Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Spécialité

Ingénierie / Systèmes Informatiques

École doctorale

Informatique, Télécommunication et Électronique Paris (ED130)

Investigate an approach to distributed storage that ensures consistency semantics tailored to the application, while retaining scalability and availability.

Saalik Hatia

Soutenue publiquement le : *mai 2023*

À Alice et Bob

Remerciements

Merci à *Ilyas Toumlilt*[Tou21] pour ce super template.

J’aimerais également remercier *ma famille* pour leur confiance.

Pour tout le reste, *il y a Mastercard*.

Enfin, "*La café, c’est la vie*".

Abstract

...

Keywords: K1, K2, K3, K4, K5

Résumé

…

Mots-clés: K1, K2, K3, K4, K5

Contents

Introduction 1

main intro part, in general it should contain at least 4 paragraphs each answers one of the following questions.

P1: What's the thesis main problem?

P2: Why the problem is a problem?

P3: what's the solution?

P4: Why the solution is better than the state of the art?

1.1 Overview

An essential component of a database is its backend, in charge of recording the lowest level of data into a *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 350k LOC, Redis is 160k LOC, and the AntidoteDB backend is 17k LOC. Any such complex software has bugs; database backend bugs are critical, possibly violating data integrity or security [**rocksdbbug**].

Using verification tools has 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 article reports an incremental approach to the rigorous and modular development of a backend. We first formalise the semantics of transactions above a versioned key-value store; this high-level specification helps to reason about correctness, both informally and with the Coq proof tool. Thanks to explicit versioning, the state at any point in time is deterministic, and all implementations are behaviourally equivalent. We specify a map-based and a journal-based variant.

Reading the specification as a kind of pseudocode, we implement it verbatim, without optimisation. More specifically, we implement a map-based and a journal-based store, both in memory and on disk.

Using them directly is impractical; features such as caching, write-ahead logging, checkpointing, journal truncation, etc., are required to improve performance. Our simplifying insight is that such features can be described and implemented by *composing* instances of the basic variants. In particular, we show how to compose a write-ahead log and to bound storage footprint. In future work, we believe the same approach can justify sharding, and geo-distribution. The formal rules for correct dynamic composition are particularly simple.

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 modifying the store. The transaction model appeals to 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.

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

Our contributions can be summarised as follows:

- A formal model of a concurrent, transactional backend store, with three variants: map- and journal-based, and compositional, and a model for the correct composition of stores.
- Interpreting the formal model in terms of system design and implementation challenges; applying it to the implementation to volatile and persistent maps, and a crash-tolerant journal.
- Implementation by composition of a write-ahead log, with caching and bounded storage footprint.
- Experimental evaluation, testing for correctness and showing that our rigorous approach does not preclude performance.

1.2 Contributions

The main results of this dissertation are as follows:

- R1
- R2
- R3
- R4

- R5

Our experimental evaluation shows that:...

1.3 Publications

Some of the results presented in this thesis have been published as follows:

- myFirstPublication
- mySecondPublication

During my thesis, I explored other directions and collaborated in several projects that have helped me to get insights on the challenges of ... These efforts have led me to contribute to the following publications and deliverables:

- TechReport
- ANRProjectDelivrable
- EUProjectDelivrable

1.4 Organization

This thesis is divided into three parts. The rest of this document is organized as follows:

- Part I introduces the common background of our work, formulate the problem, presents the existing solutions and discusses the use-case requirements, this part is divided into three chapters:
  - Chapter 2 provides a complete and up-to-date review of the MyDomain.
  - Chapter **??** presents a use-case point of view of the ...
  - Chapter 3 studies and compares the solutions that have been designed in the state of the art of ...
- Part II, in light of what we saw in the existing work, we will here justify some protocol choices used in our approach...

- Part III provides an experimental evaluation demonstrating the benefits of our approach, compared to other solutions from the state of the art.
- the last part IV we summarize our contribution, and present our vision for the future requirements towards more ...

Part I

Background

2 Distrbuted databases

2.1 Consistency models

2.2 Data structures

State of the art Key Value Store

2.3 Consistency issues

3 Related Work

Intro paragraph

3.1 Overview

...

3.1.1 Discussion

...

Part II

Contributions

In this chapter I present my Contributions

4 Formal specification of a database backend

4.1 System model

Before tackling the bulk of the paper, this section states some definitions and assumptions. Table **??** summarises our notations.

4.1.1 Components of a store

A store has a key-value interface, with versions identified by their timestamps. Keys, values, and timestamps are opaque types, denoted by the sets Key, $Value$ and $\mathtt{TS}$, ranged over by meta-variables k, $v$and $\mathtt{t}$ respectively.

Timestamps are partially ordered by $\leq$. Timestamps are *concurrent* if not mutually ordered: $\mathtt{t}_1 \parallel \mathtt{t}_2 \triangleq \mathtt{t}_1 \not\leq \mathtt{t}_2 \wedge \mathtt{t}_2 \not\leq \mathtt{t}_1$.

Two timestamps have a least upper bound $\max$ and a greatest lower bound $\min$. As a shorthand, $\max_{\mathcal{T}}(\mathtt{dt}) \triangleq \max(\Pi_{\mathtt{dt}}(\mathcal{T}))$ is the least-upper-bound of the $\mathtt{dt}$ dimension in the set of tuples $\mathcal{T}$; and similarly for $\mathtt{ct}$. We note $\mathtt{t}_1 \geqq \mathtt{t}_2 \triangleq \mathtt{t}_1 \not< \mathtt{t}_2$, read "greater, equal, or concurrent." The notations $\min$ and $\leqq$ are defined symmetrically.

Classically, the timestamp type can be represented for instance by a scalar integer (for strong-consistency) or a vector of integers, with the classical definitions for $\leq$ or $<$ [**alg:rep:738**; **alg:rep:738bis**]. In this case, we define $\max$ as follows: $\forall i : \max(\mathtt{t}_1, \mathtt{t}_2)[i] = \max(\mathtt{t}_1[i], \mathtt{t}_2[i])$; and similarly for $\min$.

Effects

A transaction updates a key by assigning a new value. An update operation is called an *effect* hereafter, noted δ.[1] As an assignment masks any earlier assignments to the same key, earlier ones can be ignored. We assume that concurrent effects, if any, can

[1] Our theory also supports operation-based CRDTs and more general effects [**syn:rep:sh143**], but for simplicity we keep this out of scope of this paper.

be merged by an appropriate merge operator, which is is commutative, associative and idempotent [**syn:rep:sh143**; **app:rep:1716**]. For instance, the popular last-writer-wins approach merges concurrent assignments by putting a deterministic total order on their timestamps and keeping only the highest one.

Store

A store, denoted $\sigma \in \Sigma$, is a mapping from a key-timestamp pair $(k, \mathtt{t})$ pair to an effect δ. An absent key-timestamp mapping is identified with $\perp$. Store API $\mathsf{lookup}(\sigma, k, \mathtt{t})$ returns the effect that store σ associates with key k at time `t`. For all practical purposes, this can be seen as a value.

