Leveraging formal specification to implement a database backend

Saalik Hatia

Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Spécialité

Ingénierie / Systèmes Informatiques

École doctorale

Informatique, Télécommunication et Électronique Paris (ED130)

Leveraging formal specification to implement a database backend

---

Saalik Hatia

Soutenue publiquement le : *01 juin 2023*

Devant un jury composé de :

Emmanuelle Anceaume, Research Scholar, IRISA, CNRS	*Reviewer*
Gaël Thomas, Professor, Telecom SudParis	*Reviewer*
Carla Ferreira, Associate Professor, Nova	*Examiner*
Antoine Miné, Professor, LIP6, Sorbonne Université	*Examiner*
Serdar Tasiran, Principal applied scientist, Amazon	*Examiner*
Patrick Valduriez, Distinguished Research Scholar, Sorbonne Université	*Examiner*
Marc Shapiro, Distinguished Research Scholar, Sorbonne Université, LIP6	*Advisor*

To her

Acknowledgement

"Don’t go to university you’ll get lost," said my high school physics teacher when I told her I applied for Université Pierre et Marie Curie. She was right, I got lost, but I embarked on an amazing journey. Started with an enrolment in Physics and ended up doing a Ph.D. in Distributed Systems. There are a lot of people that made this possible, so I will get right into thanking them.

First I would like to thank the reviewers Emmanuelle Anceaume and Gaël Thomas, for their time reading and reviewing this thesis. I would also like to thank the examiners Carla Ferreira, Antoine Miné, Serdar Tasiran, Patrick Valduriez for taking the time to be part of the jury.

When Marc Shapiro asked me to continue my internship into a Ph.D., I politely declined. Few years later here I am, writing the acknowledgements. This thesis would not exist had you accepted my initial answer. Thank you for trusting me, help me think critically, question myself and better my writing. You also pushed me to present my work even when I thought it was insignificant. During the Ph.D you were always ready to have a meeting, to talk, to entertain an idea which I am grateful for. I hope you enjoyed working with me as much as I did, I will continue to push forward the way you taught me.

A huge thank you to Annette Bieniusa, Carla Ferreira and Gustavo for collaborating with us on this thesis. This work is the result of hours upon hours of discussions, in multiple countries and I hope you had as much pleasure working with me as I had with you.

Thank you to the members of the lab who were my teachers, inspired me to continue in research and then became my colleagues. Thank you Julien Sopena, you made me want to teach and are probably responsible for me being always late to class. Thank you Jonathan LeJeune for being always there and help whenever I needed. Thank you Pierre Sens for being always so nice and having great advice. Thank you to Fabrice Kordon it was a pleasure teaching together and having all those tech discussions. Thank you also to Luciana Arantes, Swan Dubois, Claude Dutheillet, Franck Petit, Yann Thierry-Mieg.

To my fellow past and current Ph.D student and lab colleagues, thank you for all the interesting discussions we had about work, tech, politics, sports and the other absurdly diverse set of hobby everyone has that I learned from. Thank you Aymeric Agon-Rambosson, Maxime Ayrault, Maxime Bittan, Antoine Blin, Marjorie Bournat, Florent Coriat, Cédric Courtaud, Arnaud Favier, Guillaume Fraysse, Yoann Ghigoff, Sara Hamouda, Redha Gouciem, Francis Laniel, Alexandre Lavigne, Gabriel Le Bouder, Ludovic Le Frioux, Etienne Le Louët, Célia Mahamdi, Benoit Martin, Darius Mercadier, Hakan Metin, Sreeja Nair, Ayush Pandey, Baptiste Pires, Laurent Prosperi, Thomas Romera, Lucas Serrano, Ilyas Toumlilt, Vincent Vallade, Dimitrios Vasilas, Vincent Vallade, Daniel Wilhelm. I hope you enjoyed our conversations as much as I did, you are some of the smartest and kindest people I met. I was planning on writing a sentence for each and every one of you but I'm not planning on writing a second manuscript.

To the family I met during my bachelor. We had fun, we became adults together and formed bonds that will never be broken. Thank you Julien Henon, Clémentine Larcena, Michal Rudek, Jonathan Sid-Otmane, Oskar Viljasaar.

To the family I met before university. We've known each other for so long, yet our friendship never faded. Thank you Jeremy Assal, Irwin Assal, Leslie Assal, Niels Assal, Olivier Dopke, Hélène Fernandez, Sina Ghassemi, Dean Huseini, Anne-Laure Kersaudy, Lola Lellouche, Zoe Lellouche, Vinuja Maniccavasagam, Alfred Rodriguez, Savannah Salvoni, Elena Rouche, Hermina Siassia, Nicolas Sidahmed, Laura Zribi.

To Claire-Marine Galletti I am grateful to have you in my life. I know it was not easy to support me throughout this Ph.D and I would not have made it without you (and Tataki).

Finally, thank you to my parents for raising me into who I am, for giving me all the opportunities in life. Thank you to my brother and sister for supporting me and for being the best siblings I could ask for.

I thought I would be the one writing funny acknowledgement, but I also failed, so one last time thanks to each and every one of you.

Abstract

Conceptually, a database storage backend is just a map of keys to values. However, to provide performance and reliability, a modern store is a complex, concurrent software system, opening many opportunities for bugs. This thesis reports on our journey from formal specification of a store to its implementation. The specification is terse and unambiguous, and helps reason about correctness. Read as pseudocode, the specification provides a rigorous grounding for implementation. The specification describes a store as a simple transactional shared memory, with two (behaviourally equivalent) variants, map- and journal-based. We implement these two basic variants verbatim in Java. We specify the features of a modern store, such as a write-ahead log with checkpointing and truncation, as a dynamic composition of instances of the two basic variants. Our experimental evaluation of an implementation has acceptable performance, while our rigorous methodology increases confidence in its correctness.

Keywords: Databases, Distributed systems, Consistency, Modular architecture, Formal specification

Résumé

Conceptuellement, un système de stockage de base de données n'est qu'une correspondance entre des clés et des valeurs. Cependant, pour offrir des performances élevées et une fiabilité, une base de donnée moderne est un système complexe et concurrent, rendant le système prône aux erreurs. Cette thèse relate notre parcours, allant de la spécification formelle d'une base de données à son implémentation. La spécification est courte et non ambigüe, et aide à raisonner sur la justesse. La lecture du pseudocode de la spécification fournit une base rigoureuse pour une implémentation. La spécification décrit la couche de stockage comme une mémoire partagée transactionnelle simple, avec deux variantes (au comportement équivalent), basées sur une map et un journal. Nous implémentons ces deux variantes en restant fidèles à notre spécification. Nous spécifions les fonctionnalités d'une base de données moderne, ayant un système de journalisation avec des snapshots et de la troncature, comme une composition des deux variants. Finalement, nous présentons une évaluation expérimentale avec des performances qui sont acceptables pour une implémentation qui est correcte.

Mots-clés: Base de données, Systèmes distribués, Cohérence, Architecture modulaire, Spécification formelle

Contents

1 Introduction

Databases are an integral part of modern computing, serving as the backbone of applications, services, and systems that are used in every aspect of our lives. From local applications running on individual devices to massive data centers powering global-scale services, databases store and manage data, ensuring accessibility, consistency, and reliability.

As the scale of applications and services expands, so does the complexity of managing data. Local applications typically rely on single-node databases, which focus on optimizing data access and storage within the confines of a single device. Moving to large-scale data centers introduces new challenges such as concurrent access, distributed storage, and high availability. These demands necessitate feature-rich database systems capable of balancing performance, scalability, and fault tolerance.

In geo-distributed settings, where data must be replicated across geographically dispersed locations to ensure low-latency access and global availability, the complexity grows even further. This introduces additional challenges related to data consistency, synchronization, and conflict resolution, as well as increased susceptibility to network partitions and other failures.

As these requirements grow more complex, database systems have been adding feature after feature in an ad-hoc manner, resulting in complex, monolithic systems that are difficult to understand, maintain, and extend. In order to address these challenges, we argue in this thesis that it is realistic to formally specify the database and derive the implementation rigorously from the specification. By leveraging formal models and specifications, we can build robust and efficient transactional database backends that address the diverse challenges and requirements of modern distributed systems, from local applications to geo-distributed settings.

1.1 Overview

In the first part of this thesis, we delve into the realm of databases, focusing on various aspects crucial to understanding the complexities of distributed systems. Fol-

lowed by an overview of transactions, guarantees they provide, and the challenges of implementing them in distributed systems. We then provide a background on concurrency control mechanisms and describe various consistency models, specifically Transactional Causal Consistency (TCC). Lastly, we touch upon various data structures employed in state-of-the-art databases.

The second part of this thesis presents the contributions of our research. We begin by introducing a formal specification of a database backend, which includes defining the system model and the semantics of transactions.

In the third part, we explore the implementation of our specifications into multiple variants, employing either a journal-based backend with Write-Ahead Logging (WAL) or a map-based backend. This allows us to analyze the trade-offs and benefits of each approach in detail.

In the fourth part, we discuss the implementation of the Conductor, a mechanism that handles composition of multiple combined backends while enforcing our system specification. This approach allows resulting database to combine features, such as crash recovery, of each backend, while mitigating some of the drawbacks.

Finally, we talk about ongoing efforts and future research topics to continue and extends our work. We hope that this work will pave the way for future research in this area.

1.2 Contributions

The main results of this dissertation are as follows:

- A formal specification of a database backend, with rules for two variants: map- and journal-based. With composition we introduce a safe way of composing independent stores, and a way to bound the storage footprint of a store.
- A verbatim implementation of the specification in Java, with a map-based and a journal-based store, both in memory and on disk.
- A composition of the two stores, with a journal-based store as the primary store and a map-based store as the secondary store. This is the first step in implementing a full-featured database with cache and checkpointing.
- An experimental evaluation of the implementation, showing that our rigorous approach does not preclude performance.

Our experimental evaluation shows that the map-based store is limited by memory size, while the journal-based store is IO-bound. We also show with experiments that each of our variants are equivalent. We show the difference in performance between the different implementations and explain the result based on the specification.

