e., $\mathtt{t}_1 \not\leq \mathtt{t}_2 \wedge \mathtt{t}_2 \not\leq \mathtt{t}_1$, are said concurrent, noted $\mathtt{t}_1 \parallel \mathtt{t}_2$.

The type for timestamps is denoted by TS and generically instantiated by the meta-variable t. We use the following notation:

- $\max(\mathtt{t}_1, \mathtt{t}_2) \triangleq \mathtt{t}_1 \sqcup \mathtt{t}_2$ is the least upper bound of timestamps.
- We generalise max to a set in the obvious way. As a shorthand, $\max_{\mathcal{T}}(\mathtt{dt}) \triangleq \max(\Pi_{\mathtt{dt}}(\mathcal{T}))$ is the least-upper-bound of the projection of the set of tuples $\mathcal{T}$ over the dt dimension; and similarly for the ct dimension.
- $\mathtt{t}_1 < \mathtt{t}_2 \triangleq \mathtt{t}_1 \leq \mathtt{t}_2 \wedge \mathtt{t}_1 \neq \mathtt{t}_2$.
- $\mathtt{t}_1 \geqq \mathtt{t}_2 \triangleq \mathtt{t}_1 \not< \mathtt{t}_2$. $\geqq$ is read "greater, equal, or concurrent."

The notations min, > and $\leqq$ are defined symmetrically.

A vector of integers is a popular concrete representation for timestamps [7, 13]. In this case, for indexes i, j ranging over such a vector:

- $\mathtt{t}_1 \leq \mathtt{t}_2 \iff \forall i : \mathtt{t}_1[i] \leq \mathtt{t}_2[i]$
- $\mathtt{t}_1 < \mathtt{t}_2 \iff \mathtt{t}_1 \leq \mathtt{t}_2 \wedge \exists i : \mathtt{t}_1[i] \neq \mathtt{t}_2[i]$
- $\forall i : \max(\mathtt{t}_1, \mathtt{t}_2)[i] = \max(\mathtt{t}_1[i], \mathtt{t}_2[i])$
- $\mathtt{t}_1 \parallel \mathtt{t}_2 \iff \exists i, j : \mathtt{t}_1[i] < \mathtt{t}_2[i] \wedge \mathtt{t}_1[j] > \mathtt{t}_2[j]$
- and so on.

A *clock* is an object that returns unique and monotonically-increasing timestamps. Concretely, any participant in the system may have its local clock, noted *now*(), such that only events timestamped $\leq$ *now*() are visible to it; if it attempts an operation with a larger timestamp, the operation will not return until *now*() increases sufficiently. We do not assume a global clock.

3.1.5 Mappings. In the spirit of MVCC, to update a key's mapping creates a new version, without overwriting an existing mapping; a version is write-once. A version is valid for the range of timestamps starting from its commit timestamp, until the next version, if any.

A store records mappings of a key and a timestamp to an effect. Updating a key k under timestamp t creates a new version tagged with t, without overwriting any existing versions.

For example, suppose store σ records effect 27 in version $\mathtt{t}_1$ of key k. This ensures that lookup$(\sigma, k, \mathtt{t}_1)$ will return 27. To increment k by 10 it might record a new version containing effect

"inc_10" under the same key, with timestamp $t_2 > t_1$. If there are no versions between t_1 and t_2, then, lookup(σ, k, t_2) will return 37.

Note that an assignment masks any earlier versions of the same key.

3.2 Data types and concurrent effects

We distinguish two kinds of value data types: classical vs. RDTs.

Classical data types. A classical data type is one whose only kind of effect is assignment, and does not allow assignments with concurrent timestamps. A standard sequential data type, such as integer or struct, is classical; so is a linearisable data type. A classical data type implies strong consistency, defined as totally-ordered timestamps. The merge operator discussed later is not defined for classical data types.

Replicated Data Types (RDTs). Replicated Data Types (RDTs) support more general and concurrent effects, by supporting a commutative, associative and idempotent merge operator [2, 17]. The current value of an RDT key results from applying the effects that follow the most recent assignment, and merging concurrent effects.

One example is a counter equipped with increment and decrement operations; as increment and decrement commute, merging simply composes them in any order. A classical data type augmented with a suitable merge becomes an RDT [10, 16]. A simple example is the last-writer-wins register, that merges two values by choosing the one with the larger timestamp, deterministically.

Assumptions about data type semantics. We make the following assumptions, which are true for both classical data types and RDTs:

- Effects are associative, i.e., $(\delta'' \circ \delta') \circ \delta = \delta'' \circ (\delta' \circ \delta)$, where $\circ$ denotes functional composition.
- Effects to different keys commute, i.e., $k \neq k' \implies \delta'_{k'} \circ \delta_k = \delta_k \circ \delta'_{k'}$.
- Where merge(δ, δ') is defined, its arguments commute, i.e., merge(δ, δ') = merge(δ', δ).

We assume that all versions of a same key have the same data type.

3.3 Transactions

A transaction is a sequence of effects, with the following properties. All the transaction's reads come from a same *snapshot.* A running transaction is *isolated*, i.e., its effects are not visible outside of it. A transaction terminates in an all-or-nothing manner: it either *aborts*, and its effects are discarded, making no changes to the store; or it *commits atomically*, and its effects become visible in the store all at once.

An effect δ is said committed if it belongs to a committed transaction, and uncommitted otherwise. When a transaction commits, it is assigned a *commit timestamp,* noted `ct`. Notation $\delta_k^{\mathtt{ct}}$ refers to effect δ updating key k and committed with timestamp `ct`.

When it starts, a transaction has a *snapshot timestamp* (a.k.a. dependency timestamp) noted `dt`. A transaction's snapshot `dt` has visibility of an effect if and only if the latter's commit timestamp `ct` is less or equal `dt`.

**** GIVE CONCRETE EXAMPLE

3.4 Visibility

Effects are ordered by the *visibility* relation $\delta_1 \prec \delta_2$ (read "δ_1 is visible to δ_2") defined as follows:

(External visibility EXT) $\delta \prec \delta'$ if they belong to different transactions, τ and τ' respectively, where τ' can read from τ; formally, τ has committed, and $\tau.\texttt{ct} < \tau'.\texttt{dt}$.

(Internal visibility INT) $\delta \prec \delta'$ if both execute in the same transaction, and δ' executes before δ'.

▹**(TODO:) In Section ??, we will prove that visibility is also transitive.**◃

Visibility is a partial order.[1] Committed effects that are not ordered are said *concurrent*, noted $\|$. Note that an uncommitted effect is *not* considered concurrent to another transaction, since it is not visible outside of its transaction.

3.5 Fault models

Failures are non-byzantine stop-restart. The system can stop due to a clean power-down or to a crash. Memory supports blocking reads and non-blocking writes. We consider three kinds of memory:

- Volatile memory (the default) models DRAM. Volatile memory is lost on restart.
- Persistent memory models secondary storage. Before a power-down, in-progress writes first terminate. If the system crashes, in-progress writes may take effect or not, non-deterministically.
- Crash-tolerant memory models an append-mode log file. Writes are sequentially ordered. If some write succeeds, all preceding writes will have succeeded. Furthermore, it supports a blocking *flush* operation. If *flush* returns successfully, or is followed by a power-down, all preceding writes will have succeeded. If a crash occurs before *flush* returns, it is guaranteed that some prefix of the preceding writes terminated successfully.

4 FORMAL MODEL OF TRANSACTIONS

The semantics of transactions is generic and independent of store variant. We first specify transactions formally, then study the implementation challenges.

4.1 Composing effects

The current state of a key is the result of applying the effects to the key in visibility order, and merging concurrent ones. We formalise this intuition with the *effect composition* rule in Figure 4. Here, $\circ$ denotes functional composition: $\forall v, (\delta_2 \circ \delta_1)(v) \triangleq \delta_2(\delta_1(v))$. Recall that $\circ$ binds right-to-left.

In detail: The first two lines state that the null effect $\perp$ (corresponding for instance to a non-initialised key) can be ignored. The third line ensures that the same effect is not applied in duplicate. The fourth line states that keys are independent: effects to different keys commute (they can be applied in any order). The next two lines apply effects that are related by visibility in visibility order. The penultimate one ensures that concurrent effects are merged. Any other case would compose a non-committed effect with an effect that is not visible, and would therefore be an error; hence the final line.

The $\sqcup$ operator is associative, commutative and idempotent. It defines a least upper bound between effects, in a join-semilattice whose least element is $\perp$.