Updating a key k under timestamp `t` creates a new version tagged with `t`; a version is write-once. A version is valid for the range of timestamps starting from its timestamp (excluded), until the next version, if any.

For example, suppose store σ records assigns 27 to version 100 of key k (assuming for the sake of example scalar timestamps). Then, $\mathsf{lookup}(\sigma, k, 101)$ returns 27. Furthermore, if there are no other versions between 100 and 110, $\mathsf{lookup}(\sigma, k, 111)$ will also return 27.

4.1.2 Transactions

An example transactional execution is illustrated in Figure 4.4(a).

A transaction is a sequence of effects. Its reads come from a same *snapshot* defined by a the transaction's *snapshot timestamp* (a.k.a. dependency timestamp) noted `dt`. The effects of a running transaction are not visible outside of it (*isolation*). A transaction terminates in all-or-nothing manner: it either *aborts*, and it makes no changes to the store; or it *commits* with a *commit timestamp*. In this case, its effects become visible in the store atomically, labelled with the commit timestamp.

As we define next, a transaction's snapshot `dt` has visibility of an effect if and only if the latter's commit timestamp `ct` is strictly less than `dt`.

4.1.3 Visibility

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

- $\delta \overset{vis}{\rightarrow} \delta'$ if both execute in the same transaction, and δ' executes before δ'.
- $\delta \overset{vis}{\rightarrow} \delta'$ if they belong to different transactions, τ and τ' respectively, where τ' can read from τ; formally, τ has committed, and $\tau.\mathtt{ct} < \tau'.\mathtt{dt}$.

Visibility is a partial order.[2] Committed effects that are not ordered are *concurrent*, noted $\|$.

4.1.4 State of a store

Assume that at some point in time, the effects to key k are $\{\delta_i, \delta_j, \dots\}$, ordered by $\overset{vis}{\rightarrow}$. Then the expected state of a store maps k to $\max_{\overset{vis}{\rightarrow}}\{\delta_i, \delta_j, \dots\}$, defined as the result of applying $\delta_i, \delta_j, \dots$ in visibility order, while mergeing concurrent effects.

4.2 A formal model, from a systems perspective

In this section, we study a formal model of transactions and store operations, focusing on what they say from a systems perspective.

4.2.1 Semantics of transactions

Figure 4.1 presents the semantics of transactions. 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.

[2]Note that even if timestamps form a total order (strong consistency), transactions might still be concurrent, for instance under Snapshot Isolation.

Begin_Txn

$$\frac{\tau \notin \Pi_\tau(\mathcal{T}_{\mathrm{a}} \cup \mathcal{T}_{\mathrm{c}} \cup \mathcal{T}_{\mathrm{r}}) \qquad \forall t \in \Pi_{\mathtt{ct}}(\mathcal{T}_{\mathrm{r}}) : t \not< \mathtt{dt}}{(\sigma, \mathcal{F}_\sigma, \mathcal{T}_{\mathrm{a}}, \mathcal{T}_{\mathrm{c}}, \mathcal{T}_{\mathrm{r}}) \xrightarrow[\tau]{\mathsf{begin}(\mathtt{dt})} (\sigma, \mathcal{F}_\sigma, \mathcal{T}_{\mathrm{a}}, \mathcal{T}_{\mathrm{c}}, \mathcal{T}_{\mathrm{r}}')}$$

Init_Key

$$\frac{\begin{array}{c}\mathcal{T}_{\mathrm{r}} = \mathcal{T}_{\mathrm{r}}'' \cup \{(\tau, \mathtt{dt}, \mathcal{R}, \mathcal{W}, \mathcal{B}, +\infty)\} \qquad k \notin \mathcal{R} \qquad \mathsf{lookup}(\sigma, k, \mathtt{dt}) = \delta \\ \mathcal{R}' = \mathcal{R} \cup \{k\} \qquad \mathcal{B}' = \mathcal{B}[k \leftarrow \delta] \qquad \mathcal{T}_{\mathrm{r}}' = \mathcal{T}_{\mathrm{r}}'' \cup \{(\tau, \mathtt{dt}, \mathcal{R}', \mathcal{W}, \mathcal{B}', +\infty)\}\end{array}}{(\sigma, \mathcal{F}_\sigma, \mathcal{T}_{\mathrm{a}}, \mathcal{T}_{\mathrm{c}}, \mathcal{T}_{\mathrm{r}}) \xrightarrow[\tau]{} (\sigma, \mathcal{F}_\sigma, \mathcal{T}_{\mathrm{a}}, \mathcal{T}_{\mathrm{c}}, \mathcal{T}_{\mathrm{r}}')}$$

Read

$$\frac{\mathcal{T}_{\mathrm{r}} = \mathcal{T}_{\mathrm{r}}'' \cup \{(\tau, \mathtt{dt}, \mathcal{R}, \mathcal{W}, \mathcal{B}, +\infty)\} \qquad k \in \mathcal{R} \qquad \mathcal{B}(k) = v}{(\sigma, \mathcal{F}_\sigma, \mathcal{T}_{\mathrm{a}}, \mathcal{T}_{\mathrm{c}}, \mathcal{T}_{\mathrm{r}}) \xrightarrow[\tau]{\mathsf{read}_\tau(k) \to \{v\}} (\sigma, \mathcal{F}_\sigma, \mathcal{T}_{\mathrm{a}}, \mathcal{T}_{\mathrm{c}}, \mathcal{T}_{\mathrm{r}})}$$

Update

$$\frac{\begin{array}{c}\mathcal{T}_{\mathrm{r}} = \mathcal{T}_{\mathrm{r}}'' \cup \{(\tau, \mathtt{dt}, \mathcal{R}, \mathcal{W}, \mathcal{B}, +\infty)\} \qquad k \in \mathcal{R} \qquad \sigma' = \mathsf{doUpdate}(\sigma, \tau, k, \delta) \\ \mathcal{W}' = \mathcal{W} \cup k \qquad \mathcal{B}' = \mathcal{B}[k \leftarrow \delta(\mathcal{B}(k))] \qquad \mathcal{T}_{\mathrm{r}}' = \mathcal{T}_{\mathrm{r}}'' \cup \{(\tau, \mathtt{dt}, \mathcal{R}, \mathcal{W}', \mathcal{B}', +\infty)\}\end{array}}{(\sigma, \mathcal{F}_\sigma, \mathcal{T}_{\mathrm{a}}, \mathcal{T}_{\mathrm{c}}, \mathcal{T}_{\mathrm{r}}) \xrightarrow[\tau]{\mathsf{update}(\tau, k, \delta)} (\sigma', \mathcal{F}_\sigma, \mathcal{T}_{\mathrm{a}}, \mathcal{T}_{\mathrm{c}}, \mathcal{T}_{\mathrm{r}}')}$$