1.3 Publications

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

- Our paper was submitted to Systor and was rejected. We received good feedback. We are currently updating the paper for a submission to EuroSys.
- Compas 2022 - Towards correct high-performance database backends

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

- TechReport - Specification of a Transactionally and Causally-Consistent (TCC) database Hatia and Shapiro [HS20] ANR RainbowFS

Part I

Background

Database design

2

2.1 Database Backends

Database backends are responsible for handling the lowest level of data, from caching to persisting to answering queries. They form the core of database systems, providing efficient storage and retrieval mechanisms while handling various performance and consistency requirements. Most modern databases can be categorized into two main types: SQL and NoSQL. The main user-facing difference between them is the querying model, the former uses Structured Query Language (SQL) and the latter uses mainly keys to access data. Data models are also different, but there is more diversity in NoSQL databases, which can be classified into five main types: Key-Value Store, Document-based, Vector-based, Graph-based, and Column-family based.

2.1.1 Relational Databases (RDBMS)

Relational databases (Codd [Cod70], Date [Dat03], [Sto02], Bernstein and Goodman [BG81], Gray and Reuter [GR92a]), also referred to as SQL databases, typically store data in the form of tables. Tables consist of rows and columns where each row represents a record (data entites) and each column represents an attribute. In order to access and manipulate data in an RDBMS database, users must use the standardized Structured Query Language (SQL) to interface with the database. One advantage of using SQL is the application independence it provides, as the same SQL queries can be used across different database systems and should provide the same results regardless of the underlying database system. This relational model also allows for complex relationships between data entities that enable complex application logic like ACID transactions. ACID stands for Atomicity, Consistency, Isolation, and Durability and it stands for the properties that a transaction must provide to ensure data integrity and consistency. We will talk more about transactions in Section 3.1.8.

While SQL databases are well suited for handling complex data relationships, they have some drawbacks. Data is stored in a fixed schema, so as applications evolve over time the schema must be updated to accommodate changes or new data models.

When a query is executed, the database acquires a lock on the data it is accessing, which allows for isolation but also prevents other queries from accessing the same data. This can lead to performance issues during high loads and also prevents the database from scaling horizontally.

ACID properties are desirable in any database system, but they come at a cost. SQL databases are not well suited for handling unstructured data, as they require a fixed schema. They also do not scale well in distributed environments, as they rely on a centralized architecture and do not support horizontal scalability.

Examples of SQL databases are MariaDB, PostgreSQL, and SQLite.

2.1.2 NoSQL Databases

To avoid the drawbacks of SQL databases, as the internet and we applications became more prevalent NoSQL emerged. Main motivations for NoSQL databases is to provide a more flexible data model that is tailored to the need of an application. One effect of NoSQL databases having no standardized query language is that multiple data model co-exist in the space. The diversity provides a lot of flexibility, as there are databases that are made for a wide ranges of use cases from embedded systems to large scale distributed systems. This diversity leads to five main categories of NoSQL databases: Key-Value Store, Document-based, Column-family, Graph-based, and Vector-based.

Key-Value Store (KVS)

Key-Value Stores (KVS) are the simplest form of NoSQL databases. In a KVS, data is stored as a mapping from a unique key to corresponding value(s). This straightforward data model lends itself to efficient implementations and is suitable for a wide range of use cases. The primary operations supported by a KVS are the retrieval of a value based on its key (get) and the storage or update of a value associated with a key (put). Due to their simplicity and performance characteristics, KVS databases are often employed as components of larger, more complex systems, such as caching layers, SQL backend, other NoSQL database backend or distributed storage systems.

A common implementation of a Key-Value Store relies on a hash table, a data structure that efficiently maps keys to their corresponding values. Hash tables allow for rapid insertion, deletion, and retrieval of key-value pairs, making them a suitable

foundation for KVS databases. In addition to hash tables, other data structures and algorithms, such as B-trees (Bayer and McCreight [BM70]), LSM-trees (O'Neil et al. [O'N+96]), or trie-based structures, can be employed to optimize specific aspects of KVS performance, such as read or write latency, storage overhead, or memory footprint.

It is important to note that the simplicity of the Key-Value Store data model can also be a limitation in certain scenarios. For example, KVS databases do not support complex data structures, relationships between entities, or advanced querying capabilities like those provided by SQL databases or other NoSQL databases, such as document-based, graph-based based databases. Consequently, Key-Value Stores are best suited for use cases where the data model is straightforward, and operations are primarily focused on individual key-value pairs rather than complex relationships or analytical queries.

One exemple of KVS is RocksDB [Roc] an embedded database, based on LevelDB [Lev], designed for fast storage and retrieval of data. It utilizes a Log-Structured Merge Tree (LSM Tree, O'Neil et al. [O'N+96]) to store data, which combines the benefits of both log-based [RO92] and B-tree-based storage systems. We talk in more detail about LSM Trees in Section 2.4.5.

Document Databases

A Document-based NoSQL database store data in the form of documents, typically using formats such as JSON or BSON. Data is stored based on schemas like SQL databases, but they require less structure and are more flexible.

It is designed for storing, retrieving, and managing document-oriented information. They provide a flexible schema, allowing for the storage of complex and nested data structures. Example of Document-based databases are MongoDB, CouchDB, and RavenDB.

Column-family Databases

A Column-family database is a type of NoSQL database where data is organized in tables with rows and columns similar to a relational database. A key difference is that columns are grouped into column families, i.e. groups of related columns, which are the primary unit of access and storage.

It is designed for large-scale, write-heavy workloads and support agregation queries and time-series data. Examples of Column-family databases are Cassandra, HBase, and ScyllaDB.

Graph Database

Applications that require complex data relationships like social networks, recommendation engines, and knowledge graphs can benefit from using Graph-based databases.

A Graph-based database is different from SQL database in that it does not rely on a fixed schemas and offer more flexibility to developers on how data is handled. In addition, they enable efficient querying of connected data, as they are optimized to navigate and traverse relationships between nodes without the need for expensive join operations.

A Graph-based database employs various indexing and storage techniques to optimize graph traversal and querying. Some of the notable techniques include adjacency lists, adjacency matrices, and various graph partitioning methods. Additionally, Graph-based databases often support specialized query languages, such as Cypher (used by Neo4j) and Gremlin (used by Amazon Neptune and others), which allow users to express complex graph traversals and queries in a concise and intuitive manner.

In recent years, there has been a growing interest in combining graph databases with other database types to create hybrid systems. For example, some databases combine the flexibility of graph databases with the high-performance querying capabilities of column-family databases or the powerful search functionality of document-based databases. This approach allows users to take advantage of the strengths of multiple database types while minimizing their drawbacks.

Examples of Graph-based databases include Neo4j, Amazon Neptune, and OrientDB.

Vector Databases

High-dimensional vector data presents unique challenges for database systems. Traditional indexing methods, such as B-trees and hash indexes, are inefficient for high-dimensional data due to the "curse of dimensionality", which makes searching and indexing increasingly difficult as the number of dimensions increases (Beyer et al.

[Bey+99]). Historically, most state-of-the-art nearest neighbor search functionality was provided to existing databases through libraries like Annoy [Ber], Faiss [Fai], and NMSLIB [Nms]. Vector databases, also known as vector-based databases or vector similarity search engines, are specialized databases designed to handle high-dimensional vector data. These databases are useful for applications involving machine learning, image and text processing, and similarity search, where data is represented as high-dimensional vectors in a vector space.

One of the most popular techniques for indexing high-dimensional data is the Locality-Sensitive Hashing (LSH) [GIM99]. LSH is an approximate nearest neighbor search algorithm that reduces the dimensionality of the data by hashing it into a lower-dimensional space. The key idea behind LSH is that similar vectors are more likely to hash to the same bucket, thereby allowing efficient search for similar items.

Another technique used in vector databases is the use of tree-based data structures, such as k-d trees (Bentley [Ben75]), ball trees (Omohundro [Omo89]), and vantage-point trees (Yianilos [Yia93]). These data structures partition the vector space into regions based on distance metrics, which can then be used to efficiently search for nearest neighbours.

A more recent approach to indexing high-dimensional data involves using graph-based data structures, such as Hierarchical Navigable Small World (HNSW) graphs (Malkov and Yashunin [MY18]). HNSW graphs exploit the small-world properties of high-dimensional vector spaces, where most nodes can be reached from any other node through a relatively small number of hops. This enables fast search and retrieval of similar items.

In conclusion, NoSQL databases are a broad category of database systems with a wide range of data models and storage solutions. By understanding the intricacies of each type of NoSQL database, developers can choose the right database for their application and take advantage of the strengths of each database type. In the next section, we will discuss ACID properties, which are crucial for maintaining data integrity and consistency in database systems.

2.2 Transactions

A transaction can be defined as an indivisible atomic sequence of operations performed on a database.

2.2.1 ACID Properties

As defined by Haerder and Reuter [HR83], ACID stands for Atomicity, Consistency, Isolation and Durability, these are properties that a transaction must satisfy to be considered ACID-compliant (Hellerstein et al. [HSH07]).

- **Atomicity** is the All-or-nothing guarantee. This means that either all operations in a transaction commit or none of them do.
- **Consistency** This is an application-specific guarantee. Given a set of invariants, a transaction must ensure these invariants are preserved, when run in isolation, after the transaction is committed.
- **Isolation** Transactions are isolated from each other, meaning that the effects of a transaction are not visible to other transactions until the transaction has been committed. Note that there are multiple degrees of isolation known as consistency guarantees.
- **Durability** Once a transaction has been committed, its effects on the database are permanent.

Snapshot is a consistent view of a database, a term introduced by Reed in the context of distributed systems [Ree83]. Snapshots are used in a concurrent environment, to ensure that each transaction operates on a consistent and stable view of the database. Snapshots provide support for the Isolation property of ACID.

Write atomicity is the property that ensures that either all or none of the writes in a transaction are applied to the database. It an essential aspect of the Atomicity property. Implementing write atomicity can be achieved with techniques, such as Write-Ahead Logging (WAL) and Shadow Paging which we compare in Section 2.3.

Read atomicity ensures that a transaction observes a consistent Snapshot of a database while executing its read operations, satisfying the Isolation property of ACID transactions. This property is essential to prevent a transaction from reading partially committed or inconsistent data from concurrent transactions. Implementing read atomicity can be achieved with techniques such as Multi-Version Concurrency Control (MVCC) or Update-in-Place (UIP) which we compare in Section 2.4.3.

2.2.2 Concurrency Control

Concurrency control, as defined by Bernstein and Goodman [BG84], ensures that in a concurrent system to ensure that transactions execute correctly even when multiple transactions are accessing the same data simultaneously. Correctly in this context means that the transactions are executed in a way that preserves the ACID properties. To do so database systems employ concurrency control mechanisms to manage concurrent access to shared data.

There are two categories of techniques for managing these type of concurrency: locking and latching [HSH07].