[1] Note that even if timestamps form a total order (strong consistency), it is still possible to have concurrent effects. Indeed, a consistency models such as Snapshot Isolation allows the interval between snapshot and commit of committed transactions to overlap.

$$\textsc{Begin\_Txn}\quad \frac{\tau \notin \Pi_\tau(\mathcal{T}_a \cup \mathcal{T}_c \cup \mathcal{T}_r) \qquad \forall t \in \Pi_{ct}(\mathcal{T}_r) : t \not< \mathtt{dt} \qquad \mathcal{T}_r' = \mathcal{T}_r \cup \{(\tau, \mathtt{dt}, \emptyset, \emptyset, +\infty)\}}{(\sigma, \mathcal{T}_a, \mathcal{T}_c, \mathcal{T}_r) \xrightarrow[\tau]{\mathtt{begin(dt)}} (\sigma, \mathcal{T}_a, \mathcal{T}_c, \mathcal{T}_r')}$$

$$\textsc{Init\_Key}\quad \frac{k \notin \mathrm{dom}(\mathcal{R}) \qquad \mathrm{lookup}(\sigma, k, \mathtt{dt}) = \delta \qquad \mathcal{R}' = \mathcal{R}[k \leftarrow \delta] \qquad \mathcal{T}_r = \mathcal{T}_r'' \cup \{(\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}, +\infty)\} \qquad \mathcal{T}_r' = \mathcal{T}_r'' \cup \{(\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}', +\infty)\}}{(\sigma, \mathcal{T}_a, \mathcal{T}_c, \mathcal{T}_r) \underset{\tau}{\rightarrow} (\sigma, \mathcal{T}_a, \mathcal{T}_c, \mathcal{T}_r')}$$

▹(Marc 2023-02-08:) Somewhere, add the constraint that δ above must resolve to a *Value* in the top-level composed store.◃

$$\textsc{Read}\quad \frac{\mathcal{T}_r = \mathcal{T}_r'' \cup \{(\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}, +\infty)\} \qquad k \in \mathrm{dom}(\mathcal{R}) \qquad \mathcal{R}(k) = v}{(\sigma, \mathcal{T}_a, \mathcal{T}_c, \mathcal{T}_r) \xrightarrow[\tau]{\mathrm{read}_\tau(k) \rightarrow \{v\}} (\sigma, \mathcal{T}_a, \mathcal{T}_c, \mathcal{T}_r)}$$

$$\textsc{Update}\quad \frac{k \in \mathcal{E} \qquad \sigma' = \mathrm{doUpdate}(\sigma, \tau, k, \delta) \qquad \mathcal{T}_r = \mathcal{T}_r'' \cup \{(\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}, +\infty)\} \qquad \mathcal{T}_r' = \mathcal{T}_r'' \cup \{(\tau, \mathtt{dt}, \mathcal{E} \cup k, \mathcal{R}[k \leftarrow \delta(\mathcal{R}(k))], +\infty)\}}{(\sigma, \mathcal{T}_a, \mathcal{T}_c, \mathcal{T}_r) \xrightarrow[\tau]{\mathrm{update}(\tau, k, \delta)} (\sigma', \mathcal{T}_a, \mathcal{T}_c, \mathcal{T}_r')}$$

$$\textsc{Abort}\quad \frac{\mathcal{T}_r = \mathcal{T}_r' \cup \{(\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}, +\infty)\} \qquad \mathcal{T}_a' = \mathcal{T}_a \cup \{(\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}, +\infty)\}}{(\sigma, \mathcal{T}_a, \mathcal{T}_c, \mathcal{T}_r) \xrightarrow[\tau]{\mathrm{abort}_\tau} (\sigma, \mathcal{T}_a', \mathcal{T}_c, \mathcal{T}_r')}$$

$$\textsc{Commit}\quad \frac{\mathtt{ct} \notin \Pi_{ct}(\mathcal{T}_c) \qquad \mathtt{dt} \leq \mathtt{ct} \qquad \sigma' = \mathrm{doCommit}(\sigma, \tau, \mathcal{E}, \mathcal{R}, \mathtt{ct}) \qquad \mathcal{T}_r = \mathcal{T}_r' \cup \{(\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}, +\infty)\} \qquad \mathcal{T}_c' = \mathcal{T}_c \cup \{(\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}, \mathtt{ct})\}}{(\sigma, \mathcal{T}_a, \mathcal{T}_c, \mathcal{T}_r) \xrightarrow[\tau]{\mathrm{commit}\tau(\mathtt{ct})} (\sigma', \mathcal{T}_a, \mathcal{T}_c', \mathcal{T}_r')}$$

▹(Marc 2023-02-02:) Updated to allow ct=dt◃

▹(Marc 2023-02-13:) Moved visibility clause to Begin_Txn◃

Fig. 2. Operational Semantics of Transactions

$$\begin{array}{rcl} \mathrm{lookup} & : & \Sigma \times \mathit{Key} \times \mathsf{TS} \rightarrow \mathit{Eff}_\perp \\ \mathrm{doUpdate} & : & \Sigma \times \mathcal{T}_{ID} \times \mathit{Key} \times \mathit{Eff} \rightarrow \Sigma \\ \mathrm{doCommit} & : & \Sigma \times \mathcal{T}_{ID} \times \mathcal{P}(\mathit{Key}) \times \mathit{Rbuf} \times \mathsf{TS} \rightarrow \Sigma \end{array}$$

Fig. 3. Store interface

The value expected of key k, at some timestamp $\mathtt{t}$, results from applying effects to k up to $\mathtt{t}$, in visibility order, while merging concurrent effects, i.e., $\mathrm{safe}(\Theta, k, \mathtt{t}) \triangleq \bigsqcup\{\delta_k^{t'} : \delta_k^{t'} \in \Theta \wedge t' < \mathtt{t}\}$, where Θ is a legal trace of effects, and $\delta_k^{t'}$ is an effect to key k committed with timestamp t'.[2]

Note that an assignment overwrites the previous history of the key, i.e., $\delta_k \prec \delta_k' \wedge \delta_k' \in \mathit{Value} \implies \delta_k \sqcup \delta_k' = \delta_k'$. More generally, if a key's value is known at some point, earlier updates to

[2]A legal trace is one produced by the semantic rules specified in Section 4.2 hereafter.

$$\delta_k \sqcup \delta'_{k'} = \delta'_{k'} \sqcup \delta_k = \begin{cases} \delta_k & \text{if } \delta'_{k'} = \bot \quad \text{(null effect)} \\ \delta'_{k'} & \text{if } \delta_k = \bot \quad \text{(null effect)} \\ \delta_k & \text{if } \delta_k = \delta'_{k'} \quad \text{(idempotence)} \\ \delta'_{k'} \circ \delta_k & \text{if } k \neq k' \quad \text{(keys are independent)} \\ \delta'_{k'} \circ \delta_k & \text{if } \delta_k \prec \delta'_{k'} \quad \text{(apply in } \prec \text{ order)} \\ \delta_k \circ \delta'_{k'} & \text{if } \delta'_{k'} \prec \delta_k \quad \text{(apply in } \prec \text{ order)} \\ \text{merge}(\delta_k, \delta'_{k'}) & \text{if } \delta_k \parallel \delta'_{k'} \quad \text{(merge concurrent)} \\ \text{undefined} & \text{otherwise} \quad \text{(error, see text)} \end{cases}$$

Fig. 4. Effect composition

that key can be ignored (by associativity of $\sqcup$). Say the value of k at timestamp $t_0 < \mathtt{t}$ is known, then: $\mathsf{safe}(\Theta, k, \mathtt{t}) = \mathsf{safe}(\Theta, k, t_0) \sqcup \bigsqcup\{\delta_k^{t'} : \delta_k^{t'} \in \Theta \wedge t_0 < t' < \mathtt{t}\}$. This justifies the common algorithm for implementing lookup: locate the most recently-known value of the key (e.g., a cached value, or a checkpoint), and apply later updates to it in visibility order. If we impose that the dependency timestamp of running transactions be greater than t_0, then the prefix of Θ up to t_O can be replaced by the *checkpoint* of values computed at t_0. We return to checkpointing in Section 9.12.

4.2 Semantics of transactions

Figure 2 presents the semantics of transactions, in a standard formal syntax. Although intimidating, it can be read as pseudocode, as we explain now. In addition, it is fully formal and unambiguous and easily translated to Coq statements.

For reference, we summarise our notations in Table 1. They are explained hereafter.

4.2.1 Informal presentation. Each rule describes an atomic transition: if the conditions above the horizontal line are satisfied, the transition under the line can take place. Unprimed variables represent pre-transition state (pre-state), primed ones post-transition (post-state). A label on the transition arrow under the line represents a client API call.

As an example, consider rule Begin_Txn. Below the horizontal line is API `begin(dt)` to start a new transaction, transitioning from pre-state $(\sigma, \mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}_\mathrm{r})$ on the left of the arrow $\xrightarrow[\tau]{\mathtt{begin(dt)}}$, to post-state $(\sigma, \mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}_\mathrm{r}')$ on the right. In the post-state, only the set of running transactions changes (as indicated by fact that $\mathcal{T}_\mathrm{r}$ is primed); notably, the store σ is not modified.