Abort

$$\frac{\mathcal{T}_{\mathrm{r}} = \mathcal{T}_{\mathrm{r}}' \cup \{(\tau, \mathtt{dt}, \mathcal{R}, \mathcal{W}, \mathcal{B}, +\infty)\} \qquad \mathcal{T}_{\mathrm{a}}' = \mathcal{T}_{\mathrm{a}} \cup \{(\tau, \mathtt{dt}, \mathcal{R}, \mathcal{W}, \mathcal{B}, +\infty)\}}{(\sigma, \mathcal{F}_\sigma, \mathcal{T}_{\mathrm{a}}, \mathcal{T}_{\mathrm{c}}, \mathcal{T}_{\mathrm{r}}) \xrightarrow[\tau]{\mathsf{abort}_\tau} (\sigma, \mathcal{F}_\sigma, \mathcal{T}_{\mathrm{a}}', \mathcal{T}_{\mathrm{c}}, \mathcal{T}_{\mathrm{r}}')}$$

Commit

$$\frac{\begin{array}{c}\mathcal{T}_{\mathrm{r}} = \mathcal{T}_{\mathrm{r}}' \cup \{(\tau, \mathtt{dt}, \mathcal{R}, \mathcal{W}, \mathcal{B}, +\infty)\} \qquad \mathtt{ct} \notin \Pi_{\mathtt{ct}}(\mathcal{T}_{\mathrm{c}}) \\ \mathtt{dt} \leq \mathtt{ct} \qquad \sigma' = \mathsf{doCommit}(\sigma, \tau, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct}) \qquad \mathcal{T}_{\mathrm{c}}' = \mathcal{T}_{\mathrm{c}} \cup \{(\tau, \mathtt{dt}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct})\}\end{array}}{(\sigma, \mathcal{F}_\sigma, \mathcal{T}_{\mathrm{a}}, \mathcal{T}_{\mathrm{c}}, \mathcal{T}_{\mathrm{r}}) \xrightarrow[\tau]{\mathsf{commit}\tau(\mathtt{ct})} (\sigma', \mathcal{F}_\sigma, \mathcal{T}_{\mathrm{a}}, \mathcal{T}_{\mathrm{c}}', \mathcal{T}_{\mathrm{r}}')}$$

Fig. 4.1.: Operational Semantics of Transactions

$$\begin{array}{rcl}
\mathsf{lookup} & : & \Sigma \times \mathit{Key} \times \mathtt{TS} \to \mathcal{E\!f\!f}_\perp \\
\mathsf{doUpdate} & : & \Sigma \times \mathcal{T}_{ID} \times \mathit{Key} \times \mathcal{E\!f\!f} \to \Sigma \\
\mathsf{doCommit} & : & \Sigma \times \mathcal{T}_{ID} \times \mathcal{P}(\mathit{Key}) \times \mathcal{P}(\mathit{Key}) \times \mathit{Sbuf} \times \mathtt{TS} \to \Sigma
\end{array}$$

Fig. 4.2.: Store interface

Informal presentation

The semantics are written as a set of rules. A rule consists of a set of *premises* above a long horizontal line, and a *conclusion* below. A premise is a logical predicate, specifying expected pre-state (unprimed variables) or post-state (primed variables). 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.

To explain the syntax, consider for example rule BEGIN_TXN. The conclusion describes the transition made by API $\mathsf{begin}(\mathtt{dt})$ from pre-state $(\sigma, \mathcal{F}_\sigma, \mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}_\mathrm{r})$ on the left of the arrow $\xrightarrow[\tau]{\mathsf{begin}(\mathtt{dt})}$, to post-state $(\sigma, \mathcal{F}_\sigma, \mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}_\mathrm{r}')$ on the right. The premise is a set of logical conditions; one that uses only non-primed variables is a pre-condition on the prestate; if it contains a primed variable, it is a post-condition that constrains the post-state. Note that in the right-hand side of this conclusion, only $\mathcal{T}_\mathrm{r}$ is primed, indicating that the other elements of the state do not change. Such a transition is atomic, i.e., there are no intermediate states from a semantic perspective; any intermediate states in the implementation must not be observable.

The system is defined as a tuple $(\sigma, \mathcal{F}_\sigma, \mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}_\mathrm{r})$ consisting of a store, its field, and the sets of aborted, committed, and running transactions' descriptors. Recall from Table **??** that a transaction descriptor is a tuple composed of its identifier τ, its *dependency timestamp* $\mathtt{dt}$, its *read set* $\mathcal{R}$, its *dirty set* $\mathcal{W}$, its *state buffer* $\mathcal{B}$, and its *commit timestamp* $\mathtt{ct}$. We will ignore $\mathcal{F}_\sigma$ until Section 4.5.

The rules are parameterised by commands lookup, $\mathsf{doUpdate}$ and $\mathsf{doCommit}$ (Figure 4.2), which are specialised for each specific store variant, as detailed later.

Transaction begin

BEGIN_TXN describes how API $\mathsf{begin}(\mathtt{dt})$ begins a new transaction with dependency timestamp $\mathtt{dt}$. The first premise chooses a fresh transaction identifier τ. The last premise adds a transaction descriptor to the set of running transactions.

The snapshot of the new transaction is timestamped by $\mathtt{dt}$, passed as an argument. Remember that a snapshot includes all transactions that committed with a strictly lesser commit timestamp. The *visibility* premise $\forall t \in \Pi_{\mathtt{ct}}(\mathcal{T}_\mathrm{r}) : t \not< \mathtt{dt}$ states that the new transaction may not depend on some transaction that is not yet terminated. Suppose this premise 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 4.3.3.

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

Reads and writes

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

Rule INIT_KEY 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. The first premise takes the descriptor of the current transaction τ from the set of running transactions $\mathcal{T}_r$. The second one checks that k is not already in the read set, ensuring that the state buffer is initialised once per key. The third reads the appropriate key-version by using lookup (specific to a store variant). The next two premises update the read set, and initialise the state buffer with the return value of lookup. The final premise puts the transaction descriptor, containing the updated read set and state buffer, back into the descriptor set of running transactions.

▷(SPACE:) Skip read, write, abort. init-key is precond of read and write.◁

For space reasons, we omit a textual description of READ and UPDATE, left as an exercise for the reader. Note that both have premise $k \in \mathcal{R}$, requiring a prior buffer miss. READ calls to the variant-specific command doUpdate, later in the context of the relevant variant.

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 $\mathcal{T}_r$ to $\mathcal{T}_a$, 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. The first premise is as usual. Commit timestamp ct must satisfy the constraints stated in the next two conditions: it is unique (it does not appear in $\mathcal{T}_c$); and it is greater or equal to the dependency. 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 marked as committed.