Locking

Locking [GR92b] is a concurrency control technique to ensure ACID properties of transactions, particularly the Isolation property. Locks are primarily employed to manage concurrent transactions and prevent conflicts that may arise due to simultaneous access to shared data.

A lock is designed to protect an associated ressource from concurrent access. There is different types of locks, such as readers–writer locks and exclusive locks. Readers-writer locks allow multiple transactions to read a ressource concurrently but prevent a concurrent transaction from modifying the associated ressource. An exclusive lock allows only a single transaction at a time to access the associated data item, blocking any concurrent transaction from reading or writing it.

Two-Phase Locking (2PL) Papadimitriou [Pap79] is another lock-based concurrency control mechanism that guarantees serializability (see 3.1)). The 2PL protocol consists of two phases: the growing phase, in which a transaction acquires locks but cannot release them, and the shrinking phase, in which the transaction releases locks but cannot acquire new ones.

However, locking mechanisms can lead to issues such as deadlocks and starvation, affecting the performance and throughput of the database system.

Latching

Latching is a low-level synchronization mechanism used to manage concurrent access to shared data structures in multi-core systems. Latches are similar to locks

in their purpose but are employed at finer granularity, protecting the internal data structures of a database system where a lock protects a user-level data item.

Latches are generally used to ensure the consistency and integrity of shared data structures, such as B-trees, when accessed by multiple threads in a multi-core environment. Latches are more lightweight than locks and are managed by the database system's internal components rather than its transaction management subsystem.

Latches can be implemented using various strategies, such as reader-writer latches, spinlocks, or mutexes.

2.3 Consistency Models

A consistency model defines the rules and guarantees provided by a concurrent or distributed system regarding the ordering of operations and the visibility of data. The different non-transactional consistency models are detailed by Viotti and Vukolić [VV16]. Cerone et al. [CBG15] followed by specifying transactional consistency models. We will present a few of them that are relevant to our work and let the reader refer to the paper for a more in-depth discussion.

Consistency models are primarily concerned with the following aspects:

Visibility: The consistency model specifies when the effects of an operation, such as an update or a write, become visible to other operations, such as reads or queries.

Ordering: The consistency model defines the permissible order of operations and the relative order in which the effects of these operations are observed by other components of the system.

2.3.1 Sequential Consistency

Sequential consistency is defined by Lamport [Lam79] as a consistency model that ensures that the result of any execution is the same as "*... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*".

2.3.2 Linearizability

Linearizability [HW90] is a stricter consistency model than sequential consistency. A Linearizable system is Sequentially consistent and consistent with the real-time ordering of operations. In a Linearizable system, each operation appears to take effect at a single point in time, which is between the invocation and the return of that operation. All linearization points are totally ordered, and this total order respects real time.

If an operation $\mu_[1]$ finishes before another operation $\mu_[2]$ starts, then $\mu_[1]$ appears to take effect before $\mu_[2]$.

2.3.3 Snapshot isolation

Snapshot Isolation (SI) [Ber+95] is an isolation level that guarantees that each transaction observes a consistent view of the database chosen at the beginning of the transaction. This ensures read atomicity, i.e., a transaction observes a consistent snapshot of the database while executing its read operations, preventing it from reading inconsistent data written by concurrent transactions.

This allows multiple transactions to execute concurrently without the need for locking. When a transaction wants to commit, the database will check if there are any conflict in regards to the consistency requirements of the database. If there is no conflict the transaction is committed and made *visible*, meaning all the changes are made visible to other transactions atomically and in an all-or-nothing manner. We define visibility in detail in Section 3.1.9.

While the SI idea is simple, it is not trivial to implement. Having only one version of an object makes it difficult to isolate the effects from and to other transactions. One way databases implements SI is with Multi-Version Concurrency Control (MVCC) discussed in Section 2.4.3.

2.3.4 Serializability

Serializability [Ber+95] is a consistency model that guarantees that the result of concurrent execution of transactions is equivalent to some serial execution of the transactions. Intuitively this means that transactions are made visible in a total order. This total order also applies the transactions corresponding snapshots and commits.

2.3.5 Causal Consistency

Under causal consistency each process may perceive a different serialization of operations, but they must respect certain ordering guarantees. Intuitively, under causal consistency, if two events have a natural ("causal") order, they become visible in that order, but unordered events ("concurrent") can become visible either way. In order to explain causal consistency, we introduce three new relations: *program order* ($\rightarrow$), *write-into* ($\mapsto$) and *happened-before* ($\rightsquigarrow$).

Program Order ($\rightarrow$)

Program order ($\rightarrow$) refers to the order in which operations are executed within a single process [Aha+95]. This order is defined by the sequence of instructions in the process's program.

The following properties hold for program order:

- For any two operations μ_i and μ_j executed by the same process P, if μ_i occurs before μ_j in the program order, then $\mu_i \rightarrow \mu_j$. This implies that the order of operations executed by a single process is maintained.
- Program order is a partial order. It only imposes an ordering on operations executed by the same process.
- Program order is not transitive across different processes. In other words, if $\mu_i \rightarrow \mu_j$ and μ_j belongs to process P_1 while μ_k belongs to process P_2, it does not imply that $\mu_i \rightarrow \mu_k$.
- The program order relationship is combined with other relationships, such as write-into ($\mapsto$) and happened-before ($\rightsquigarrow$), to define the causal order for operations in a distributed system. Causal order ensures that the partial orderings of operations established by these relationships are respected, even as different processes observe different orderings of all operations in H.

Write-Into ($\mapsto$)

Write-into ($\mapsto$) defines a dependency between write operations and read operations. We define writes as $w(x)v$, where x is the key and v is the value written and reads as $r(x)v$, where x is the key and v is the value read. Intuitively, if a value is read in a

state μ_i, it must have been written at some point. Ahamad et al. [Aha+95] define these relations as follows:

- All reads follow a write: If $\mu_i \mapsto \mu_j$, then there must be a key x and a value v such that $\mu_i = w(x)v$ and $\mu_j = r(x)v$.
- A read may depend on one write only. For any operation μ_j, there is at most one operation μ_i such that: $\mu_i \mapsto \mu_j$.
- If a read follows no write on a value, the value returned is the initial value: If for $\mu_j = r(x)v$, there is no μ_i such that $\mu_i \mapsto \mu_j$, then $v = \perp$.

Happened-Before ($\rightsquigarrow$)

Lamport defines the happened-before relationship ($\rightsquigarrow$) as follows [Lam78]:

- $\mu_i \rightsquigarrow \mu_j$ if $\mu_i \mapsto \mu_j$. If a process P_i sees two operations ordered in it's local execution, then these operations are linked with the happened-before relationship. If two operations of a process P_i,
- $\mu_i \rightsquigarrow \mu_j$ if $\mu_i \mapsto \mu_j$. If μ_j reads a value written by μ_i, then μ_i happened-before μ_j.
- $\mu_i \rightsquigarrow \mu_j$ if $\mu_i \rightsquigarrow \mu_k$ and $\mu_k \rightsquigarrow \mu_j$. Happened-before is a transitive relationship. If two operations μ_i and μ_j are not linked by the happened-before relationship, then they are not causally related, and these two operations are concurrent.

Causal Order

Causal consistency applies over the local history of each process in the system. They may all see a different ordering of all operations in H, but this order must respect the causal order of all operations. The causal order is the transitive closure of the union of program order ($\rightarrow$) and write-into ($\mapsto$) relations. The intuition of causal consistency is that while linearizability tries to impose a total order linked with real time, causal consistency attempts to impose a partial order linked with logical time. Program order imposes an order dependency that must be respected, and a data dependency is also introduced with the write-into relationship.

Causal consistency is related to the so-called four session guarantees. A session is an ordered set of operations. The term "session" is loosely defined and depends largely on the system or application being used. In this context, we understand a session as

the succession of operations observed by a process. These definitions are inspired by Viotti and Vukolić [VV16]:

1. **Read your writes**: If a process P applies a write operation $\mu_i = w(x)v$, then all subsequent reads without an interleaving write in this session must return this value $\mu_{i+n} = r(x)v$.

2. **Monotonic reads**: Let there be two operations applied by a process P: $\mu_i = w(x)u$ followed by $\mu_j = w(x)v$. Once any read operation has returned $\mu = r(x)v$, then no other operation may return $\mu = r(x)u$. Intuitively, this means that our data stores move only forward in time and cannot return previous values.

3. **Monotonic writes**: All writes belonging to the same session must be visible in the same order.

4. **Writes follow read** (also called session causality): Any write operation made during the session is ordered after a write in any session whose effect was seen by a read operation in this session. If $\mu_j = r(x)v$ is a read operation made during the current session, and $\mu_i = w(x)v$ is a write operation on the same key made during another session, μ_i reads the value written by μ_j: $\mu_i \mapsto \mu_j$, then if a write $\mu_k = w(y)u$ is made afterwards during the session, μ_i must be visible before μ_k.

2.3.6 Eventual Consistency

In Eventual Consistency (EC), the order of operations is not guaranteed. Instead in the absence of new updates, given the same set of operations, all replicas will eventually converge to the same state. Here replicas refers to a copy of the database that is stored on a different machine. EC is essentially a liveness condition whereas the others focus on safety.

2.3.7 Strong Eventual Consistency

Strong eventual consistency (SEC) is a stronger version of eventual consistency that focus on safety. In SEC, two replicas that have the same set of operations visible are in an equivalent state and any update delivered to some replica will eventually be delivered to all replicas.

To achieve SEC, developers use specific data structures that are designed to be convergent and offer deterministic merge operations, described in section 2.4.1.

2.3.8 Transactional Causal+ Consistency

Transactional causal+ consistency (TCC+) is a consistency model that is based on causal consistency. In TCC+, snapshots provide the isolation for transactions and the system uses data structure like Conflict-free Replicated Data Types (CRDTs) to provide strong convergence of the replicas.

2.4 Data structures

In this section we present data structures and techniques used in distributed systems.

2.4.1 Conflict Free Replicated Data Types

Conflict-free Replicated Data Types (CRDTs) are data structures that enforce the strong convergence property of SEC. CRDTs were introduced by Shapiro et al. [Sha+11] as a way to ensure that replicas can eventually merge into a consistent state, even in the presence of conflicts, by using deterministic merge functions and specialized data structures.

CRDTs achieve this by being built on the following properties:

Commutativity: The order in which concurrent updates are applied does not affect the final state. This means that if two updates A and B are concurrent, then applying A followed by B should yield the same result as applying B followed by A.