The part above the horizontal line can be read as a kind of pseudocode. The first clause chooses a transaction identifier τ that was not used by any transaction (it is not in the union of transaction descriptors, $\mathcal{T}_\mathrm{a} \cup \mathcal{T}_\mathrm{c} \cup \mathcal{T}_\mathrm{r}$, projected by Π_τ along the τ dimension). The second clause (with the universal quantifier) contains a precondition (detailed later) which must be true. The next clause creates a transaction descriptor, a 5-tuple (i.e., a `struct`), whose role is described later. Note that the transition is labeled by τ, indicating that multiple transactions can begin independently of one another; however the semantics is that each such transition is atomic.

4.2.2 Parameters. The rules describe a transaction system, which is a tuple $(\sigma, \mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}_\mathrm{r})$ consisting of a store σ, and sets of transaction descriptors $\mathcal{T}_\mathrm{a}$, $\mathcal{T}_\mathrm{c}$, and $\mathcal{T}_\mathrm{r}$, which keep track of aborted, committed and running (ongoing) transactions respectively.

A transaction descriptor is a tuple whose elements are, respectively, the transaction identifier τ, its *dependency timestamp* `dt`, its *dirty set* $\mathcal{E}$, its *read buffer* $\mathcal{R}$, and its *commit timestamp* `ct`. As

stated previously (Section 3.4), the two timestamps define visibility between transactions. Initially, after rule Begin_Txn, both buffers are empty and the commit timestamp is invalid. For each key that is accessed, rule Init_Key initialises the buffers, and rule Update updates them. Computation of the actual commit timestamp is deferred to the Commit rule.

The rules are parameterised by commands lookup, doUpdate and doCommit, specified in Figure 3. These commands will be specialised for each specific store variant, as detailed later: the map-based variant in Section 6, the journal-based variant in Section 7, and composition variant in Section 9.

4.2.3 Transaction begin. The API begin(dt) is enabled by Rule Begin_Txn, to begin a new transaction with dependency timestamp dt. Its semantics was briefly explained above. The rule does not set any constraints on dt; semantically, any value is legal.

The *visibility* clause $\forall t \in \Pi_{\mathtt{ct}}(\mathcal{T}_\mathrm{r}) : t \not< \mathtt{dt}$ states that the transaction may not depend on some running transaction (i.e., not yet terminated). Suppose this clause was not present: then this transaction might attempt to read a value that has not yet been written or will be aborted. We discuss the synchronisation required for enforcing it in Section **??**.

The visibility clause can be re-written as $\min_{\mathcal{T}_\mathrm{r}}(\mathtt{ct}) \not< \mathtt{dt}$ or equivalently $\min_{\mathcal{T}_\mathrm{r}}(\mathtt{ct}) \geqq \mathtt{dt}$, where $\geqq$ is read "greater than, equal to, or concurrent with."

4.2.4 Reads and writes. The rule Init_Key initialises the read buffer for some key k. As it does not have an API label, it can be called arbitrarily. It does not modify the store, but does modify the current transaction's descriptor within the set of running transactions $\mathcal{T}_\mathrm{r}$. The first clause above the horizontal line simply separates the descriptor of the current transaction (identified by τ) from the others. The second clause checks that k is not already in the read buffer, ensuring that this rule is called no more than once per key. The third clause calls lookup (specific to a store variant) for the requested key and the transaction's dependency timestamp. The next clause initialises the read buffer with the return value of lookup. The final clause above the line stores the updated buffers into the descriptor.

The API $\mathtt{read}_\tau(k)$, which returns the content of the read buffer, is enabled by Rule Read. It does not modify the store. The first clause pulls out the transaction descriptor. The third one identifies the return value as the value of k stored in the read buffer. The second clause requires that the requested key k be already mapped in the read buffer; if not (this key has not been used in this transaction before), this can be interpreted as a cache miss, which is resolved by first executing transition Init_Key.

The API $\mathtt{update}(\tau, k, \delta)$ applies effect δ to key k; it is enabled by Rule Update. It updates both the store and the transaction descriptor. The first two clauses are similar to Read, and similarly cause a cache miss if the key has not been used before (avoiding blind writes). It updates the read buffer (to support "read-my-own-writes"), and marks the key as "dirty," i.e., modified by this transaction. It calls the variant-specific command doUpdate, discussed later in the context of each variant.

4.2.5 Transaction termination. A transaction terminates, either by aborting with no effect, or by committing, which applies its effects to the store.

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

API commitτ(ct) takes a commit timestamp argument. It is enabled by rule Commit, which modifies the store, the running set, and the committed set. Above the line, the first clause is as

usual. The client is free to choose any value for `ct` as long as it satisfies the constraints stated in the rule: it is unique (it does not appear in $\mathcal{T}_c$); it is greater or equal to the dependency.

Command doCommit (discussed later, in the context of each 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.

4.2.6 Correctness and performance challenges. The specification is unambiguous and fits in a single page. In Section 5, we translate it into Coq for verification. Furthermore, we just showed how to read it as a kind of pseudocode. Translating it to sequential, imperative code is relatively straightforward; our Java implementation is called *transaction coordinator* and can be found ▹(CODE:) Where?◃.

Different coordinators can execute in parallel, and failures may occur. Examination of the specification in this light reveals a number of correctness challenges:

(1) The store is accessed concurrently by different transaction coordinators, being read in rules Init_Key and written in rules Update and Commit. Unsurprisingly, care is needed to avoid interference, in order to maintain the illusion that transitions are atomic.

(2) The read buffer and dirty set, being private to a coordinator, pose no difficulty; the set $\mathcal{T}_a$ of aborted transactions is not used anywhere and does not need to be implemented. The set of committed transactions $\mathcal{T}_c$ requires a modest amount of synchronisation between concurrent commits.

(3) The visibility clause of Begin_Txn $\forall t \in \Pi_{\mathtt{ct}}(\mathcal{T}_r) : t \not< \mathtt{dt}$ reads from $\mathcal{T}_r$, which is modified by both Begin_Txn and Commit. Furthermore the choice of `ct` may be delayed until Commit. Therefore, enforcing this clause synchronisation between transaction begin and commit.

(4) Commit is the rule that ensures that a transaction is durable. A major challenge is ensuring that its transition appears atomic despite failures.

The visibility clause of Begin_Txn forbids to read from a non-committed transaction.[3] To illustrate why it is necessary, consider the following example: *(i)* Transaction $T1$ has commit timestamp 1; *(ii)* Transaction $T2$ starts with dependency timestamp $2 > 1$; thus $T2$ must read the writes of $T1$; *(iii)* however, $T1$ is slow and its committed effects reach the store only after the first read by $T2$. Clearly, this would be incorrect: $T2$ should not start as long as $T1$ is still running and has not finalised its transition to committed.

The specification also reveals challenges related to performance:

- The store accumulates versions without bounds. An implementation will address this issue by restricting the use of timestamps, and garbage-collecting obsolete versions.[4]
- In practice, the specification can be implemented in many ways, with different performance characteristics. The classical ad-hoc approach introduces complexity and rapidly diverges from the specification. We argue that this can be addressed in a systematic way, by suitable composition of a small number of basic variants.

We address these challenges in the rest of this paper.

[3] Alternatively, we could write a precondition clause on Commit or on Init_Key, with similar effects.

[4] The sets $\mathcal{T}_a$ and $\mathcal{T}_c$ also appear to grow without bounds. However, the former is a convenience and need not be implemented. The latter is used by Begin_Txn and Commit; inspection of their preconditions (visibility and uniquement clauses) shows that it suffices to record the lower and upper bounds of dependency and commit timestamps.

4.3 Traces and histories

Notation $\delta_k^{\mathtt{ct}}$, called a *tagged effect*, refers to δ updating key k at commit timestamp `ct`. We call (legal) *trace* a set of tagged effects (ordered by $\prec$), produced by the rules of Figure 2; a trace is denoted by meta-variable Θ. A *history* is a linearisation of a trace.

4.4 Consistency

Cerone et al. [3] define Transactional Causal Consistency by the following properties: *(i)* (EXT) The value of a key first read by a transaction is that computed by previous transactions; *(ii)* (INT) later reads include the transaction's own updates; *(iii)* (TRANSITIVE VISIBILITY) Visibility is transitive. We claim without proof that the rules of Figure 2 enforce EXT and INT. This should be obvious by inspection.