4.2.2 Consistency

formel:db:rep:1856 define Transactional Causal Consistency by the following properties: 1. The first value of a key read by a transaction is the value computed by previous transactions; 2. a later read also includes the transaction's own updates to that key; and, 3. visibility is transitive. It should be clear by inspection that our transaction semantics enforce these conditions.

Stronger conditions, such as parallel snapshot isolation (PSI), snapshot isolation (SI), or serialisability (SER), can be satisfied by adding appropriate concurrency control conditions to BEGIN_TXN and COMMIT. **formel:db:rep:1856** detail the concurrency control conditions for several consistency conditions [**formel:db:rep:1856**]. For instance, to ensure PSI, one should add to COMMIT a precondition that the transaction's write set is disjoint from that of all concurrent committed transactions.

4.3 From specification to implementation

The specification is unambiguous and fits in a single page. In a separate work, we translate it into Coq for verification; we prove in particular that the specification is functional, i.e., any two implementations of it are behaviourally equivalent. In particular, given equivalent histories (ones containing the same updates with the same timestamps, in any legal order), any two stores will return the same result to the same call to lookup.

4.3.1 Implementation issues

Close examination of the spec is sufficient to uncover some interesting facts, and possible implementation issues:

1. Transactions access the store concurrently, in INIT_KEY, UPDATE and COMMIT. However, they never make conflicting access the same key-version concurrently. The corresponding synchronisation can be lightweight.

2. The set of committed transactions $\mathcal{T}_c$ grows without bounds. However, it is used only to ensure the uniqueness of transaction identifiers and commit timestamps; only a summary thereof needs to be maintained (not the full set), as discussed in Section 4.3.3. $\mathcal{T}_a$ is also unbounded, but is only a notational convenience and need not be implemented.

3. COMMIT is the rule that ensures that a transaction is durable. A major challenge is ensuring that its transition appears atomic despite failures. We defer this discussion to later. ▷(TODO:) **Where is this described?**◁

4. The store accumulates versions without bounds. This is addressed in Section 4.5.5.

5. Enforcing the visibility premise of BEGIN_TXN requires synchronisation between concurrent transactions. We discuss this in detail in Section 4.3.3.

There are many possible implementations of the specification, e.g., with or without sharding, with or without replication, using a WAL, using a cache, using multiple storage layers, etc. The literature so far explains these variants informally, which makes them complex and far from the specification. We argue in the rest of this paper that this can be addressed in a systematic way, by suitable composition of a small number of basic variants.

4.3.2 Implementing the transaction coordinator

The specification of Figure 4.1 describes the *transaction coordinator*, an entity that sits between clients and the store. Our coordinator implementation is mostly a direct, unoptimised translation of the specification viewed as pseudocode, into sequential Java code. Every premise in the spec has a corresponding assertion in the Java code. A coordinator handles a single transaction at a time, whose state (buffer and read- and dirty-sets) remains private to the coordinator.

To test behavioural equivalence between stores requires to compare their results deterministically. For this purpose, a coordinator can call into any of our stores, or into multiple stores at the same time, with the same arguments (transaction identifier, dependency timestamp, commit timestamp, etc.).

Coordinators for different transactions run in parallel. They access the store (or the multiple stores) concurrently; we describe the synchronisation to avoid data races below, under the corresponding store variant. They also share the summaries of committed and running transactions, which are protected by locks or atomics. Finally, synchronisation is required to enforce the visibility premise, as we describe next.

4.3.3 Enforcing the visibility premise

The visibility premise of BEGIN_TXN $\forall t \in \Pi_{\mathtt{ct}}(\mathcal{T}_r) : t \not< \mathtt{dt}$ forbids to read from a non-committed transaction.[3] We can re-write it to the simpler form: $\mathtt{dt} \leqq \min_{\mathcal{T}_r}(\mathtt{ct})$.

To illustrate why this premise is necessary, consider the following example: 1. Transaction $T1$ has commit timestamp 1; 2. Transaction $T2$ starts with dependency timestamp $2 > 1$; thus $T2$ must read the writes of $T1$; 3. however, $T1$ is slow and its committed effects reach the store only after the read by $T2$. Clearly, this would be incorrect. To avoid this, $T2$ must not start until $T1$ has finalised its transition to committed. Clearly, this requires synchronisation between concurrent transactions. We implement it as follows.

We implement a centralised timestamp server, which maintains two monotonically non-decreasing atomic counters $max\_allowed\_dt \leqq min\_allowed\_ct$, as described next.

When a transaction starts (rule BEGIN_TXN), it chooses a dependency $\mathtt{dt} \leqq max\_allowed\_dt$. A transaction requests a new commit timestamp $\mathtt{ct} \geqq min\_allowed\_ct$ from the timestamp server, at the latest during the transition from running to committed (see rule COMMIT). After the coordinator's call to doCommit returns, the coordinator notifies the timestamp server. The server stalls it if there are any in-flight commitments with a lower timestamp; once this is cleared (this transaction is the one committing with the lowest timestamp), the server atomically increments $min\_allowed\_ct$ to $\mathtt{ct} + 1$ and returns. Afterwards (asynchronously), it can advance $max\_allowed\_dt$ up to being equal to $min\_allowed\_ct$; this makes this transaction visible to future transactions. This strict ordering ensures that no running transaction depends on a non-committed transaction.

▷(TODO:) Show how this works out when we choose $\mathtt{ct} = \mathtt{dt}$.◁

▷(Saalik:) Journal store semantics have issues that needs to be corrected.◁

[3] As an alternative, it could be replaced by a suitable premise on COMMIT or on INIT_KEY.

$$
\begin{aligned}
\mathsf{lookup}(\sigma, k, \mathtt{dt}) &= \max_{\xrightarrow{vis}}\{\sigma(k,t) : t < \mathtt{dt}\} \\
\mathsf{doUpdate}(\sigma, \_, \_, \_) &= \sigma \\
\mathsf{doCommit}(\sigma, \_, \_, \mathcal{W}, \mathcal{B}, \mathtt{ct})(k) &= \begin{cases} \sigma(k) \cup \{(\mathcal{B}(k), \mathtt{ct})\} & \text{if } k \in \mathcal{W} \\ \sigma(k) & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 4.3.: Operations of map store

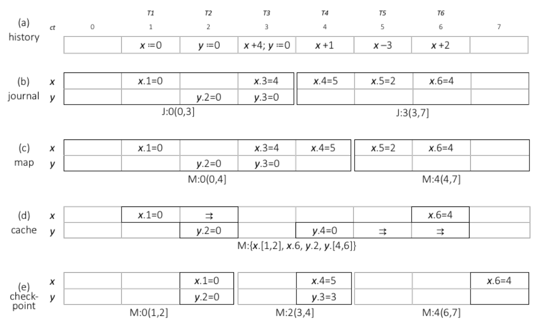

Fig. 4.4.: Store variants and store composition. To simplify the figure, timestamps are assumed integer (scalar), and the history assumed sequential. Notation: x, y: keys; $:\!-\ 0$: assign key with value 0; $+2, -3$: increment key's value by 2, decrement by 3; $y.4\ +1$: increment y by 1 with commit timestamp 4. $\rightrightarrows$ extends validity of a cache entry; in J:1:$(2,3]$: J = journal/M = map, *low_history*=1, domain $(\mathit{low\_lookup} = 2, \mathit{high\_lookup} = 3]$.