Associativity: The grouping of updates does not affect the final state. If three updates A, B, and C are applied, it does not matter if A is combined with B and then combined with C, or if A is combined with the result of B combined with C.

Idempotence: Applying the same update multiple times has the same effect as applying it once. This property ensures that if an update is accidentally duplicated, it will not cause any inconsistencies in the system.

There are two main types of CRDTs: state-based and operation-based.

State-based CRDTs (CvRDTs)

With State-based CRDTs, also called Convergent Replicated Data Types (CvRDTs), updates are propagated by sending the entire local state to other replicas, where they are merged with the local state. The merge function of a CvRDTs is commutative, associative, and idempotent. The merge can be defined as a least upper bound (LUB) over a join-semilattice[BM99]. State-based CRDTs can become large over time increasing the cost of sending the entire state to other replicas. To solve this Almeida et al. [ASB15] introduced Delta CRDTs, where only difference between the states are sent to other replicas.

Operation-based CRDTs (CmRDTs)

Operation-based CRDTs, also called Commutative Replicated Data Types (CmRDTs), are lighter as updates are propagated by sending only the operations to other replicas. The replica receives the update and apply them locally. Update operations are commutative, however, they are not idempotent. CmRDTs assumes a messaging layer that guarantees that messages delivered in a causal order and exactly-once.

2.4.2 All-or-Nothing: Shadow paging vs write-ahead logging

There are two common techniques to achieve the all-or-nothing property of transactions: shadow paging and write-ahead logging.

Shadow paging

Shadow paging is originally a disk-based technique where a database maintains two separate page tables: current and shadow. When a transaction starts, a copy of the current page is created, the shadow page. The shadow page is not referenced anywhere so updates can be made to it without affecting the current page. When the transaction commits, the shadow page table becomes the current page table in an atomic operation. In case a transaction aborts, the shadow page is discarded.

Write-ahead logging

Write-ahead logging (WAL) [Moh+92], also known as journaling, is a technique where a database maintains a log of operations called journal or log. When a transaction starts, it writes all its operations to the log. When a transaction commits, a commit record is written in the log. Depending on the implementation, the updates can be written during the transaction execution or when the transaction commits. In case a transaction aborts, an abort record is written in the log and the log is used to undo the changes made by the transaction.

Write-ahead logging is also used to perform recovery in case of a crash. As the log provide a history of operations that can be reapplied to the database to bring it to a consistent state.

2.4.3 Update-in-place vs MVCC

Update-in-place and Multi-Version Concurrency Control (MVCC) are different approaches to managing concurrent access to data in a database system. Both methods provide a consistent view of the data and maintain isolation between concurrent transactions.

Update-in-place

Update-in-place is a technique where the database directly modifies data on disk or in-memory when a transaction makes changes. This approach is straightforward but require locking mechanisms to maintain consistency and isolation between concurrent transactions.

Update-in-place are space efficient but locking can cause contention and limit concurrency, as transaction needs to wait for locks to be released before they can proceed.

Multi-Version Concurrency Control

Multi-Version Concurrency Control (MVCC) [Ree78] provides isolation by maintaining multiple versions of each object. With MVCC transactions read the database at a given time without interference from other transactions providing SI, and writes are isolated from reads and writes from other transactions.

Two common ways of implementing MVCC is to maintain multiple version of each object or to maintain a log of all the operations like we described in 2.4.2 that have been performed on the database.

Versioning maintain multiple version of the same object to allow transaction to read them without another transaction modifying the object. And also this allows a transaction to write to the database without other transaction being able to see the changes until the transaction is committed. *Versionning* allows for fast reads but it is space consuming as it requires to store multiple version of the same object and it requires additional application logic in order to do correct garbage collection.

Logging is a more space-efficient way of implementing MVCC as it is logging all the operations that have been performed on the database. This way a transaction can reconstruct (materialize) any version of an object by replaying the log. While overall space efficient compared to versioning, it is slower as it requires to replay the log from the beginning to materialize an object.

2.4.4 Timestamps and Clocks

Timestamps are used in database systems to order events, such as transactions or updates, and maintain a consistent view of the data. Scalar timestamps (e.g., Lamport timestamps) and vector clocks are two common mechanisms used to order events in distributed systems.

Scalar Timestamps are simple, monotonically increasing values assigned to events. They provide a partial order of events, which is sufficient for many applications. However, scalar timestamps cannot capture causality between events, leading to potential inconsistencies in the presence of concurrent updates.

Vector Clocks are arrays of scalar timestamps, one for each process in the distributed system. They maintain a causal order of events, capturing the happened-before relationship between events in the system. Vector clocks allow for better visibility control and consistency in distributed systems, at the cost of increased storage and communication overhead.

2.4.5 Log-Structured Merge-tree

Log-Structured Merge-tree (LSM-tree) is a data structure introduced by O'Neil et al. [O'N+96], designed to provide efficient and scalable write performance in key-value stores.

An LSM-tree is composed of multiple levels, at least two, with first level C_0 being in-memory and the rest on disk ($C_1, C_2, ...$). One property of LSM-tree is that each level has a different size, with the first level being the smallest and most frequently updated and the last level being the largest and less frequently updated. C_0 stores key-value pairs in an in-memory data structure. When C_0 is full the level is flushed to disk to next level. When a persistent level is full, it is merged with the next level. When a read is performed, the levels are scanned from the smallest to the largest until the key is found.

The LSM-tree architecture is used in modern databases and storage systems, such as LevelDB, RocksDB, and Apache Cassandra, due to its ability to handle high write throughput while having a read complexity of $O(\log n)$. Modern LSM-tree implementations store smaller level on fast storage and larger levels on slower but bigger storage. This approach balances performance and storage cost as frequent accessed level benefits from faster storage and larger level can be stored cost-effective storage. They also use optimizations like using a write-ahead log for the memory level to provide crash resistance. Other notable optimizations are, SSTables to store the data on disk, which are immutable and sorted key-value stores, and Bloom filters Bloom [Blo70] to reduce the number of disk reads.

2.4.6 Distributed Transactions

2.4.7 Sharding

Sharding is a technique used to distribute data across multiple nodes. Where a node can be a physical machine or a virtual machine. It involves splitting the data into multiple partitions and assigning the storage and handling of these partitions to different nodes. This approach allows the database to spread a workload among multiple nodes, thus scaling the database horizontally.

There are multiple ways of sharding a database, the most common are:

- **Range-based sharding**: Data is partitioned based on a specific range of keys.

- **Hash-based sharding**: A consistent hashing function is applied to an attribute value to determine the shard where the data should be stored. This approach can help distribute the data evenly across shards, reducing hotspots and improving performance.

Replication

Replication is a technique used to further distribute data. A replicated database maintains multiple copies of the same data on different nodes in order to improve availability and fault tolerance. Replication can be performed at different levels of granularity, from the entire database to partition level or even individual objects. A full copy of the database is called a replica. Having replicas allows the database to spread the workload among multiple nodes, thus improving performance and availability.

There are multiple replication strategies used:

Synchronous replication is a strategy where the database waits for the replicas to acknowledge the write before returning to the client. This is needed to provide strong consistency across replicas but result in high latency for writes and can lead to unavailability in a replica crashes.

Asynchronous replication is a strategy where the database does not wait for the replicas to acknowledge the write before returning to the client. Once the transaction is committed to the local replica it returns to the client. And asynchronously sends the updates to other replicas.

K-Stability is a strategy where the database waits for k replicas to acknowledge the write before returning to the client. This strategy is a trade-off between synchronous and asynchronous replication. It provides strong consistency and availability as long as k replicas are available.

2.4.8 Two-Phase Commit

One of the primary challenges in implementing transactions in distributed systems is ensuring that all participating nodes agree on the outcome of the transaction, either

committing or aborting it. Two-phase commit (2PC) is a widely used protocol for achieving this agreement. In 2PC, a coordinator node is responsible for initiating the transaction and collecting votes from participating nodes to decide whether to commit or abort a transaction. The 2PC algorithm can be described in two main phases:

1. **Prepare Phase**: The coordinator sends a prepare message to all participating nodes, requesting them to prepare for the transaction commit. Each participant ensures that it can commit the transaction, locks the necessary resources, and responds with a vote (either commit or abort).

2. **Commit or Abort Phase**: After collecting votes from all participants, the coordinator decides whether to commit or abort the transaction based on the received votes. If all participants vote to commit, the coordinator sends a commit message to all participants, indicating that they should commit the transaction. Otherwise, the coordinator sends an abort message, instructing participants to abort the transaction.

2.4.9 Challenges in Ensuring Consistency in Distributed Systems

Large-scale distributed systems are used everywhere, from social networks to e-commerce platforms and financial services. These systems often rely on distributed databases designed to scale horizontally, providing high availability and fault tolerance.

In such systems, writing data to a database involves more than just updating a single machine with a single copy of the data. As previously discussed, data is distributed across multiple nodes, and multiple copies of the data are maintained while adopting weaker consistency models unlike Serializability.

One of the primary challenges in ensuring consistency in distributed systems stems from the CAP theorem [GL02] [Bre12]. This theorem states that, in the presence of a network partition, a distributed system must choose between strong consistency and availability. For most large-scale applications requiring high throughput, prioritizing strong consistency at the expense of availability is not a viable option.

Moreover, implementing the techniques outlined in the previous sections to ensure consistency in distributed systems can be quite complex, particularly in systems that are built in an ad-hoc manner. This complexity can lead to inconsistencies and errors, potentially causing significant problems for the applications relying on these

systems. We believe that there is a need for a formalism-based approach to design and build reliable and consistent distributed storage.

2.4.10 Conclusion

In this chapter, we have presented different types of databases and the different properties they provide. We present different consistency models and ways to describe how operations are ordered in a distributed system. We have also presented the different techniques used to handle data access and modifications.

We highlighted the challenges of ensuring consistency in distributed systems and the need for a formalism-based approach to design and build reliable and consistent distributed storage.

Part II

Contributions

As we venture into the second part of this thesis, we shift our focus to the contributions of our research, centered around the design and implementation of a formally specified and modularly composed database backend. Our objective is to demonstrate the feasibility and effectiveness of a rigorous approach to storage system design, which is both theoretically sound and practically viable in terms of performance and scalability.

Building upon the foundational concepts explored in the first part of the thesis, we begin by introducing the formal specification of a database backend. This involves defining the system model and the semantics of transactions, which serve as the basis for reasoning about the correctness and behavior of our proposed storage system. By formalizing the system's properties, invariants, and constraints, we aim to ensure that our implementation adheres to the desired consistency and isolation guarantees, providing a solid foundation for the development of a reliable and robust storage system.