▹**(TODO:) We prove in Coq that the rules also ensure TRANSITIVE VISIBILITY. Interestingly, if the clause** $\forall \mathtt{t} \in \Pi_{\mathtt{dt}}\,(\mathcal{T}_{\mathtt{c}} \cup \mathcal{T}_{\mathtt{r}})\,:\,\mathtt{ct} \not\prec \mathtt{t}$ **is omitted from** Commit **(as was the case in our first attempt), transitive visibility can be violated.**◃ ▹**(Marc to @Annette+@Carla:) That clause is crucial to correctness. I suggest to try the proof with and without it!** ◃

Furthermore, when restricted to sequential data types and totally-ordered timestamps, the rules satisfy their specification of Parallel Snapshot Isolation (a.k.a. Non-Monotonic Snapshot Isolation) of Cerone et al. [3].

▹**(TODO:) Assuming total order and enforcing no-write-conflict, prove SI.**◃

We conjecture that SI and SSER can be encoded by strengthening the preconditions in Begin_Txn and Commit, as follows. In SI, snapshots follow the prefix property; under Strict Serialisability, visibility is a total order. More detail is out of scope of this paper.

▹**(FUTURE WORK:) Consistency/isolation levels:**◃

- weak/TCC. Geo-replicate across N sites. Timestamp is a vector size N. Partial order abstracts geo-replication. Each entry in a *vector* (clock or timestamp) counts the events in the *sequential* history of a *replica*. Events of a replica are totally ordered w.r.t. the same replica.
- SI. Total order of commits, no concurrent writes to same key. Timestamps totally ordered (scalar).
- SSER. No concurrent reads or writes to same key. Timestamps totally ordered (scalar).

4.5 **** TODO ****

**** For implementation convenience, commit timestamps are locally monotonically increasing, i.e., greater than any commit that the session/client has observed so far. Set to *now*(). Not imposed by the model.

**** To enable GC, we will assume a monotonically-increasing minimal dependence timestamp. Not imposed by the model. Any key-versions underneath it can be replaced by their composition and GC'ed away.

; the history of a store. Unfolds with time. Current time = *now*(). Snapshots/commits visible iff $\leq$ *now*() (non-visible events don't return). Prefix of ongoing transaction visible. Geo-distribution: each site has its own *now*().

5 ▹(TODO:) TODO◃ GENERAL LEMMAS

5.1 General results

All fine-grain interleavings of effects in a history, while maintaining the relative order of transactions, are equivalent:

$$\begin{aligned}
\text{lookup}(\sigma, k, \texttt{dt}) &= \bigsqcup \max_{t<\texttt{dt}} \{\sigma(k,t)\} \\
\text{doUpdate}(\sigma, _, _, _) &= \sigma \\
\text{doCommit}(\sigma, _, \mathcal{E}, \mathcal{R}, \texttt{ct})(k) &= \begin{cases} \sigma(k) \cup \{(\mathcal{R}(k), \texttt{ct})\} & \text{if } k \in \mathcal{E} \\ \sigma(k) & \text{otherwise} \end{cases}
\end{aligned}$$

where $\max_{t<\texttt{dt}}$ returns the element with the highest timestamp less than dt.

▹(Marc 2023-01-15:) Premature optimisation? Let lookup return the composition of all values, and optimise it to max.◃

Fig. 5. Operations of map store

$$\begin{aligned}
\text{lookup}(\sigma, k, \texttt{dt}) &= \bigsqcup (\text{committed_before}(\sigma, k, \texttt{dt})) \\
\text{doUpdate}(\sigma, \tau, k, \delta) &= \sigma \rhd (\text{update}, \tau, k, \delta) \\
\text{doCommit}(\sigma, \tau, _, _, \texttt{ct}) &= \sigma \rhd (\text{commit}, \tau, \texttt{ct})
\end{aligned}$$

with the following notation:

- σ is a journal-based store.
- $\rhd$ represents concatenation.
- 'update' and 'commit' are tags to distinguish the type of a journal record.
- committed_before$(\sigma, k, \texttt{dt})$ denotes the subsequence of journal σ, of records tagged with key k, that have a commit timestamp less or equal to dt; formally:

$$\text{committed\_before}(\sigma, k, \texttt{dt}) \triangleq \{(\delta, \texttt{t}) \mid \sigma = \sigma_0 \rhd (\text{update}, \tau, k, \delta) \rhd \sigma_1 \rhd (\text{commit}, \tau, \texttt{t}) \rhd \sigma_2 \text{ and } \texttt{t} < \texttt{dt}\}$$

Fig. 6. Operations of Journal store

Lemma 5.1 (Reducibility to sequential trace). *Concurrent trace reducible to equivalent sequential trace, under MVCC/GSI model.*

Other lemmas:

- For a given k and t, lookup(k, t) is unique. The store is a function. (Note: reads block until $t \leq now()()$.)
- Any two variants of the generic store are equivalent (same reads, same writes).
- It follows that *composition*, as defined in Section 9, is OK.

Invariants:

- Commit timestamps assigned by a site are monotonically increasing. Anything stronger, e.g., the LUB of site timestamps is monotonically increasing?
- Any other useful invariants?

**** ▹(TODO:) **** Lemma (prove it!): For all σ, σ' built from the same trace Θ, $\sigma(k, \texttt{t}) = \sigma'(k, \texttt{t})$. ◃

**** ▹(TODO:) **** A more recent value hides any previous effects, i.e., $\bigsqcup \max_{t<\texttt{dt}} \{\sigma(k,t)\} \in \mathit{Value} = \bigsqcup \{ \sigma(k, \texttt{t}) \mid \texttt{t} < \texttt{dt} \}$ ◃

**** ▹(TODO:) **** Therefore: Effect summaries: cut off the history at some lower bound. Saves space, against the promise that all current or future transaction's dt is at least the lower bound. ◃

6 MAP-BASED STORE

6.1 Map store semantics

Our first variant is the *map-based store*, which stores assignments. When a transaction commits, doCommit eagerly assigns new versions of the keys marked dirty. When reading, lookup returns the most recent version in the store, since it replaces any older one.

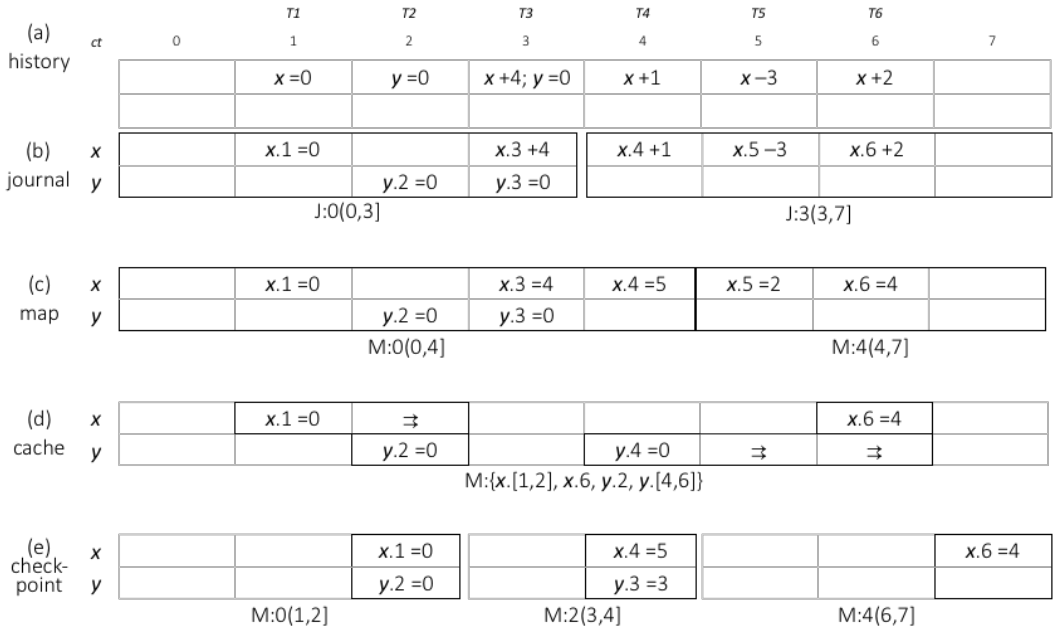

Fig. 7. Store variants and store composition. Subfigure (a) represents a history; the following ones different variants and ranges. To simplify the figure, timestamps are assumed integer (scalar), and the history is represented as a sequence of transactions with strictly monotonic dependency and commit timestamp. Notation: x, y: keys; =0: assign key with value 0; +2, −3: increment key's value by 2, decrement by 3; y.4 +1: increment y by 1 with commit timestamp 4. ⇉ extends validity of a cache entry; in J:1(2, 3]: J = journal/M = map, *low_history*=1, domain (*low_lookup* = 2, *high_lookup* = 3].

A map store is illustrated informally in Figure 7(b).[5] A map store can be viewed as a matrix of keys (rows) and timestamps (columns), where a cell contains either a Assignment or $\perp$. Each successive value associated with a key is a new, write-once version, called *key-version*. This map store grows without bound; we consider bounding its size later (Section 9.10).