4.4 Basic variants

4.4.1 Map store semantics

Our first variant is the random-access *map-based store*, located either in memory or on disk. As illustrated informally in Figure 4.4(b), a map store can be abstracted as an infinite matrix, with a row per key and a column per time unit.[4] An empty store contains all $\perp$. The cell at index $(k, \mathtt{t})$ is populated iff some transaction committed an update to key k at timestamp $\mathtt{t}$. A version remains valid until the next following version in the same k row.

The map-store algorithm is described by the semantics of Figure 4.3. Starting from an initially empty map store:

- 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

[4]In the figure, ignore for now the dividing line between 4 and 5, and the bound indications underneath. To simplify the illustration, it assumes that time ranges over the natural integers. We will consider bounding the store's size later in this paper (Section 4.6).

transaction's dependency timestamp dt. If there are multiple concurrent most-recent versions, it merges them.

- Command doUpdateis a no-op for this variant.
- When a transaction commits with timestamp ct, doCommit eagerly copies its update to every dirty key k, from the state buffer into coordinates $(k, \mathtt{ct})$ of the matrix.

4.4.2 Map store implementation

As above, we implement the map store in Java verbatim. We purposely avoid optimisations that would obscure the intent. We check every premise of the spec with an assertion in the Java code. The in-memory implementation uses a standard concurrent Java hashmap to map keys to an ordered list of timestamped versions. Our persistent map store is the same, with the addition of Java serialisation/deserialisation to/from disk.

As described in Figure 4.3, a lookup in the map store queries the map with arguments k and dt; the map returns the entry for k with the highest timestamp strictly less than dt. The doUpdate function is a no-op. The doCommit function adds a new version to each dirty key, with the transaction's commit timestamp ct; since commit timestamps are unique, different transactions are guaranteed to write to different locations.

This implementation assumes graceful shutdown, purposely avoiding any complex fault-tolerance mechanism. (We discuss fault tolerance later ▷**(TODO:)** ref?◁). More precisely, the in-memory map is assumed lost on restart. Writing to disk is assumed idempotent: writing the same data to disk twice (possibly with a crash in the middle) is indistinguishable from a single write. In case of a crash, an in-progress write may be persisted or not, non-deterministically.

After writing the dirty keys, the doCommit of the on-disk map records the commit timestamp to one of two on-disk locations, alternating between the two; we assume that such a write is atomic. Each location is monotonically increasing.[5] When the store restarts after a power-down, it takes the highest of the two, ensuring that commits are taken into account correctly.

[5] Recall from Section 4.3.3 that the coordinator ensures that transactions commit in growing timestamp order and that commit becomes visible after doCommit returns.

4.4.3 Journal store semantics

Our second variant is the journal-based store. A journal is accessed sequentially, and materialises values lazily on lookup.[6] Figure 4.4(b) provides an informal illustration.[7]

Figure **??** gives the formal semantics of a journal store. Function doUpdate appends an update record to the journal, containing arguments transaction identifier τ, key k, and effect δ. Similarly, doCommit appends a commit record containing the transaction identifier and commit timestamp. The read-set, dirty-set and state buffer 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, ignoring all but the most recent assignment, and merging concurrent effects).

4.4.4 Journal store implementation

We provide an in-memory and an on-disk implementation. We make the following assumptions of the disk. Writes are sequentially ordered; if a write succeeds, all preceding writes have succeeded. There is a blocking *flush* operation; if it returns successfully, or is followed by a power-down, all preceding writes have succeeded. If a crash occurs before *flush* returns, it is guaranteed that some prefix of the preceding writes terminated successfully.

Thus, flushing the single commit record to secondary storage ensures crash-atomicity. 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 aborted on recovery.

Being sequential, a persistent journal generally has better write and worse read response time than a persistent map store.

Our implementation of the journal store is a straightforward, translation of the specification into Java, deliberately avoiding optimisations, and checking premises with assertions. The in-memory implementation is an append-only data structure, containing a sequence of records; the on-disk journal writes its records to a sequential file. There are three main types of records:

[6]Technically, this is a redo log. An undo log would store inverse effects.
[7]Again, ignore for now the dividing line between 3 and 4, and the bounds indicated underneath.

- A begin record marks the beginning of a transaction. It contains its identifier and a dependency timestamp.
- An update record contains an update, tagged by the identifier of the corresponding transaction.
- A record marking the end of a transaction is either abort or commit. Both carry the transaction identifier; a commit also contains its commit timestamp `ct`.

Records of type begin and abort are not strictly required by the specification, but they facilitate parsing the journal and recovery. **▷(Marc:) Add 'doBegin' to Begin_Txn in order to write the begin record.◁**

Disk writes are asynchronous. However, doCommit performs *flush* to ensure that all the records of the transaction are stored persistently in case of a crash. The coordinator calls into the timestamp server to advance $min\_allowed\_ct$ only once doCommit has returned, after flush has succeeded. This ensures that the transaction becomes visible only once its commit has persisted to disk, ensuring correct recovery after a crash.

Function lookup reads the journal sequentially, and applies the effects of the committed transactions that are visible to the current transaction. When it reads an update, it defers applying it until it reads a matching commit record.

The journal store writer is implemented using a single-thread executor, to ensure that the journal is written sequentially. Reading while a concurrent transaction is writing is not a problem, since the timestamp server ensures that commits become visible in increasing timestamp order. As the journal contains all updates, the store can materialise any version required, without any concurrency issues.

The records from multiple transactions may be mixed in the journal (they are not necessarily in sequence). The implementation enforces the invariant that each transaction is structured as a begin record, followed by any number of matching update records, followed either by a matching commit, or a matching abort record. Duplicate commit or abort records are ignored. After a crash, the recovery procedure closes any incomplete transaction with an abort record, and initialises the timestamp server with the highest commit timestamp found in the journal.