In the third part of the thesis, we delve into the implementation of our specifications into multiple variants, employing either a journal-based backend with Write-Ahead Logging (WAL) or a map-based backend. This exploration allows us to analyze the trade-offs and benefits of each approach in detail, examining factors such as performance, scalability, and crash recovery capabilities. By comparing and contrasting the different backend implementations, we aim to provide valuable insights into the practical considerations and challenges that arise in the design and implementation of modern storage systems.

In the fourth part of the thesis, we discuss the composition of multiple database backends to achieve a desirable balance of features. We introduce the Conductor, a mechanism that handles the composition of multiple combined backends while enforcing the system specification. This approach allows the resulting database to capitalize on the features, such as crash recovery, of each backend, while mitigating some of the drawbacks. By demonstrating the feasibility and effectiveness of a modular composition approach, we hope to inspire new avenues of research and innovation in the field of storage systems.

Finally, we touch upon ongoing and unfinished research efforts to adapt the non-distributed semantics presented in this thesis to accommodate distribution. We discuss the challenges of tracking causality in distributed systems and maintaining a consistent view of the database state while still providing the same guarantees. Our goal is to pave the way for future research in this area, exploring the potential of formally specified and modularly composed storage systems in the context of distributed environments.

In summary, the second part of this thesis presents our contributions to the field of storage systems, focusing on the design, implementation, and evaluation of a formally specified and modularly composed database backend. By showcasing the potential of a rigorous and principled approach to storage system design, we hope to encourage further research and innovation in this exciting and rapidly evolving domain.

3 Formal Specification of a Database Backend

3.1 System Model

In this chapter we explore the key concepts, principles, and abstractions involved in designing a database backend that can effectively support transactions, functionality like journaling, crash resistance, read write performance, and consistency guarantees. We begin by introducing the core components of our database backend, use them to define the system model, and then present the semantics of transactions.

3.1.1 Fault model

We first define the fault model that we will use to reason about correctness and behavior of a storage system. Failures are assumed 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. Meaning that a read will block until the value is available, but a write may fail if the memory is busy. 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 non-deterministic prefix of the preceding writes terminated successfully.

3.1.2 Keys, values, and timestamps

A store operates on a key-value interface, where each entry is associated with a unique key, and a key maps to multiple data versions. Each version consists of a pair of value and timestamp, which itself identifies the specific version of the key. To better understand these components, let us delve into the details of keys, values, and timestamps.

Keys

A Key is an element of the opaque type Key, represented by the meta-variable k. A Key serves as unique identifier of a data item within the store. Keys are opaque and can only be compared for equality; and no other operations can be performed on them. This property ensures that keys remain distinct, unambiguous identifiers for accessing and updating data items in the store. A key maps to multiple versions of the associated data, allowing for tracking the history of changes and supporting various consistency and isolation models in a transactional system.

Values

A value, on the other hand, represents the actual data associated with a given key and version. It belongs to opaque type $Value$, represented by meta-variable v.

Timestamps

A Timestamp plays several roles, in particular identifies a specific version of a key-value pair within the store. It is an element of opaque type `TS`, represented by meta-variable `t`. A timestamp is associated with an update made to a key-value pair, providing a means to track the history of changes and enabling the retrieval of specific versions. Timestamps can be compared to determine the order of updates, allowing the store to maintain a consistent view of the data and to provide the necessary guarantees for concurrent transactional systems.

Timestamps are partially ordered by $\leq$. This does not preclude strong-consistency models, which require a total order. Timestamps are *concurrent* if not mutually ordered: $\mathtt{t}_1 \parallel \mathtt{t}_2 \triangleq \mathtt{t}_1 \not\leq \mathtt{t}_2 \wedge \mathtt{t}_2 \not\leq \mathtt{t}_1$.

3.1.3 Timestamps and clocks

Two timestamps have a least upper bound $\max$ and a greatest lower bound $\min$. We generalize $\min$ and $\max$ to a set in the standard fashion; as a shorthand, $\max_{\mathcal{T}}(x) \triangleq \max(\Pi_x(\mathcal{T}))$ is the least-upper-bound of the x dimension[1] in the set of tuples $\mathcal{T}$. 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.

As is standard, the timestamp type can be represented by a vector of integers, with the classical definitions for $\leq$ or $<$ [Mat88; Fid88]. 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 $\min$. Timestamps become particularly useful in the context of replication.

A *clock* is an object that returns unique and monotonically-increasing timestamps.

3.1.4 Effects

The current state of the store results from applying a *history* of updates, which we call *effects*.

We define *effect*, denoted by the meta-variable δ and the domain of effects $\mathcal{E}\mathit{ff}$, as a function that maps a value to another value. We distinguish between two types of effects: *non-assignments* and *assignments*.

An *operation*, or non-assignement, is the reification of an update, representing a function that maps a value to another value. For example, the increment function $\mathsf{incr}_{10}(\mathsf{arg})$ adds 10 to its argument, which is assumed to be an integer value. We use the meta-variable *Op* to denote an operation and the domain of operation $\mathcal{O}p$.

An *assignment*, denoted by $:-$, is the constant effect. For instance, ":= 27" always returns 27. Since an assignment masks any earlier assignments or effects to the same key, the earlier ones can be ignored.

Assignments are useful for simplifying and optimizing the history of a key-value pair by consolidating previous effects into a single assignment or by combining the last assignment with subsequent effects into a new assignment. This consolidation allows the system to ignore and potentially garbage collect previous effects, reducing the overhead of tracking the entire history of effects on a key.

[1] In this context, the term "dimension" refers to the position within the timestamp tuple. For example, given a tuple (a, b, c), the first dimension corresponds to the value of a, the second to b, and the third to c.

3.1.5 Traces

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 $\overset{vis}{\to}$ defined in the next Section 3.1.6), produced by the rules of Figure 3.1; a trace is denoted by meta-variable Θ.

3.1.6 History

Before diving into the details of operations and assignments, it is essential to introduce the concept of a history. A *history* is a sequence of all effects applied in a given execution. It is a linearization of a trace. By organizing effects into histories, it becomes easy to manage updates, resolve conflicts in concurrent transactional systems, and optimize the storage and retrieval of data. We represent history order by the relation $\overset{\text{hist}}{\to}$.

For example, the history:

$$H_k = Op_k(19) \overset{\text{hist}}{\to} Op_k(90) \overset{\text{hist}}{\to} k := 42 \overset{\text{hist}}{\to} \delta_k(1) \overset{\text{hist}}{\to} \delta_k(5)$$

Represents the sequence of effect $Op_k(19)$, $Op_k(90)$, $:= 42$, Op_5, and $Op_k(1)$ applied to k.

Here, the assignment $:= 42$ masks (i.e. overrides) the effects of $Op_k(19)$ and $Op_k(90)$, as their outcome is replaced by the assignment resulting in an equivalent *history*:

$$H_k := 42 \overset{\text{hist}}{\to} \delta_k(1) \overset{\text{hist}}{\to} \delta_k(5)$$

If there are concurrent δ, they can be merged by an appropriate merge operator, which is commutative, associative, and idempotent [Sha+11; Bur+14]. For instance, the widely used last-writer-wins approach merges concurrent assignments by imposing a deterministic total order on their timestamps. This method ensures that the most recent update to a key-value pair takes precedence, enabling a consistent view of the data in a concurrent transactional system.

In the following subsection, we will explore the roles of effects in the context of histories and their impact on the overall efficiency and consistency of the system.

3.1.7 Snapshots

A snapshot represents a specific view of the history, which includes all operations up to a given point in time.

A snapshot is characterized by a *snapshot timestamp*, denoted `dt`. The *snapshot timestamp*, or *dependency timestamp*, indicates the point in time at which the snapshot was taken, and is used to determine the *visibility* of operations in the snapshot. We discuss visibility in Section3.1.9.

3.1.8 Transactions

A transaction is a sequence of operations. All it reads come from a same *snapshot*, as defined by the transaction's *dependency timestamp* that includes all transactions that committed before such as:

$$\forall \delta_{\texttt{ct}} \in \texttt{dt} \iff \texttt{ct} < \texttt{dt}$$

The operations of a running transaction are not visible outside of it, ensuring *isolation*. A transaction terminates in an all-or-nothing manner: it either *aborts*, making no changes to the store, or it *commits* with a *commit timestamp*. In the latter case, its operations become visible in the store atomically, labeled with the commit timestamp.

A transaction descriptor is a tuple composed of its identifier τ, its *dependency timestamp* `dt`, its *read set* $\mathcal{R}$, its *dirty set* $\mathcal{W}$, its *state buffer* $\mathcal{B}$, and its *commit timestamp* `ct`. The tuple is represented as:

$$(\tau, \texttt{dt}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \texttt{ct})$$

Where $\tau$is the transaction identifier, $\mathcal{R}$ is the set of keys read by the transaction, $\mathcal{W}$ is the set of keys written by the transaction, $\mathcal{B}$ is a buffer where are stored read value and updates of the transaction.

An example transactional execution is illustrated in Figure 5.1(a).

3.1.9 Visibility

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

- $\delta \overset{vis}{\to} \delta'$ if both execute in the same transaction and δ' executes before δ'.
- $\delta \overset{vis}{\to} \delta'$ if they belong to different transactions, τ and τ' respectively, where τ' can read from τ. Formally, τ has committed, and $\tau'.\texttt{dt}$ is greater than $\tau.\texttt{ct}$.

Visibility is a partial order, meaning that even if timestamps form a total order (strong consistency), transactions might still be concurrent if for two transaction $\tau$and τ', their range $]\texttt{dt}, \texttt{ct}]$ overlap, for instance under Snapshot Isolation.

Committed operations that are not ordered are said to be *concurrent*, denoted as $\delta \| \delta$'. Note that an uncommitted operation is *not* considered concurrent to another transaction since it is not visible outside of its transaction.

A transaction's snapshot `dt` has visibility of an operation if and only if the latter's transaction commit timestamp `ct` is strictly less than `dt`.

3.1.10 Values associated with a store

Assume that at some point in time, the operations on key k are $\{\delta_i, \delta_j, \dots\}$, ordered by $\overset{vis}{\to}$. Intuitively, the current value associated with key k can be computed by applying all the visible operations to k in order, merging concurrent operations.

Operations earlier than the most recent assignment can be ignored, as they do not influence the result. If a store contains only assignments, then its expected value for key k is $\max_{\overset{vis}{\to}}\{\delta_i, \delta_j, \dots\}$, defined as the most recent assignement, or if multiple assignments are concurrent, the most recent after of merging them.