The cell at index $(k, \mathtt{t})$ is populated iff some transaction commits an update to key k at with timestamp $\mathtt{t}$. Conversely, if key-timestamp pair $(k, \mathtt{t})$ exists in the domain of map store σ, noted $(k, \mathtt{t}) \in \mathsf{dom}(\sigma)$ or equivalently $\sigma(k, \mathtt{t}) \neq \perp$, that means that a transaction with commit timestamp $\mathtt{t}$ assigned k with value $\sigma(k, \mathtt{t})$.

This invariant is enforced by the map store semantics of Figure 5, starting from an initially empty map store, as follows:

- Command $\mathsf{lookup}(\sigma, k, \mathtt{dt})$ (called in the context of rule Init_Key), finds in store σ the most recent version of key k that is visible from the current transaction's dependency timestamp $\mathtt{dt}$. If there are multiple concurrent most-recent versions, it merges them.
- Command doUpdate, invoked by rule Update, is a no-op for this variant.

[5]Ignore for now the dividing line between 4 and 5, and the bound indications underneath.

- In the context of Commit, command doCommit$(\sigma, \tau, \mathcal{E}, \mathcal{R}, \mathtt{ct})$ copies, for each key updated by transaction τ, as indicated by dirty set $\mathcal{E}$, into a new version, labelled by the transaction's commit timestamp ct.[6]

6.2 Map store implementation

The map-based store (MStore) is implemented in Java and use a hash map to store the values. The MStore is a crash-free store, which means that it does not have a recovery mechanism. This means that the MStore assumes that the system will always gracefully shutdown.

The challenge talked about in *section* 4.2.6 are addressed in the MStore implementation in the following way. The lookup in the MStore is simple as it queries the map with the pair k and *adep* and returns a value with a lower *committimestamp*. To serve multiple concurrent transaction the MStore uses MVCC (Multi-Version Concurrency Control) to ensure that reads and writes do not interfere with each other. For each $k$the map stores v with a timestamp that represent its commit time. When a read operation is performed, the MStore selects the most recent version of the key-value pair that is older than the timestamp of the read operation. This ensures that the read operation sees a consistent view of the k. This means the doUpdate is a no-op as any update to the map would be visible but updates the $\mathcal{R}$ and the $\mathcal{E}$ of the transaction object instead. An advantage of doUpdate is that it does not require keeping track of aborted transactions as an abort simply deletes the transaction object. This also means that each client can work on its local copy of an object without concurrency issues. The doCommit takes a commit timestamp and updates the commit timestamp of every v that is present in the Write Set and then add them to the map. Once finished, the transaction is added to the $\mathcal{T}_c$, and the timestamp server is notified that the commit has finished advancing *max_allowed_timestamp*.

6.3 Checkpoints

▹(TODO:) A checkpoint is a specific map store. A checkpoint is equivalent to { the full store restricted to timestamps above the checkpoint's lower bound }.◃ ▹(Marc:) Move into composition section?◃

7 JOURNAL STORE

An alternative variant is the journal-based store. The journal stores computations, and materialises values lazily on lookup (this is in contrast to the map-based store, which eagerly records Assignments in the store).[7]

Figure 7(b) provides an informal illustration.[8]

Figure 6 gives the formal semantics of a journal store. Let us start with doUpdate, called in the context of the Rule Update from Figure 2. It simply appends an update record to the journal, containing arguments transaction identifier τ, key k, and effect δ. Similarly, doCommit, called in the context of Rule Commit, appends a commit record containing the transaction identifier and commit timestamp. The effect and read buffers are discarded.

The real action is in lookup. To find the value of k, it extracts from the log the effects of all transactions that the current transaction depends upon, i.e., those whose commit-timestamp is before dt, and composes these effects (in visibility order, merging concurrent effects, per Figure 4).

▹(TODO:) Stop backwards search at Assignment.◃

[6] Strictly speaking, copying only the dirty keys is not logically necessary. However, this is justified in practice, since the space of keys is essentially unbounded (e.g., if keys are arbitrary strings).
[7] Technically, this is a redo log. An undo log would store inverse effects.
[8] Again, ignore for now the dividing line between 3 and 4, and the bounds indicated underneath.

Note that, just like the map-based store, the memory footprint of the journal-based store is unbounded. Reading and writing the journal requires care, as it is a shared resource among concurrent transactions. As a journal is sequential, flushing the single commit record to secondary storage ensures crash-atomicity. Indeed, in the event of a crash, either the commit record and all preceding records were written, making the whole transaction durable, or it has not, and the transaction is considered aborted on recovery. Thanks to sequential writes, a persistent journal generally has better throughput than a persistent map store. Our experiments in Section 10 confirm these well-known differences.

7.1 ▹(TODO:) TODO◃ Journal store proofs

In the Coq specification we proved that there is a bisimulation between the journal and the store. The following theorems were defined and proved:

```
Theorem journal_to_store_trace :
  forall tr j trx,
  journal_trace (empty_journal, ([], [], None)) tr (j, trx) ->
  exists s, store_trace (empty_store, ([], [], None)) tr (s, trx).

Theorem store_to_journal_trace :
  forall tr s trx,
  store_trace (empty_store, ([], [], None)) tr (s, trx) ->
  exists j, journal_trace (empty_journal, ([], [], None)) tr (j, trx).
```

Note that the semantic model does not consider interleaving of transactions, i.e., transactions are sequentially executed.

Proof objective:

- Journal bi-simulates generic store for the same history.
- From Lemma "Concurrent trace reducible to equivalent sequential trace," it follows that The absolute ordering in a journal stream is irrelevant, as long as it respects visibility: *(i)* the relative ordering within a transaction, and *(ii)* the timestamp ordering of commit records of different transactions.

7.2 Journal store implementation

The journal-based store (JStore) is implemented in java using the same API as the MStore. JStore is implemented as a lazy store, i.e., it does not materialize values until they are requested. It is built using an append-only structure in Java where all the transaction operations are recorded in the form of records. There are three main types of records in this implementation:

- Begin record, system record that marks the beginning of a transaction. It contains a $\mathcal{T}_{ID}$and a dependency timestamp.
- Effect records, contains updates performed inside the transaction and the transaction identifier it belongs to.
- End record, system record that marks the end of a transaction. It is either an abort record or a commit record. The former only contains the transaction identifier and the latter also contains the `ct`of the transaction.

During the life cycle of a transaction the *begin*, updates and end operations of a transaction are persisted in the journal in a write-ahead manner. This provides crash resilience as it is possible to implement a recovery mechanism that will replay the journal and recover the state of the JStore.

The Coordinator still maintains a $dirtyset\mathcal{E}$ and a $readbuffer\mathcal{R}$ but it is used only for read-your-own-writes.

Because of the lazy nature of the JStore, the lookup is implemented by reading the journal and applying the effects of the transactions that are visible to the current transaction. This is necessary because all the updates are directly written to the JStore through the doUpdate operation. This put the burden of checking the visibility on the lookup which checks if updates have a matching commit record in the journal.

The JStore is implemented using a single thread executor to ensure that the journal is written sequentially. Reading while a concurrent transaction is writing to the journal is not a problem as any futur $\texttt{ct}not < \mathcal{T}_\text{r}.\texttt{ct}$. By storing the all the δ in the journal, the JStore can materialize any v at any time solving any concurrent read problem.

Similarly to the MStore, the JStore ensures that doCommit is visible atomically by notifying the end of the commit to the timestamp server.

On the other hand $\mathcal{T}_\text{a}$ leave a trace in the journal therefore the journal writes an abort record that invalidates the transaction update records.

The JStore being crash resistant the recovery ensures that for every transaction there is a matching begin and end record. If a commit record is missing, the transaction is aborted by writing an abort record. The recovery also initializes the timestamp server with the highest `ct` found in the journal.

Once completed the JStore can resume operations.

8 DESIGN CHALLENGES AND IMPLEMENTATION CHALLENGE

This section describes the general architecture of the system and the challenges we faced during the implementation around the store.

One design choice was to decouple the client handling from the different stores. This was done by implementing a coordinator role that handles the client requests and would be responsible for handling the transactions. This give us the flexibility to implement different stores and to easily switch between them. The coordinator maintains a transaction object, and execute the operations on the store when appropriate.

Another design choice was to implement a timestamp server that would be responsible for generating monotonically increasing unique timestamps and allow the coordinator to query atomically the current timestamp. The timestamp server was fairly simple in the beginning where it would maintain a single timestamps that it would modify and read atomically.

The *ct* of a transaction is selected at the beginning of the transition from running to committed. The coordinator ask for the timestamp server to generate a new timestamp atomically and assign it to the transaction. *doCommit* of the store is called with the *ct* of the transaction and can take some time to complete.