4.5 Composing stores

The basic store variants of the previous section are simple but have performance issues, e.g., they grow unbounded. Modern stores improve performance with

$$
\begin{aligned}
\mathsf{lookup}(\sigma, k, \mathtt{dt}) &= \begin{cases} \text{as in Fig. 4.3 or ??} & \text{if } (k, \mathtt{t}) \in \mathcal{D}_\sigma \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{doUpdate}(\sigma, \tau, k, \delta) &= \begin{cases} \text{as in Fig. 4.3 or ??} & \text{if } (k, \mathtt{t}) \in \mathcal{D}_\sigma \\ \sigma & \text{otherwise} \end{cases} \\
\mathsf{doCommit}(\sigma, \tau, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct}) &= \begin{cases} \text{as in Fig. 4.3 or ??} & \text{if } (k, \mathtt{t}) \in \mathcal{D}_\sigma \\ \sigma & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 4.5.: Store σ with domain $\mathcal{D}_\sigma$

$$
\begin{aligned}
&\mathsf{lookup}(\{\sigma_1, \sigma_2\}, k, \mathtt{dt}) = \\
&\qquad \max_{<}\{\mathsf{lookup}(\sigma_1, k, \mathtt{dt}), \mathsf{lookup}(\sigma_2, k, \mathtt{dt})\} \\
&\mathsf{doUpdate}(\{\sigma_1, \sigma_2\}, \tau, k, \delta) = \\
&\qquad \mathsf{doUpdate}(\sigma_1, \tau, k, \delta) \parallel \mathsf{doUpdate}(\sigma_2, \tau, k, \delta) \\
&\mathsf{doCommit}(\{\sigma_1, \sigma_2\}, \tau, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct}) = \\
&\mathsf{doCommit}(\sigma_1, \tau, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct}) \parallel \mathsf{doCommit}(\sigma_2, \tau, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct})
\end{aligned}
$$

Fig. 4.6.: Operations of composed store

ADD_MINISTORE

$$
\frac{\sigma = \{\sigma_1, \sigma_2, \dots\}\mathcal{D}_\sigma = \mathcal{D}_{\sigma_1} \cup \mathcal{D}_{\sigma_2} \cup \dots \forall (k, \mathtt{t}) \in \mathcal{D}_{\sigma_0} \cap \mathcal{D}_\sigma : \mathsf{lookup}(\sigma_0, k, \mathtt{t}) = \mathsf{lookup}(\sigma, k, \mathtt{t})\sigma' = \sigma \cup \sigma_0 \mathcal{D}_{\sigma'} = \mathcal{D}_\sigma \cup \mathcal{I}}{(\sigma, \mathcal{F}_\sigma, \mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}_\mathrm{r}) \xrightarrow{\mathsf{addmini}(\sigma, \mathcal{F}_\sigma, \sigma_0, \mathcal{F}_{\sigma_0})} (\sigma, \mathcal{F}_{\sigma'}, \mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}_\mathrm{r})}
$$

REMOVE_MINISTORE

$$
\frac{\begin{gathered}\sigma = \{\sigma_0, \sigma_1, \sigma_2, \dots\} \\ \mathcal{D}_\sigma = \mathcal{D}_{\sigma_0} \cup \mathcal{D}_{\sigma_1} \cup \mathcal{D}_{\sigma_2} \cup \dots \qquad \sigma' = \sigma \setminus \sigma_0 \qquad \mathcal{D}_{\sigma'} = \mathcal{D}_{\sigma_1} \cup \mathcal{D}_{\sigma_2} \cup \dots\end{gathered}}{(\sigma, \mathcal{F}_\sigma, \mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}_\mathrm{r}) \xrightarrow{\mathsf{rmvmini}(\sigma, \mathcal{F}_\sigma, \sigma_0, \mathcal{F}_{\sigma_0})} (\sigma, \mathcal{F}_{\sigma'}, \mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}_\mathrm{r})}
$$

Fig. 4.7.: Operational semantics of store composition. σ is the composition of ministores σ_1, σ_2, ... $\mathcal{D}_\sigma$ is the domain component of $\mathcal{F}_\sigma$.

features such as caching, write-ahead logging, layered storage, etc. We argue in this section that these features can be represented as *composition* of our 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, as formalised in Figure 4.6.

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 4.4(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 4.4(b) or the map in (c).

Another example is a write-ahead log (WAL). A WAL combines a sequential, fault-tolerant journal ministore, with a 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 4.4(c) might be combined with the second journal in Subfigure (b). Furthermore, this WAL might be composed with the in-memory cache of Subfigure (d) to decrease average response time.

The LSM-Tree approach [**app:1730**] decomposes a store 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 represent this as a composition of map ministores that differ in the time domain.

In future work, we plan to formalise sharding and geo-replication [**rep:pro:sh182**] by composition.

4.5.1 Field and domain of a store

We associate store σ with auxiliary information, its *field* $\mathcal{F}_\sigma$. The field is itself composed of a lower bound $low\_history \in \texttt{TS}$ and a *domain* $\mathcal{D}_\sigma \subseteq Key \times \texttt{TS}$. We 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 4.5.

We will call *total* store one whose field is the full universe $(0, Key \times \texttt{TS})$, 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 $(low\_lookup, high\_lookup]$. Timestamp $low\_history$ is significant only for segment domains, and represents the beginning of the history used in computing 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).

A *checkpoint* is a map whose domain is restricted to a single point: $(high\_lookup - 1, high\_lookup] = \{high\_lookup\}$. A checkpoint contains only the latest version of each key in the range $(low\_history, high\_lookup]$.

For instance (assuming for simplicity that timestamps are integers), a ministore $\sigma{=}J{:}10{:}(20, 30]$ is a journal, representing the history between times 10 (exclusive) and 30 (inclusive), but for which $\mathsf{lookup}(\sigma, k, \texttt{t})$ is significant only if $20 < \texttt{t} \leq 30$. Figure 4.4(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, $low\_history \leq low\_lookup < high\_lookup$. 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 $high\_lookup$).

4.5.2 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 4.6, where $\|$ denotes parallel composition.

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. 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 4.4 the composition of mini-journals $J{:}0{:}(0,3]$ and $J{:}3{:}(3,7]$ behaves like $J{:}0{:}(0,7]$. Similarly, combining the second checkpoint $M{:}2{:}(3,4]$ with mini-map $M{:}4{:}(4,7]$ behaves like $M{:}2{:}(3,7]$. Mini-map $M{:}0{:}(0,4]$ combined with mini-journal $J{:}3{:}(3,7]$ behaves like $M{:}0{:}(0,7]$. In contrast, it is not useful to compose checkpoint $M{:}0{:}(1,2]$ with $J{:}3{:}(3,7]$, as this would leave a gap.

Formally,[8] composing $M{:}LH_1{:}(LL_1, HL_1]$ with $M{:}LH_2{:}(LL_2, HL_2]$ where there are no *gaps* between the two, i.e., $HL_1 \geq LL_2 \wedge HL_1 \geq LH_2$, behaves like $M{:}LH_1{:}(LL_1, HL_2]$. 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 4.4, composing $M{:}0{:}(0,4]$ with checkpoint $M{:}4{:}(6,7]$ behaves like $M{:}0{:}(6,7]$.

Formally, if $HL_1 \ngeq LL_2 \wedge HL_1 \geq LH_2$, then the composition of $M{:}LH_1{:}(LL_1, HL_1]$ with $M{:}LH_2{:}(LL_2, HL_2]$ behaves like $M{:}LH_1{:}(LL_2, HL_2]$.