For example, suppose store σ maps assignment 27 to version 100 of key k (assuming, for the sake of example, scalar timestamps). Then, $\mathsf{lookup}(\sigma, k, 101)$ should return 27. If there are no other versions between 100 and 110, $\mathsf{lookup}(\sigma, k, 111)$ should also return 27. If the next mapping is at timestamp 120, to $\mathsf{incr}_{10}()$, then $\mathsf{lookup}(\sigma, k, 121)$ should return 37.

We formalize this in Section 3.2.1.

BEGIN_TXN

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

INIT_KEY

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

READ

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

UPDATE

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

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

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

Fig. 3.1.: Operational Semantics of Transactions

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

Fig. 3.2.: Store interface

$$\text{[H]}\quad \delta_k \sqcup \delta'_{k'} = \delta'_{k'} \sqcup \delta_k \;=\; \begin{cases} \delta_k & \text{if } \delta'_{k'} = \bot & \text{(null effect)} \\ \delta'_{k'} & \text{if } \delta_k = \bot & \text{(null effect)} \\ \delta_k & \text{if } \delta_k = \delta'_{k'} & \text{(idempotence)} \\ \delta'_{k'} \circ \delta_k & \text{if } k \neq k' & \text{(independent keys)} \\ \delta'_{k'} \circ \delta_k & \text{if } \delta_k \overset{vis}{\rightarrow} \delta'_{k'} & \text{(apply in } \overset{vis}{\rightarrow} \text{ order)} \\ \delta_k \circ \delta'_{k'} & \text{if } \delta'_{k'} \overset{vis}{\rightarrow} \delta_k & \text{(apply in } \overset{vis}{\rightarrow} \text{ order)} \\ \mathsf{merge}(\delta_k, \delta'_{k'}) & \text{if merge def.} \wedge\ \delta_k \parallel \delta'_{k'} & \text{(merge concurrent)} \\ \text{undefined} & \text{otherwise} & \text{(error)} \end{cases}$$

Fig. 3.3.: Effect composition

The first two lines state that the null effect $\bot$ (corresponding for instance to a non-initialized 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.

3.2 Formal Model of Transactions

In this section, we provide a generic formal specification of transactions and store operations. Later sections will formalize specific variants.

3.2.1 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 formalize this intuition with the *effect composition* rule in Figure 3.3. 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.

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

$$\begin{aligned} \delta \sqcup \delta' &= \delta' \sqcup \delta && (commutative) \\ (\delta'' \sqcup \delta') \sqcup \delta &= \delta'' \sqcup (\delta' \sqcup \delta) && (associative) \\ \delta \sqcup \delta &= \delta && (idempotent) \end{aligned}$$

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., $\mathsf{safe}(\Theta, k, \mathtt{t}) \triangleq \bigsqcup\{\delta_k^{t'} : \delta_k^{t'} \in \Theta \wedge t' < \mathtt{t}\}$, where Θ is a legal trace of effects, and $\delta_k^{t'}$ is an effect to key k committed with timestamp t'. A legal trace is one produced by the semantic rules specified in Section 3.2.2.

Note that an assignment overwrites the previous history of the key, i.e., $\delta_k \overset{vis}{\rightarrow} \delta'_k \wedge \delta'_k \in \mathit{Value} \implies \delta_k \sqcup \delta'_k = \delta'_k$. More generally, if a key's value is known at some point, earlier updates to 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 computing the current state of a key. *Lookup* its most recently-known value (from a cache or a checkpoint) and apply later updates in visibility order. If we impose that the dependency timestamp of running transactions be greater than *min_dt*, then the prefix of Θ up to *min_dt* can be replaced by the *checkpoint* of values computed at *min_dt*. We return to checkpointing in Section 5.1.1.

3.2.2 Semantics of transactions

Figure 3.1 presents the semantics of transactions. The specification is fully formal and unambiguous: we use it to reason about the system, and it is meant to be eventually translated to the language of a proof tool such as Coq. Most interestingly, it can be read as pseudocode, as we explain now.

3.2.3 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.

3.2.4 Parameters

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.

The rules describe a transaction system, which is a tuple $(\sigma, \mathcal{F}_\sigma, \mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}_\mathrm{r})$ consisting of a store σ, its associated field $\mathcal{F}$, 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. We will ignore $\mathcal{F}_\sigma$ until Section 5.1.

Recall from Section 3.1.8 that a transaction descriptor is a tuple composed of its identifier τ, its *dependency timestamp* `dt`, its *read set* $\mathcal{R}$, its *dirty set* $\mathcal{W}$, its *state buffer* $\mathcal{B}$, and its *commit timestamp* `ct`. The two timestamps define visibility between transactions, as defined previously (Section 3.1.9). Initially, after rule BEGIN_TXN, the sets and the buffer are empty and the commit timestamp is invalid. For each key that is accessed, rule INIT_KEY initializes the buffers, and rule UPDATE updates them. Computation of the actual commit timestamp may be deferred to the COMMIT rule.

The API of a *store* consist of commands lookup, doUpdate and doCommit, specified in Figure 3.2. These commands serve the following general purposes:

- lookup(σ, k, `dt`) is responsible for retrieving the current value of a key $k$from store σin the context of transaction τ, considering its dependency timestamp and read set.
- doUpdate(σ, τ, k, δ) is used to modify the value of a key k, updating the dirty set $\mathcal{W}$ and state buffer $\mathcal{B}$ accordingly to reflect the changes made within the transaction.
- doCommit(σ, τ, $\mathcal{R}$, $\mathcal{B}$, $\mathcal{W}$, `ct`) is in charge of committing transaction τ, atomically applying its effects to the store $\sigma$and setting the commit timestamp.

These commands will be specialized for each specific store variant, as detailed later: the map-based variant in Section 4.2.1, the journal-based variant in Section 4.2.3, and composition variant in Section 5.1.

We now describe the rules in turn, then identify the implementation issues that they raise.

3.2.5 Transaction begin

BEGIN_TXN describes how API begin(dt) begins a new transaction with dependency timestamp 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 dt, passed as an argument. Remember that a snapshot includes all transactions that committed with a strictly lesser commit timestamp.

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.

3.2.6 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 $\mathcal{T}_\mathrm{r}$. The second one checks that k is not already in the read set, ensuring that the state buffer is initialized 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.

In Rule READ, API $\mathsf{read}_\tau(k)$ returns the content of the state buffer. It does not modify the store. The first clause pulls out the transaction descriptor, as above. The second one requires that the key is in the read set, thus ensuring that INIT_KEY has been

applied. The third one identifies the return value as the value of k stored in the state buffer.

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

3.2.7 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 $\mathcal{T}_\mathrm{r}$ to $\mathcal{T}_\mathrm{a}$, marking it as aborted. It does not make any other change.

API commit$_\tau$(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 $\mathcal{T}_\mathrm{c}$); 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 moved to the set of committed transactions.

3.3 Conclusion

In this chapter, we have presented the formal specification of a transaction within the context of distributed systems. We have laid the groundwork for understanding the key components of a transaction, such as effects, operations, assignments, and effect composition rules.

As we conclude this chapter, it is important to recognize that understanding the formal semantics and specifications of transactions is only the first step towards building effective distributed transactional systems. The challenge lies in translating

these theoretical concepts into practical implementations that can be efficiently deployed in real-world scenarios.

In the next chapter, titled "From Specification to Implementation" (4), we will explore the process of turning the formal transaction specifications introduced in this chapter into concrete implementations for distributed systems. We will discuss various strategies for implementing these concepts, address the challenges and trade-offs involved in designing and optimizing transactional systems, and investigate techniques for ensuring correctness, performance, and fault tolerance.

4 From Specification to Implementation

In this chapter, we will delve into the process of making the formal specification into functional implementations. The objective is to create multiple implementations that closely follows the specification, supporting different functionalities while having good performance. By adhering to the specification, we can ensure that the resulting system exhibits the intended behaviours and properties, while being able to adapt and extend the implementations for different store variants.

We start by discussing the general approach taken to convert the formal rules and parameters into code, focusing on the techniques and strategies employed to maintain a high degree of correspondence between the specification and the implementation. Then we present multiple store implementations, showcasing how the specification can be translated into different stores, each with their unique characteristics and optimizations.

4.1 Implementation Approach

In this section, we discuss the overall approach taken to implement the formal specification, outlining the key principles, strategies, and design patterns employed. In order to implement the specification, we chose Java as the programming language, primarily due to our familiarity with the language. It also benefits from an extensive ecosystem and strong typing. Utilizing the Java Standard Library, we were able to leverage the collection of classes and utilities it provides, ensuring a robust and reliable implementation.

For future work, using Java we can easily interface with different databases, such as MySQL, PostgreSQL, and MongoDB, and use them as the underlying store.

During the initial stages of development, we used the Google Guava library to implement certain aspects of the system. Google Guava is a popular library that offers a range of helpful utilities, extending the capabilities of the Java Standard

Library. However, as our implementation progressed, we decided to simplify our codebase by removing optimizations that relied on Guava.

4.1.1 Implementing common components

Timestamps and timestamp generator

In the current implementation of the system, timestamps are represented as simple integer values, allowing for easy management and comparison. However, the system is designed to be flexible and easily extensible to support other timestamp representations, such as vector clocks, in the future.

The Timestamp class contains a single integer field and implements the *Comparable* interface for easy comparison between instances. To prepare for potential extension to vector clocks, it provides a *compare* method returning a *TimestampCompareType* enumeration value (LOWER, EQUAL, HIGHER, or UNKNOWN), allowing for comparison functions that can handle concurrent timestamps, essential when working with vector clocks.

The TimestampGenerator class ensures unique and monotonically increasing commit timestamps. It uses an AtomicInteger, *min_allowed_ct*, to store and increment the current minimum allowed commit timestamp value. Another AtomicInteger, *max_allowed_dt*, keeps track of the maximum allowed dependency timestamp value. A sorted set, *commitRunning*, stores the commit timestamps of currently running transactions. We will talk about the relationships and interaction between these values in more detail in Section 4.1.4.

The generator provides methods for generating new timestamps (next), peeking at the current maximum allowed dependency timestamp (peek), and notifying when a commit ends (endCommitNotify). To support recovery, the generator offers a persist() method for persisting the current minimum allowed commit timestamp value to disk. The restore() method reads the value from disk and returns it, and a static delete() method allows for deleting the persisted timestamp file if necessary.