This meant the following situation could occur: a $T1$ transaction start its commit phase, the timestamp server is queried and the doCommit of the store is called with $T1.ct$. Meanwhile another transaction starts and is assigned a higher timestamp than $T1.ct$ without the $T1$ transaction having committed. This would lead to a violation of the definition of visibility as $T1$ would be visible partially to other transactions. This problem is solved by enforcing: $dt \not> \min_{\mathcal{T}_R}(ct)$

To avoid strong synchronisation we maintain two different atomic integers *max_allowed_dt* and *min_allowed_ct*. Both are monotonically increasing and $max\_allowed\_dt \not> min\_allowed\_ct$.

$$
\begin{aligned}
\mathsf{lookup}(\sigma, k, \mathtt{dt}) &= \begin{cases} \text{as in Figure 5 or 6} & \text{if } (k,\mathtt{t}) \in \mathcal{D}_\sigma \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{doUpdate}(\sigma, \tau, k, \delta) &= \begin{cases} \text{as in Figure 5 or 6} & \text{if } (k,\mathtt{t}) \in \mathcal{D}_\sigma \\ \sigma & \text{otherwise} \end{cases} \\
\mathsf{doCommit}(\sigma, \tau, \mathcal{E}, \mathcal{R}, \mathtt{ct}) &= \begin{cases} \text{as in Figure 5 or 6} & \text{if } (k,\mathtt{t}) \in \mathcal{D}_\sigma \\ \sigma & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 8. Store operations, restricted to domain

$$
\begin{aligned}
\mathsf{lookup}(\{\sigma_1,\sigma_2\}, k, \mathtt{dt}) &= \mathsf{lookup}(\sigma_1, k, \mathtt{dt}) \sqcup \mathsf{lookup}(\sigma_2, k, \mathtt{dt}) \\
\mathsf{doUpdate}(\{\sigma_1,\sigma_2\}, \tau, k, \delta) &= \mathsf{doUpdate}(\sigma_1, \tau, k, \delta) \parallel \mathsf{doUpdate}(\sigma_2, \tau, k, \delta) \\
\mathsf{doCommit}(\{\sigma_1,\sigma_2\}, \tau, \mathcal{E}, \mathcal{R}, \mathtt{ct}) &= \mathsf{doCommit}(\sigma_1, \tau, \mathcal{E}, \mathcal{R}, \mathtt{ct}) \parallel \mathsf{doCommit}(\sigma_2, \tau, \mathcal{E}, \mathcal{R}, \mathtt{ct})
\end{aligned}
$$

Fig. 9. Operations of composed store

ADD_MINISTORE

$$
\frac{\mathcal{S} = \{\sigma_1, \sigma_2, \dots\} \qquad \mathcal{D}_\mathcal{S} = \mathcal{D}_{\sigma_1} \cup \mathcal{D}_{\sigma_2} \cup \dots \qquad \forall (k,\mathtt{t}) \in \mathcal{D}_{\sigma_0} \cap \mathcal{D}_\mathcal{S} : \mathsf{lookup}(\sigma_0, k, \mathtt{t}) = \mathsf{lookup}(\mathcal{S}, k, \mathtt{t}) \qquad \mathcal{S}' = \mathcal{S} \cup \sigma_0 \qquad \mathcal{D}_{\mathcal{S}'} = \mathcal{D}_\mathcal{S} \cup \mathcal{D}_{\sigma_0}}{(\mathcal{S}, \mathcal{D}) \xrightarrow{\mathsf{addmini}(\mathcal{S}, \mathcal{D}_\mathcal{S}, \sigma_0, \mathcal{D}_{\sigma_0})} (\mathcal{S}', \mathcal{D}_{\mathcal{S}'})}
$$

REMOVE_MINISTORE

$$
\frac{\mathcal{S} = \{\sigma_0, \sigma_1, \sigma_2, \dots\} \qquad \mathcal{D}_\mathcal{S} = \mathcal{D}_{\sigma_0} \cup \mathcal{D}_{\sigma_1} \cup \mathcal{D}_{\sigma_2} \cup \dots \qquad \mathcal{S}' = \mathcal{S} \setminus \sigma_0 \qquad \mathcal{D}_{\mathcal{S}'} = \mathcal{D}_{\sigma_1} \cup \mathcal{D}_{\sigma_2} \cup \dots}{(\mathcal{S}, \mathcal{D}) \xrightarrow{\mathsf{rmvmini}(\mathcal{S}, \mathcal{D}_\mathcal{S}, \sigma_0, \mathcal{D}_{\sigma_0})} (\mathcal{S}', \mathcal{D}_{\mathcal{S}'})}
$$

▹(Marc:) Make $\mathcal{D}$ explicit argument, or part of store?◃

Fig. 10. Operational semantics of store composition. $\mathcal{S}$ is a composition of ministores $\sigma_1, \sigma_2, \dots$, and $\mathcal{D}_\sigma$ is the domain of σ.

The *min_allowed_ct* is updated by the timestamp server when a new timestamp is generated. And the *max_allowed_dt* is updated by the store when a transaction finishes committing. We also ensure that all the transaction are visible in the order of commit timestamp issues by the timestamp server. The timestamp server keeps a list of transaction that have committed and only advances the *max_allowed_dt* in the order of the list.

Our simple stores impelementation present challenges related to performance that are difficult to solve. Both stores grow without bounds and both having every v ever written it makes a garbage collection necessary. One way to this problem is to partially forget the *history* of the store without violation the consistency requirement of the store.

9 COMPOSING STORES

▹(Implementation Goals:)

- **journal + checkpoint + truncate. Maintains the fault tolerance properties.**
- **checkpointing**
- **cache**

◃

▹(TERMINOLOGY:) In this section, "domain" has a slightly different meaning from previous ones. Alternatives: realm, area, field, province?◃ ▹(NOTATION:) It may be more consistent if a segment is closed on its upper bound and open on its lower bound?◃

The basic store variants described so far (map and journal-based) are simple but have performance issues, e.g., they grow unbounded. We argue in this section that the complex design of modern stores can be explained as a *composition* of basic variants.

The composition rules are particularly simple. A composed store is simply a set of stores (called ministores), each restricted to a specific *domain*. The lookup, doUpdate and doCommit operations over the composition extend recursively to its component ministores.

For instance, a cache is an in-memory record of recently-used versions; ignoring the details of the caching policy, the cache is a map ministore, whose domain is an arbitrary subset of key-timestamp pairs. For instance, Figure 7(d) represents a cache containing versions $\{(x,1),(x,6),(y,2),(y,4)\}$, where $(x,1)$ and $(y,4)$ are known to remain valid until timestamps 2 and 6 respectively. A cache is fast, but must be composed with a (possibly slower) persistent store covering the full range of keys and timestamps; for instance either the journal in Figure 7(b) or the map in (c).

Another example is a write-ahead log (WAL). A WAL combines the high sequential throughput and fault-tolerance of a journal ministore with a fast-response random-access map ministore. When an update commits, it is written (in a crash-tolerant manner) to the journal; later, it is copied and persisted to the map. Thus, the journal's time domain includes the most recent updates, whereas the map's is slightly delayed. For instance, the first map in Figure 7(c) might be combined with the second journal in Subfigure (b). Furthermore, this WAL might be paired with the in-memory cache of Subfigure (d) to decrease average response time.

The LSM-Tree approach [14] decomposes a map into a series of layers. The top layer contains the most recent updates, which percolate downwards towards the bottom layer as they age. When reading, the system queries the layers from top to bottom in succession, returning the first value found. We will characterise this approach as a composition of map ministores that differ in the time domain, most recent at the top.

In future work, we plan to describe a geo-distributed store, such as Antidote [1], as a composition of mini-stores with concurrent time domains.

9.1 Domain of a store

We associate store σ with its *domain*, noted $\mathcal{D}_\sigma \subseteq Key \times \mathsf{TS}$, and modify the store operations lookup, doUpdate and doCommit to be significant only within the associated domain; outside, they are no-ops. This is formalised in Figure 8.

We will call *top-level* store one whose domain is the full universe $Key \times \mathsf{TS}$, and *ministore* one whose domain is a strict subset. A ministore may restrict the key domain, the time domain, or both. For instance, *sharding* composes ministores covering disjoint subsets of the key domain.

Caches aside, the rest of this paper focuses on stores whose time domain is a continuous *segment* of time $(low\_lookup, high\_lookup]$. We also associate each such ministore timestamp *low_history* that represents the beginning of the history used to compute this segment. We summarise this

information with notation X:*low_history*(*low_lookup*, *high_lookup*], where X is M (for a map) or J (for a journal).