4.5.3 Modifying a composition

A composition typically does not remain static but gets modified over time. Consider for instance the WAL, where an update is first written to a journal, and later copied to a map. The map's *high_lookup* is always a bit behind the most recent commits, but increases monotonically with time. Conversely, once an update has been copied,

[8]For brevity, for the rest of this section, we will use the abbreviations LH, LL and HL for *low_history*, *low_lookup* and *high_lookup* respectively.

it can be truncated away from the journal, increasing the latter's low_lookup. Thus a WAL is a store composed of a map and a journal, whose domains get modified concurrently with the execution of transactions.

Figure 4.7 specifies the rules for adding or removing a ministore. The composed store is denoted σ, with domain $\mathcal{D}_\sigma$, and the added/removed ministore is noted σ_0 with domain $\mathcal{D}_{\sigma_0}$.

Recall that, for any query $\mathsf{lookup}(\sigma, k, \mathtt{t})$, any store σ that is produced by applying the transaction semantics (Figure 4.1) returns the same result, as long as the arguments are in domain, i.e., $(k, \mathtt{t}) \in \mathcal{D}_\sigma$. We must ensure that this remains true for a store produced by the new rules of Figure 4.7. Hence, in Rule ADD_MINISTORE, the precondition $\forall (k, \mathtt{t}) \in \mathcal{D}_{\sigma_0} \cap \mathcal{D}_\sigma : \mathsf{lookup}(\sigma_0, k, \mathtt{t}) = \mathsf{lookup}(\sigma, k, \mathtt{t})$.

Otherwise, the rules are very simple. ADD_MINISTORE states that adding a ministore to a composition extends its domain, whereas REMOVE_MINISTORE states the reverse.

Note that there is no rule for extending or shrinking ministore's domain, as this can be expressed as a combination of adding and removing. For instance, consider a composed store with a single component, $\sigma = \{\sigma_1\}$. For the sake of argument, say $\sigma_1 = J{:}10{:}(20, 30]$, and we wish to extend the store's upper bound to 45. For this, copy the contents of σ_1 into $\sigma_2 = J{:}10{:}(20, 45]$; add σ_2 to σ; and finally remove σ_1 from σ. Doing it in this order ensures that σ continues to service lookup, doUpdate and doCommit operations correctly and without interruption.

Let us return to the WAL, moving updates from the journal to the map and truncating the journal. For the sake of argument, consider a WAL $\sigma = \{M{:}0{:}(0, 100], J{:}100{:}(100, 140]\}$. This composition has no gaps. To represent growing the map, we add a new map $M'{:}0{:}(0, 120]$, initialised by copying the contents of M, adding the result of $\mathsf{lookup}(\sigma, k, \mathtt{t})$ for all $\mathtt{t} \in (100, 120]$. Once this is done, M is redundant, and we remove it from σ. Alternatively, we could add an incremental segment $M''{:}100{:}(100, 120]$ and not remove M.

Now we can truncate the journal. We copy J into $J'{:}110{:}(110, 200]$, add J' to σ, and remove J. As we were careful to never introduce gaps, this ordering ensures that σ continues to service transactions uninterrupted.

4.5.4 Total store

A total store is a segment store that represents the whole history, i.e., with $\mathit{low\_history} = 0$. To capture this, we add to the rules of Figure 4.7 the following for a total store: • A total store is created as a segment store with $\mathit{low\_history} = 0$. • In REMOVE_MINISTORE, the result domain $\mathcal{D}_{\sigma'}$ must be a segment store that satisfies $\mathit{low\_history} = 0$.

Removing a ministore from the total store should not interfere with the reads of a running transaction. Therefore, in Figure 4.1, we add the following premise to COMMIT for a total store: $\sigma.\mathit{low\_lookup} < \mathtt{dt}$. ▷**(Marc:) We could move this premise to Init_Key; finer grain but possibly harder to prove.**◁

▷**(TODO:) Exhibit new total remove-ministore rule and commit rule.**◁

4.5.5 Garbage collection

As a store accumulates versions, the older versions of a key become obsolete and take up space unnecessarily. Garbage collection refers to the removal of such obsolete versions from a total store. Obviously, a removed version cannot be read by transactions any more. Garbage collection is a direct application of store composition. As an example, consider a store with upper bound 200; say it contains the single segment $M{:}0{:}(0, 200]$. To garbage-collect to timestamp 100, we may replace that single segment with the combination of a checkpoint $M'{:}0{:}(100, 100]$ with incremental map $M''{:}0{:}(100, 200]$.

4.6 Implementing a write-ahead log by composition

To manage composition, we introduce a "conductor" store.[9] In order to compose dynamically, the conductor supports the addmini, rmvmini interface of Figure 4.7. Although the semantics treat all ministores equally, in practice a conductor maintains a topology of ministores and their domains, in order 1. to access the ministores in an efficient order, and 2. to ensure fault tolerance. As any store, the conductor also supports the lookup, doUpdate, doCommit interface, which recurse through the ministores as specified in Figure 4.6, but follows the topology.

[9]Unfortunately mixing musical metaphors.

The preferred topology (ignoring caches) has a journal at the top layer, to leverage its write throughput and fault tolerance properties. Function doCommit returns as soon as the journal's doCommit returns. More details hereafter.

For performance, the conductor recurses into a ministore only if the arguments are in its domain. It directs lookup to the top layer, then to lower ones, returning to the client as soon as a ministore returns non-$\perp$. Function doUpdate applies to all ministores in range, but this generally reduces to the top-layer journal.

4.6.1 Write-ahead log (WAL)

As an illustrative example, consider the case of a write-ahead log (WAL).

A WAL combines a journal J at the top layer, with a checkpoint M. An update goes first to the journal, and is later merged into the checkpoint. As upper bound $M.HL$ of the checkpoint advances, the journal can be truncated, advancing its lower bound $J.LL$. To maintain the gap-freedom invariant $J.LL \leqq M.HL$, the correct order is therefore to first checkpoint, then truncate.

In more detail, the algorithm is as follows. Assume the conductor is managing a total store composed of checkpoint $M{:}0{:}(HL-1, HL]$ and $J{:}LL{:}(LL, HL]$, where $M.HL \geqq J.LL$. It creates a new checkpoint $M'{:}0{:}(HL'-1, HL']$ with $M.HL < M'.HL'$.

The conductor persists a description of the checkpoint in the journal itself. A CheckpointBegin record describes an upcoming checkpoint, and includes both values $M.HL$ and $M'.HL'$. Once M' has been persisted (remember from Section 4.4.4 that this is atomic), the conductor identifies (if any) the first running transaction τ within M' that terminates beyond M', i.e., such that $\tau.\mathtt{dt} \leq M'.HL' < \tau.\mathtt{ct}$. The conductor appends a CheckpointEnd record to the journal, which contains the identifier of the begin record of τ. The conductor adds the new checkpoint M' to the composition, then removes the old one M. Now, the $high\_lookup$ of the checkpoint has safely increased to $M'.HL'$.