Having a persistency mechanism is not strictly necessary for the system to function, as we consider a persistent store to be able to compute the last commit timestamp from the store information.

CRDTs and their implementation

We use CRDTs (Conflict-free Replicated Data Types) as a means of managing concurrent updates within the transactional framework. To implement CRDTs, we have created an abstract class, which serves as the foundation for more specific CRDT types. We implemented two types of CRDTs for our variants: PN Counters and Last-Write-Wins (LWW) Registers. This design choice allows for easy extensibility, enabling the addition of more complex CRDTs as needed.

The abstract CRDT class provides a common interface for the concrete CRDT implementations, facilitating interaction and management of these data structures. Each concrete CRDT class inherits from the abstract CRDT class, implementing necessary methods such as update and merge.

The merge operation in each CRDT type guarantees the least upper bound property, ensuring that the system converges to a consistent state.

PN Counters : PN Counters, which allow concurrent increments and decrements without conflicts, consist of two separate counters: one for increments (P) and one for decrements (N). When implementing the PN Counter class, the update and merge methods are defined to support concurrent updates. In the merge operation, the system ensures the least upper bound property by taking the element-wise maximum of the P and N counters from both CRDT instances. The PNCounter implementation here can be used as both state-based and operation-based, depending on the store variant. For state-based stores, each transaction will have a copy of the PNCounter, and the merge operation is performed during lookup. For operation based stores, the increment and decrement operations are stored and applied to an old or new copy of the PNCounter during lookup. The positive and negative values for each client are stored in a ConcurrentHashMap, with a ClientUid as the key which is a unique identifier for each client. Each ClientUid maps to a ConcurrentHashMap, which stores the timestamp and value for each operation. The merge operation takes another PNCounter as an argument, and for each ClientUid in the other counter, it checks if the client is in this counter. If the client is not in this counter, it adds it. If the client is in this counter, it merges the values. The merge operation is shown in Listing 4.1.

Arguably for our current implementation it can be less complex. In our implementation a client can execute only one transaction at a time, so we do not need to keep track of all the operations. We chose to keep a tried and tested implementation, and it also allows for future extensibility.

```java
public void merge(Value<?> other) {
  if (!(other instanceof PNCounter)) {
      throw new IllegalArgumentException("Can only merge
          PNCounters");
  }
  PNCounter otherNew = (PNCounter) other;
  // for each ClientUId in the other counter
  for (ClientUId client : otherNew.positive.keySet()) {
      // if the client is not in this counter, add it
      if (!positive.containsKey(client)) {
          positive.put(client, otherNew.positive.get(client))
              ;
      } else {
          // if the client is in this counter, merge the
              values
          positive.get(client).putAll(otherNew.positive.get(
              client));
      }
  }
  // for each ClientUId in the other counter
  for (ClientUId client : otherNew.negative.keySet()) {
      // if the client is not in this counter, add it
      if (!negative.containsKey(client)) {
          negative.put(client, otherNew.negative.get(client))
              ;
      } else {
          // if the client is in this counter, merge the
              values
          negative.get(client).putAll(otherNew.negative.get(
              client));
      }
  }
}
```

Listing 4.1: Merge implementation for PN Counters.

LWW Registers : Last-Write-Wins (LWW) register is a straightforward CRDT type that resolves conflicts between updates by selecting the update with the highest timestamp. Each LWW register contains a value and an associated timestamp. When two updates occur concurrently, the value with the highest timestamp takes precedence. In the case of a tie, a deterministic conflict resolution strategy (such as

lexicographic ordering of replica IDs) is employed. This guarantees the least upper bound property.

Transaction

The Transaction class represents a transaction descriptor, as defined in the formal part of the specification (Section 3.1.8). The Transaction class contains fields corresponding to each element of the transaction descriptor tuple: the transaction identifier (*tid*), dependency timestamp (*dt*), read set (*readSet*), dirty set (*dirtySet*), state buffer (*stateBuffer*), and commit timestamp (*ct*). These fields are initialized in the constructor, and getter and setter methods are provided for accessing and updating them.

The main purpose of the Transaction class is to maintain the state and information associated with a given transaction throughout its execution. Each transaction object maintains a private state buffer, which stores the transaction's view of the data it has read or written, allowing for transactional isolation and atomic visibility upon commit. For every write the *key* is added to the dirty set. This way, by creating a separate Transaction object for each transaction, the system achieves isolation between concurrent transactions, ensuring that the operations of a running transaction are not visible outside of it.

Additionally, the Transaction class provides methods for adding values to the state buffer and updating the dirty set, based on the specific CRDT types involved. These methods ensure that the transaction system correctly handles the different types of CRDTs, such as LWW and PNCounter, when updating the state buffer.

4.1.2 Implementation challenges and considerations

A close examination of the specification and our initial prototype implementations has allowed us to uncover several interesting aspects and potential implementation challenges.

1. Transactions access the store concurrently in INIT_KEY, UPDATE, and COMMIT. However, they never make conflicting access to the same key-version concurrently. Consequently, the corresponding synchronization can be lightweight, reducing the potential overhead of ensuring transactional consistency.

2. The set of committed transactions $\mathcal{T}_c$ grows without bound. However, it is only used to ensure the uniqueness of transaction identifiers and commit timestamps. Instead of the full set, we can maintain only a summary of the identifiers and timestamps in use (not the full set), in order to minimize resource consumption. This approach, which reduces to a few atomic integers, is discussed in Section 4.1.4.

3. While $\mathcal{T}_a$ is also unbounded, it serves merely as a notational convenience and does not require implementation.

4. COMMIT is the rule that ensures a transaction's durability. A significant challenge lies in ensuring that its transition appears atomic despite failures. This issue is discussed in Section 4.1.4.

5. Enforcing the visibility premise of COMMIT necessitates synchronization between concurrent transactions. This synchronization process is examined in depth in Section 4.1.4.

6. The store accumulates versions without bound. To mitigate this issue, a garbage collection strategy can be employed to remove obsolete versions. This approach is discussed in Section 5.1.5.

There are numerous possible implementations of the specification, including those with or without sharding, with or without replication, using a Write-Ahead Log (WAL), employing a cache, or utilizing multiple storage layers. Existing literature often explains these variants informally, resulting in complex descriptions that deviate from the specification. We argue that these challenges can be addressed systematically by composing a small number of basic variants. This approach simplifies understanding and implementation, enabling the development of more robust and efficient systems in various contexts.

4.1.3 Enforcing the causality premise

A *causally consistent cut* is a set of effects that is closed under the visibility relation, i.e., if some effect is in the cut, all effects that are transitively visible to it are also in the cut.

$$causal\_cut(\Delta \subseteq \mathit{Eff}) \triangleq \forall \delta, \delta' \in \mathit{Eff} : \delta \overset{vis}{\rightarrow} \delta' \wedge \delta' \in \Delta \implies \delta \in \Delta$$

If we consider whole transactions, then:

$$causal\_cut(\mathcal{T} \subseteq \mathcal{T}_r \cup \mathcal{T}_c) \triangleq \forall \tau, \tau' \in \mathcal{T} : \tau.\texttt{ct} < \tau'.\texttt{dt} \wedge \tau' \in \mathcal{T} \implies \tau \in \mathcal{T}$$

The premise $causal\_dep(\texttt{dt})$ in Rule BEGIN_TXN requires the transaction's dependencies to form a causally-consistent cut. This means that, for any transaction τ that this transaction depends upon, it transitively depends on any transaction τ' that τ itself depends on. Formally:

$$causal\_dep(\texttt{dt}) \triangleq causal\_cut(\{\tau \in \mathcal{T}_c | \tau.\texttt{ct} < \texttt{dt}\})$$

The implementation naturally enforces this premise by maintaining an upper bound on the commit timestamps of transactions that are known to have committed. This is correct because, thanks to the visibility premise discussed next, transactions commit in timestamp order.

Our implementation maintains a single, monotonically increasing, atomic integer to represent the timestamps. To enforce this premise Timestamp Generator advances $max\_allowed\_dt$ to `cta` only when all the transactions with `ct` less than `cta` have committed. This way any transaction that starts with an allowed snapshot. We show this in Listings 4.3 and 4.2.

4.1.4 Enforcing the visibility premise

The visibility premise of COMMIT $\nexists T \in \mathcal{T}_c \cup \mathcal{T}_r : \texttt{ct} < T.\texttt{dt}$ forbids to read from a non-committed transaction.

We can re-write it to a simpler form:

$$\begin{aligned} \nexists T \in \mathcal{T}_c \cup \mathcal{T}_r : \texttt{ct} < T.\texttt{dt} &\equiv \forall \texttt{t} \in \Pi_{\texttt{dt}}(\mathcal{T}_c \cup \mathcal{T}_r) : \texttt{ct} \geqq \texttt{t} \\ &\equiv \texttt{ct} \geqq \max_{\mathcal{T}_c \cup \mathcal{T}_r}(\texttt{dt}) \end{aligned}$$

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 finalized its transition to committed. This requires synchronization between concurrent transactions. We implement it as follows.

To enforce this premise in the implementation, the timestamp generator described in Section 4.1.1, 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 `dt`such as `dt` $\leqq max\_allowed\_dt$. A transaction requests a new commit timestamp `ct` $\geqq min\_allowed\_ct$ from the timestamp server, at the latest during the transition from running to committed (in rule COMMIT). The request of `ct` is a *getAndIncrement* operation on $min\_allowed\_ct$. It is added to the list of pending commit timestamps and then returns the `ct`. After the coordinator's call to doCommit returns, the coordinator notifies the timestamp server. If there are any in-flight transactions with a lower timestamp, the server adds the `ct` to a list of pending commit timestamps. Otherwise, the server atomically advance $max\_allowed\_dt$ up to being equal to $min\_allowed\_ct$; this makes this transaction visible to future transactions.

```
1 public synchronized void endCommitNotify(int ct) {
2   commitRunning.remove(ct);
3   if (commitRunning.isEmpty()) {
4       max_allowed_dt.set(ct+1);
5   } else {
6       max_allowed_dt.set(commitRunning.first());
7   }
8 }
```

Listing 4.2: endCommitNotify

4.1.5 Implementing the Transaction Coordinator

The transaction coordinator is an entity located between clients and the store. It implements the semantics described in Figure 3.1. Our implementation directly translates the specification, viewed as pseudocode, into sequential Java code, without any optimization. Each premise in the specification corresponds to an assertion in the Java code. A coordinator handles a single transaction at a time, maintaining the state (buffer and read- and dirty-sets) private to the coordinator.