For instance (assuming for simplicity that timestamps are integers), a ministore σ=J:10(20, 30] is a journal, representing the history between times 10 (exclusive) and 30 (inclusive), but for which lookup($\sigma, k, \mathtt{t}$) is significant only if $20 < \mathtt{t} \leq 30$. Figure 7(b) represents two journal ministores J:0(0, 3] and J:3(3, 7]; the second contains only the incremental effects in range (3, 7], ignoring those at or before its *low_history* = 3. Subfigure (c) represents two map ministores; the second one reflects only the incremental versions created starting at timestamp 5. Subfigure (e) shows three incremental checkpoints, recording only the last value in the domain, and only if updated between *low_history* and *high_lookup*.

Obviously, *low_history* ≤ *low_lookup* < *high_lookup*. Typically, either *low_history* = *low_lookup* (it records all versions in range) or *low_lookup* + 1 = *high_lookup* (a checkpoint summarising the updates between *low_history* and *high_lookup*).

▹(Marc:) Needs improvement.◃

9.2 Composition of stores

A composed store is simply a set of ministores. Its domain is the union of components' domains. An operation on the composed store just calls itself recursively on the component ministores.

Formally, consider ministores σ_1 and σ_2, with domains $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively. Their composition $\{\sigma_1, \sigma_2\}$ has domain $\mathcal{D}_1 \cup \mathcal{D}_2$. Its operations are defined by the equations in Figure 9, where $\|$ denotes parallel composition.[9]

We showed earlier that stores are functional and that lookup on any map or journal returns the same result. It follows that, if a key-timestamp pair is in domain of two components, it is equivalent to perform lookup on one, on the other, or on both. Formally, $(k, \mathtt{t}) \in \mathcal{D}_1 \cap \mathcal{D}_2 \implies \mathsf{lookup}(\sigma_1, k, \mathtt{t}) = \mathsf{lookup}(\sigma_2, k, \mathtt{t})$. ▹(TODO:) Check this assertion! It might be incorrect for a journal whose history does not start at 0.◃ Therefore it is often most efficient to perform lookup on the most recent ministore first, and to stop as soon as a lookup returns an Assignment. Similarly, an operation can skip recursing into a component for which the arguments are not in domain.

By abuse of notation, we identify a composed store with a store, and note $\sigma = \{\sigma_1, \sigma_2\}$.

The composition of segment stores, whose domains are adjacent or overlap, behaves like single union segment. For instance, in Figure 7 the composition of mini-journals J:0(0, 3] and J:3(3, 7] behaves like J:0(0, 7]. Combining the second checkpoint M:2(3, 4] with mini-map M:4(4, 7] behaves like M:2(3, 7]. Similarly, mini-map M:0(0, 4] combined with mini-journal J:3(3, 7] behaves like M:0(0, 7]. It is not useful to compose M:0(1, 2] with J:3(3, 7], as this would leave a gap.

Formally, composing M:$low\_history_1$(low_lookup_1, $high\_lookup_1$] with M:$low\_history_2$(low_lookup_2, $high\_lookup_2$] where there are no *gaps* between the two, i.e., $high\_lookup_1 \geq low\_lookup_2 \wedge high\_lookup_1 \geq low\_history_2$, behaves like M:$low\_history_1$($low\_lookup_1$, $high\_lookup_2$]. Similarly for J-segment ministores.

Another interesting case is when their histories are adjacent but not their domains. In this case, the history of the composition is the composition of histories, but the domain is just the higher one. For instance, in Figure 7, composing M:0(0, 4] with checkpoint M:4(6, 7] behaves like M:0(6, 7].

Formally, if $high\_lookup_1 \geq low\_history_2$, then their composition behaves like M:$low\_history_1$(low_lookup_2, $high\_lookup_2$].

[9] We defer the discussion of how to ensure the composition remains atomic to Section **??**.

▹(Marc 2023-02-18 20:16:) **** stopped here◃

9.3 Top-level (composed) store

Extra *no-gap* invariant on top-level: represents the whole history: time domain is an interval starting at 0.

Extra *no-broken-reads* invariant on top-level commit: $low\_lookup < \texttt{dt}$. ▹(Marc:) Does this duplicate the visibility clause of begin-txn?◃

Exhibit new top-level commit rule.

9.4 Changing composition

Maintain lookup is functional.

Operations: addmini, rmvmini. How they affect interval domain.

Changing bounds: add new bound, remove old bound. Maintain the no-gaps invariant: add first.

9.5 Garbage collection

Advance *low_lookup* monotonically. Frees memory. Loses information: snapshots that are in the freed zone would be in error.

Enforce no-broken-reads by either not advancing $low\_lookup < \min_{\mathcal{T}_r}(\texttt{dt})$, or by aborting running transactions that violate.

9.6 **** lost text ****

Therefore a composed store whose joint domain starts from time 0, without holes, is equivalent to the full store. Causal consistency implies that there are no holes!

9.7 Proof goals

▹(Proof goals for composition:)

- Composing cache & either map or journal = equivalent to cacheless.
- Composing journal & checkpoint, where the lower bound of the journal overlaps the upper bound of the checkpoint, is equivalent to the full store.
- Move checkpoint + truncate. Can be done concurrently with transactions.
- Chunked (= incremental) checkpoints is equivalent to non-chunked (monolithic) checkpoint.
- Multiple layers are equivalent to a single layer

◃

9.8 Compositon with the Conductor

In order to handle the compositon we introduce the Conductor. The Conductor sits between the client and the ComposedStore and maintains a topological view of the system.

The Conductor uses the same interface used by the JStore client and the MStore client. This allows a ComposedStore to replace a JStore or an MStore transparently in any system. The Conductor uses the topology of the composition to ensure safety between the different layers.

For every topology the Conductor maintains every read is started on the first layer and then transfered to next. But this mecanism is not suited for updates as they must be written in a layer that has persistency and if the target model requires it crash-resistance.

The Conductor chooses the first level accessible level in the topology that satisfies the memory model as the write point of the system.

On top of the topology the Conductor maintain a domain for each layer. These domains are used to ensure safety throughout the ComposedStore.

9.9 Implementation of a simple composed store (Saalik)

The initial composed store is a composition of two stores, a journal (*JStore*) and a map (*CheckpointStore*). As an implementation decision we use the journal as the level where we advance the *history* and the map as a checkpointing level. The higher bound of the *Jstore* advances as transactions are committed. While we could have used the map as the first level of the store, we decided to use the journal because the journal gives us the ability to perform a recovery in case of a crash.

The implementation challenges described in the Section 4.2.6 are solved by using either the JStore and the MStore.

The first implementation challenge of the ComposedStore is to ensure that checkpointing is done properly. Here is the algorithm used to perform a checkpoint:

(1) Choose the next checkpoint timestamp, t_{chk_next} such as $t_{chk\_next} > t_{min\_store}$ and $t_{chk\_next} < min_allowed\_ct$ and $t_{chk\_next} < \mathcal{T}_{\mathrm{r}}.dt$

(2) Find the commit record of the last committed transaction between t_{min_store} and t_{chk_next}.

(3) Write the CheckpointBegin record with t_{chk_prev} and t_{chk_next}.

(4) Lookup all the versions that are visible at t_{chk_next} and write them to the Checkpoint Store.

(5) Update the higher bound of the *CheckpointStore* to t_{chk_next}.

(6) Write the CheckpointEnd record with the $record_id$ that marks the new start of the *JStore*.

(7) Advance the lower bound of the *JStore* to t_{chk_next}.

While the journal had records for the begin, effect and end of a transaction, we encountered a problem where crashing during a checkpoint would result in a state where correctness was not guaranteed.

To solve the second challend we added two new records to the journal, the CheckpointBegin and CheckpointEnd records. The first record, CheckpointBegin, writes all the information about the next checkpoint in the journal. This includes the old checkpoint timestamp and the new checkpoint timestamp. Once the checkpoint is completed, the CheckpointEnd record is written to the journal, with the *record_id* of the first begin record of a transaction that has not been checkpointed.

The two new records are not sufficient to ensure that the checkpointing process is safe in case of a crash. The higher bound of the *CheckpointStore* can be corrupted while updating so we use a two phase update. We use two higher bounds, an active and an inactive one. During the first phase we write the new higher bound to the inactive one and then we swap the two.

Recovery in our implementation must be idempotent, meaning that if the recovery process is interrupted, it can be restarted and it will not corrupt the state of the system. So using the the two previous mechanism we describe the recovery process as follows:

(1) Journal recovers as describe in Section 7.2

(2) If there is a CheckpointBegin record without a CheckpointEnd record it means that the checkpointing process was interrupted.

(3) The checkpoint is restarted from the beginning, as writing a Key-Value pair that is already present in the *CheckpointStore* is idempotent.

(4) Once all the key-versions are written to the map, the Conductor check if the inactive and the active t_{store_max} are correct and if not are updated with the values found in CheckpointBegin record.

(5) A CheckpointEnd record is written to the journal.