However, the conductor must not truncate away the records of any ongoing transaction, i.e., it must stop truncation before the begin record of τ. Thus it can replace J with $J'{:}LL'{:}(LL', HL]$ where $LL' \leqq \tau.\mathtt{dt}$. Following this ordering ensures that truncation remains transparent and crash-tolerant.

The WAL recovers as follows. The journal recovers as in Section 4.4.4. If it contains a CheckpointBegin record without the matching CheckpointEnd (a checkpoint might have not terminated), recovery restarts the checkpoint from the beginning using

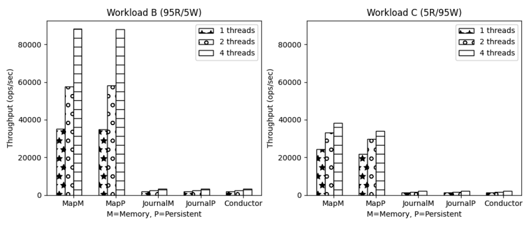

Fig. 4.8.: Average throughput of the different stores

the arguments stored in CheckpointBegin; remember that persisting a map store is idempotent. On success, the conductor appends the missing CheckpointEnd record to the journal, and finally re-initialises the domain of the checkpoint and the journal from the information stored in the journal.

4.7 Implementation and experimental results

Our implementation code is available at `https://anonymous.4open.science/r/ConductorStore-F533`. It consists of just under 3,000 lines of Java code, containing 55 assertions. Our experiments run on a 2021 14” MacBook Pro under MacOS 13.2.1, with 8 cores, 8 hardware threads, and 16 GB of RAM and a 512 Go SSD. Run-time assertion checking is enabled.

4.7.1 Performance comparison

Our performance measurements are intended as an existence proof, and aim only to show decent performance, and to compare the different variants. We run a transactional version of YCSB, with 5 operations per transaction, under three workloads, varying the number of reads and writes. Each workload executes for 60 seconds with 1, 2 or 4 concurrent coordinator threads.

Figure 4.8 plots the throughput. The overall results are not surprising, as our implementation is purposely not optimised. Unexpectedly however, the on-disk map store has higher throughput than the journal. Indeed, as the Update rule forbids blind writes, every write is preceded by a read, which is especially costly since a journal is sequential. Our crash-resistant WAL implementation (marked “Conductor” in the figure) suffers from the same issue. An obvious solution will be to compose the WAL with a cache for absorbing reads, and to allow blind writes.

4.7.2 Correctness

Assertion checking was purposely enabled (disabling it would improve performance). We ran YCSB up to 3 million operations on the map store, and 200,000 operations on the journal, triggering no asserts.

▷**(Annette:) Unclear: "the" coordinator makes it sound like there is no concurrency. Clarify that each store is accessed concurrently.**◁ We also validate experimentally the theoretical result that stores are behaviourally equivalent. To this effect, we enable the coordinator to call out to a number of stores at the same time and to compare their return values. This experiment runs a randomly-generated workload: the coordinator chooses a random key, a random value and a random dependency, and a random commit timestamp in the future; then it calls doUpdate and doCommit on every store. Truncation is disabled for this test, since it would throw away some dependencies. Then it reads the value back with lookup and compares. We run this experiment with an in-memory map, an in-memory journal, and a WAL. After 50,000 transactions, we find no divergence.

Tracking causal relations

5

TBD

6 Implementation of the system

TBD

6.1 Example of code input

```
1   let dc_connection = colony_dc.connect(CONFIG.dbURI, CONFIG.credentials);
2   let cnt = dc_connection.counter("myCounter");
3   dc_connection.update(
4     cnt.increment(3)
5   )
6
7   let peer_connection = colony_pop.connect(CONFIG.signalingServers, CONFIG.
        credentials)
8   let tx = await peer_connection.startTransaction()
9   let map = tx.gmap("myMap");
10  tx.update([
11    map.register("a").assign(42),
12    map.set("e").addAll([1, 2, 3, 4])
13  ])
14  tx.commit().then(
15    console.log(
16      await peer_connection.gmap("myMap").set("e").read()
17    )
18  )
```

Fig. 6.1.: An example of THESE2OUF program.

The TypeScript example in Figure 6.1 illustrates the API. This application opens a session (Line 1). Then, it creates and increments a CRDT counter object (Lines 2–5). Then it connects to a peer group (line 7), and updates the grow-only map (gmap) `"myMap"` in a transaction (lines 8–12). This map contains references to a register object (key `"a"`) and a set object (key `"e"`). The counter update and the commit are both asynchronous (Lines 9 and 13), returning a promise. At line 13, the client waits for the promise, and displays the content of the set.

Part III

Experimental Evaluation

Don’t forget to put the source code link ;-)

Benchmark app and setup

7

Overview ...

Workload ...

8 Performance Evaluation

The performance evaluation will focus on situations where …

8.1 Setup

…

8.2 Metrics A

…

8.3 Metrics B

…

8.4 Metrics C

…

8.5 Metrics 4

…

9 Summary and discussion

...

Part IV

Conclusion

TBD

Bibliography

[Tou21] Ilyas Toumlilt. “Colony: A Hybrid Consistency System for Highly-Available Collaborative Edge Computing”. PhD thesis. Sorbonne Université, 2021 (cit. on p. iii).

List of Figures

List of Tables

List of Listings

Résumé

A

The doctoral schools of Sorbonne Université require at least one page of summary in French. (Even if the website says that you need to provide more than one page, one is enough, which I did).

Résumons donc en français. Tout bon résumé commence par une description générale du problème de la thèse. Le problème étant que les écoles doctorales de Sorbonne Université exigent au moins une page de résumé en français. Voici donc un résumé garanti non traduit sur un DeepL.

Le deuxième paragraphe du résumé doit répondre à la question *pourquoi le(s) problème(s) présenté(s) est un vrai problème ?*.

Ensuite, le troisième paragraphe doit répondre à la question *quelles sont les solutions apportées par la thèse à ces problèmes ?* Cette thèse explore ces problèmes en profondeur, en étudiant l'état de l'art lié … et présente la solution *SystèmeCool2Ouf*, conçu pour répondre aux problématiques exposées. L'une des principales exigences est une approche *DurACuir* qui *EstVraimentVraimentDureCar...* Cependant, cela rend difficile la satisfaction des attentes en matière de *Toute solution à des petits défauts et compromis, on va pas se le cacher*.

Paragraphe 4, *En quoi la solution est meilleure que l'état de l'art ?* Pour répondre à ces défis, nous avons fait le choix d'adopter une approche … en fournissant les plus fortes garanties de …. Un défi connexe est …, que nous avons limité grâce à …

Enfin, traditionnelement on liste une référence aux contribution, Les contributions de cette thèse peuvent être résumées comme suit:

- …;
- …;
- …;

Notre évaluation expérimentale montre que …