(6) The Conductor then initialises its local states of the *JStore* and the *CheckpointStore* domain information.

9.10 Bounded Stores (Gustavo, Annette, Carla)

Consider two stores that satisfy the generic specification. The "old generation" has an upper bound, and the "new generation" has a lower bound.

Coverage requirement: the bounds overlap.

Updates go only to the new generation (i.e., no transactions underneath the lower bound).

▹(TODO:) Prove these lemmas◃

Lemma: Bounding does not violate the atomicity of transactions.

Garbage Collection Lemma: given some timestamp t, the set of key-versions committed at $< t$ that are accessible by transactions is monotonically non-decreasing. Any key-version that is not accessible remains non-accessible; consequently, obsolete key-versions in the old generation can be GC'ed.

9.11 Moving the bounds (Gustavo, Annette, Carla)

Restrictions:

- New transactions go to the new generation.
- The bounds are monotonically non-decreasing.
- The old generation's bound grows before growing that of the new generation, ensuring that they continuously overlap. key-versions are moved from the new to the old, so that complementarity continues to hold.

▹(TODO:) Prove these lemmas.◃ Truncation Lemma: Moving the bounds does not violate the atomicity and durability requirements of transactions.

9.12 Checkpointing and truncation (Saalik)

As stated in Section 4.1, a history of updates up to some timestamp can be replaced by the values of keys at that timestamp. Define a *checkpoint* σ_{t_0} at to be a map store containing these values at time t_0: $\sigma_{t_0} \triangleq \{\mathtt{t} \mapsto \mathsf{safe}(\Theta, k, t_0)\}$. Then we can compute any later version of the key from the checkpoint and its later updates: $\mathsf{safe}(\Theta, k, \mathtt{t}) = \mathsf{lookup}(\sigma_{t_0} \sqcup \bigsqcup\{\delta_k^{t'} : \delta_k^{t'} \in \Theta \wedge t_0 < t' < \mathtt{t}\}$.

One version of a composed store is done with a journal containing every new transaction and a Map-Store containing old information that we will call Checkpoint Store.

Every new transaction writes in the journal until it is over. And periodically the journal is checkpointed in the Checkpoint Store. Every time a checkpoint is created we persist a checkpoint record in the Journal. The checkpoint start record contains information about range of records that will be checkpointed. Once the checkpoint is persisted to the checkpoint store an endCheckpoint record is written to disk with the updated lower bound of the journal.

To represent the active (Safe? Important?) portion of the journal, the system maintains two values. A Low-Watermark, that represents the oldest record visible to an ongoing transaction. It is

advanced every time a new checkpoint is written to the store and can be recovered using the last endCheckpoint written in the journal. The second one High-Watermark is the most recent record that can be visible to an on-going transaction. It is advanced every time a transaction commit and can be recovered using the last commit record written in the journal.

To ensure that the Low-Watermark is safely advanced we define some rules for the checkpointing. First, at any given time there is only one checkpoint happening in the journal (this is not mandatory but is easier here). Second in the range of records that are checkpointed there must be no ongoing transaction. If there is, we either wait for the transaction to finish or we abort them forcibly. Finally, once a checkpoint is started new transactions must have a dependency higher than the Low-Watermark.

XXXXXXXXXXXXXXXXXXXXXX

Before moving bounds checkpoint records are written to the Journal. The first record is a beginCheckpoint record that contains the future lower bound of the journal. Ahead of writting the record the system ensures that there are no ongoing transaction between the active and the futur lower bound. If there are, we either wait for the transaction to finish or we abort them forcibly. Then the record is written in the journal. All objects are then persisted to the checkpoint store. After a checkpoint is done, the Checkpoint Store serves the new generation.

Once a checkpoint is written the store can trigger the garbage collection of unattainable objects in the store as well as deleting all the record that precede the lower bound of the journal.

Old generation: checkpoint store. New generation: bounded-size journal.

Moving the bounds in practice: ensure no ongoing transactions between the old and new lower bound; either by waiting for them to terminate, or by terminating them forcibly. Stop transactions from having dependencies inside the bounds of an ongoing checkpointing

Preview of experimental results: GC'ing obsolete key-versions.

10 IMPLEMENTATION AND EXPERIMENTAL RESULTS (SAALIK)

▹**(Goals:)**

- **Compare single variants with composed variant. Show that cache improves performance. Trade-off between write throughput (journal) and read latency (map).**
- **Snapshot is performed in the background.**
- **Can recover from a crash (journal, composed store). Recovery time is linear in the size of the log.**
- **Performance does not degrade over time for versions with snapshots or caches. Steady-state performance same order of magnitude as RocksDB, etc.**

◃

We plan on comparing all the different store between them.

First we establish a baseline with the journal store. We write a number of random transaction, with random writes to the store. Using this we having a baseline throughput of the Journal.

Once this is done we run the same set of transactions to the checkpoint store and and the composed store to show what are the performance benefits and hits of the different features we added.

Finally we run an experiment where we add caching to the composed store and show how caching improves performance.

Objective of experiments:

- Caching improves performance.
- Truncation bounds the size of the journal.
- GC'ing the checkpoint store is effective.

11 RELATED WORK (EVERYONE)

11.1 Compiling specifications to executable code

Costa [5], Hackett et al. [8].

11.2 Verification of complex storage software

Chajed et al. [4], Hance et al. [9], Malecha et al. [12].

11.3 Formal specification of transactions and isolation models

Cerone et al. [3], Crooks et al. [6], Lesani et al. [11]

12 LESSONS LEARNED AND FUTURE WORK (MARC)

XXXX

Explicit modeling of sites.

Sharded store.

LSM Tree.

Update in place.

ACKNOWLEDGMENTS

This research was funded in part by Agence Nationale de la Recherche (ANR) under grant ANR-19-CE25-0007 (https://www.irif.fr/~gio/adecods/index.xhtml). Gustavo: acknowledgment to ARM?

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)*. Nara, Japan, 405–414. https://doi.org/10.1109/ICDCS.2016.98

[2] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Symp. on Principles of Prog. Lang. (POPL)*. San Diego, CA, USA, 271–284. https://doi.org/10.1145/2535838.2535848

[3] 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))*, Luca Aceto and David de Frutos Escrig (Eds.), Vol. 42. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 58–71. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

[4] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Symp. on Op. Sys. Design and Implementation (OSDI)*. Usenix, Carlsbad, CA, USA, 447–463. https://www.usenix.org/conference/osdi22/presentation/chajed

[5] Renato Mascarenhas Costa. 2019. *Compiling distributed system specifications into implementations.* Master's thesis. U. of British Columbia, Vancouver, BC, Canada. https://doi.org/10.14288/1.0378287

[6] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Symp. on Principles of Dist. Comp. (PODC) (PODC '17)*. Assoc. for Computing Machinery, Washington, DC, USA, 73–82. https://doi.org/10.1145/3087801.3087802

[7] C. J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*. University of Queensland, Australia, 55–66.

[8] Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastnikh. 2023. Compiling Distributed System Models with PGo. In *Int. Conf. on Archi. Support for Prog. Lang. and Systems (ASPLOS) (ASPLOS 2023)*. Assoc. for Computing Machinery, Vancouver, BC, Canada, 159–175. https://doi.org/10.1145/3575693.3575695

[9] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. 2023. Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems. In *Symp. on Op. Sys. Design and Implementation (OSDI)*. Usenix, Boston, MA, USA, TBD. https://doi.org/TBD

[10] Paul R. Johnson and Robert H. Thomas. 1976. *The maintenance of duplicate databases.* Internet Request for Comments RFC 677. Information Sciences Institute. http://www.rfc-editor.org/rfc.html

[11] Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. 2022. C4: Verified Transactional Objects. *Proc. ACM Program. Lang.* 6, OOPSLA1 (apr 2022). https://doi.org/10.1145/3527324

[12] 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. https://doi.org/10.1145/1706299.1706329

[13] Friedmann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Int. W. on Parallel and Distributed Algorithms*. Elsevier Science Publishers B.V. (North-Holland), 215–226.

[14] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[15] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2013. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *Symp. on Reliable Dist. Sys. (SRDS)*. IEEE Comp. Society, Braga, Portugal, 163–172. https://doi.org/10.1109/SRDS.2013.25

[16] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Rapport de Recherche 7506. Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France.

[17] 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))*, Xavier Défago, Franck Petit, and V. Villain (Eds.), Vol. 6976. Springer-Verlag, Grenoble, France, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[18] 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

[19] Ilyas Toumlilt, Pierre Sutra, and Marc Shapiro. 2021. Highly-available and consistent group collaboration at the edge with Colony. In *Int. Conf. on Middleware (MIDDLEWARE)*. ACM/IFIP, Québec, Canada (online), 336–351. https://doi.org/10.1145/3464298.3493405