Coordinators for different transactions run in parallel. As they concurrently access the store (or multiple stores), and we describe the synchronization to avoid data races below, under the corresponding store variant. A coordinator uses the Timestamp Generator to keep track of running and committed transactions. Lastly, synchronization is necessary to enforce the visibility premise, as described in the previous section.

To ensure deterministic comparison of results between stores, a coordinator can call any of our stores, or multiple stores simultaneously, with the same arguments (transaction identifier, dependency timestamp, commit timestamp, etc.).

The `Coordinator` class implements each rule in a corresponding method:

- `begin`: Starts a new transaction with a dependency timestamp. There exist in the implementation a version that does not take a dependency timestamp as an argument. It ask the Timestamp Generator for $max_allowed_dt$and use it to call the version that takes a dependency timestamp as an argument.
- `read`: Reads a value from a given key in the store. Call the doCommit method of the store.
- `init_key`: Initializes a key in the transaction's state buffer if not already present before a read or write.
- `update`: Updates the value associated with a given key in the store. Calls the doUpdate method of the store.
- `commit`: Attempts to commit the current transaction. If it can, it calls the doCommit method of the store otherwise calls doAbort.
- `abort`: Aborts the current transaction. Calls the doAbort method of the store.

The coordinator is agnostic to the underlying store implementation. It calls the corresponding methods of the store to perform the requested actions even if the method is a no-op according to the store semantics.

The `Coordinator` class also contains a `run` method that listens for client requests and performs the requested actions based on the input. This method handles communication with clients with input and output streams, processes a client requests, and dispatches to the appropriate method.

To demonstrate that the Java implementation of the transaction coordinator adheres to the specification invariants, we can look at the code snippets containing assertions. These assertions are in line with the premises defined in the specification.

1. In the `begin` method:

```
1 public void begin(Timestamp dependency) throws IOException {
2   try {
3       assert (!transactionInProgress);
4       // Check if the dependency is valid
```

```
        // isDependencyValid returns true if dependency <=
           max_allowed_dt
        assert (dependency != null && !timestampGenerator.
           isDependencyValid(dependency));

        transaction = new Transaction(dependency);
        transactionInProgress = true;
        store.doBegin(transaction);
        if (out != null)
            write(Integer.toString(transaction.getDt().
               getTimestamp()));
    } catch (Exception e) {...}
}
```

Listing 4.3: beginTransaction

The assertion checks that there is no transaction currently in progress before starting a new one.

If specified, the dependency timestamp is compared against the value returned by the timestamp generator's `peek` method.

2. In the `read` method:

```
public Value<?> read(Key key) throws IOException {
    assert (transactionInProgress && transaction != null);
    if (transaction.getStateBuffer().containsKey(key)
            && transaction.getStateBuffer().get(key) != null) {
        write((String) transaction.getStateBuffer().get(key).
           get());
    } else {
        init_key(key);
        if (transaction.getStateBuffer().containsKey(key)) {
            if (transaction.getStateBuffer().get(key) != null)
               {
                Value<?> value = transaction.getStateBuffer().
                   get(key);
                if (value.get() instanceof String){
                    write((String) value.get());
                } else {
                    write(Integer.toString((Integer) value.get
                       ()));
                }
                return value;
            } else {
```

```
18                write("null");
19            }
20        } else {
21            assert false : "Key not found after initialization"
                  ;
22        }
23    }
24    if (out != null)
25        out.flush();
26    return null;
27 }
```

Listing 4.4: read

The assertion ensures that a transaction is in progress and that the transaction object is not null before performing a read operation. As per the specification, the read operation first checks if the key is present in the transaction's state buffer and returns the value if it is. Otherwise, it calls INIT_KEY to initialize the key in the state buffer.

3. In the `init_key` method:

```
1 assert (!transaction.getStateBuffer().containsKey(key));
```

Listing 4.5: init_key

This assertion checks that the key is not already present in the transaction's state buffer before initializing it.

4. In the `update` method:

```
1 public void update(Key key, Value<?> value) {
2   assert (transactionInProgress && transaction != null);
3   if (!transaction.getStateBuffer().containsKey(key)) {
4       init_key(key);
5   }
6   store.doUpdate(transaction, key, value);
7   transaction.addUpdate(key, value);
8 }
```

Listing 4.6: update

Similar to the read operation, this assertion ensures that a transaction is in progress and that the transaction object is not null before performing an update operation. As per the specification, it first checks if the key is present in the

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

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

Fig. 4.1.: Operations of map store

transaction's state buffer before calling doUpdate. Otherwise, it calls INIT_KEY to initialize the key in the state buffer.

5. In the `commit` method:

```
1  public void commit() {
2      assert (transactionInProgress && transaction != null);
3      Timestamp commit = new Timestamp(timestampGenerator.
           next());
4      assert (commit.getTimestamp() >= transaction.getDt().
           getTimestamp());
5      store.doCommit(transaction, commit);
6      timestampGenerator.endCommitNotify(commit.getTimestamp
           ());
7      transactionInProgress = false;
8      transaction = null;
9  }
```

Listing 4.7: commit

The first assertion checks that a transaction is in progress and that the transaction object is not null before committing. The second assertion ensures that the commit timestamp is greater than or equal to the dependency timestamp.

6. In the `abort` method:

```
1  assert (transactionInProgress && transaction != null);
```

Listing 4.8: abort

Similar to the commit operation, this assertion ensures that a transaction is in progress and that the transaction object is not null before aborting.

Building on the Coordinator we have implemented two stores that implement the `Store` interface.

$$
\begin{aligned}
\mathsf{lookup}(\sigma, k, \mathtt{dt}) &= \bigsqcup(\mathsf{committed\_before}(\sigma, k, \mathtt{dt})) \\
\mathsf{doUpdate}(\sigma, \tau, k, \delta) &= \sigma \rhd (\mathsf{update}, \tau, k, \delta) \\
\mathsf{doCommit}(\sigma, \tau, \_, \_, \_, \mathtt{ct}) &= \sigma \rhd (\mathsf{commit}, \tau, \mathtt{ct})
\end{aligned}
$$

with the following notation: 1. σ is a journal-based store; 2. $\rhd$ represents concatenation;
3. 'update' and 'commit' are tags to distinguish the type of a journal record.
4. $\mathsf{committed\_before}(\sigma, k, \mathtt{dt})$ denotes the subsequence of journal σ, of records tagged with key k, that have a commit timestamp less or equal to $\mathtt{dt}$; formally:

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

Fig. 4.2.: Operations of Journal store

4.2 Basic Variants

In this section, we present two basic variants of transactional storage systems, each with their unique advantages and characteristics. These variants are the map-based store and the journal-based store.

4.2.1 Map store semantics

Our first variant is the random-access *map-based store*, located either in memory or on disk. A map store is restricted to contain only assignments.

As illustrated informally in Figure 5.1(c), a map store can be abstracted as an infinite matrix, with a row per key and a column per time unit.[1] 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 4.1. 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 transaction's dependency timestamp $\mathtt{dt}$. If there are multiple concurrent most-recent versions, it merges them.
- Command doUpdate is a no-op for this variant.

[1] 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. (Section 6.1).

- When a transaction commits with timestamp `ct`, doCommit eagerly copies its update to every dirty key k, from the state buffer into coordinates $(k, \texttt{ct})$ of the matrix.[2]

We made some semantic choices in this variant that are directly related to the implementation. Command doUpdate being a no-op is not necessary here. One possible version of the map store semantics would have doUpdate add the update to the store. The doCommit command would then be responsible for updating every update in the store with the selected commit timestamp. However, if every transaction would publish its updates to the store with a `ct`set to $\perp$, we need to differentiate them by adding the transaction identifier in the metadata. Additionally another downside of this approach is that for multiple updates to the same key, we would write an equivalent number of updates to the store or delete previous uncommitted versions of the key.

We described in Section4.1.4 how we ensure that uncommitted update are invisible to other transactions.

4.2.2 Map store implementation

The in-memory implementation uses a standard concurrent Java hashmap (ConcurrentHashMap) to map keys to a CopyOnWriteArrayList of timestamped versions. CopyOnWriteArrayList is a thread-safe variant of ArrayList that allows concurrent reads and writes. This ensures that every transaction execute using its own private copy of the CopyOnWriteArrayList without being affected by concurrent updates.

The lookup algorithm retrieves the visible value of a key for a given snapshot timestamp. It is designed to work efficiently with the Multi-Version Concurrency Control (MVCC) approach used in the map store. Here is a step-by-step explanation of the lookup algorithm:

1. Retrieve the list of timestamped values for the given key.
2. Sort the list based on their timestamps.
3. Iterate through the sorted list to find the most recent value whose timestamp is less than or equal to the given snapshot timestamp. Timestamps being

[2] Strictly speaking, one could copy all the keys. However, this optimisation of copying only the dirty keys is justified in practice, as the space of keys is essentially unbounded (e.g., if keys are arbitrary strings).

scalar values there is a total order on them. In the future, in the presence of concurrent updates a merge operation is performed.

4. Return the found value as the visible value for the key at the given snapshot timestamp.

The following code snippet shows the lookup algorithm:

```
public Value<?> lookup(Key key, Timestamp timestamp) {
    CopyOnWriteArrayList<Value<?>> values = store.
        getOrDefault(key, null);
    if (values != null) {
        values.sort(Comparator.comparing(Value::getTimestamp)
            );
        Value<?> result = null;
        result = values.get(values.size() - 1);
        if (result.getTimestamp().compareTo(timestamp) < 0) {
            return result;
        }
        for (Value<?> v : values) {
            if (v.getTimestamp().compareTo(timestamp) <= 0) {
                result = v;
            } else {
                break;
            }
        }
        return result;
    } else {
        return null;
    }
}
```

Listing 4.9: Map store lookup algorithm

The doCommit method checks for the existence of a CopyOnWriteArrayList associated with a given key, creates a new one if needed, and adds the new value. If it does, the method adds the new value to the existing CopyOnWriteArrayList. If not, it creates a new CopyOnWriteArrayList, adds the new value to it, and then associates the new ArrayList with the given key in the store.

The following code snippet demonstrates this process:

```
public synchronized void doCommit(Transaction transaction,
    Timestamp commit) {
```

```
2       HashMap<Key, Value<?>> stateBuffer = transaction.
            getStateBuffer();
3       Set<Key> keys = transaction.getDirtySet();
4       for (Key key : keys) {
5           Value<?> value = stateBuffer.get(key);
6           value.setTimestamp(commit);
7           CopyOnWriteArrayList<Value<?>> values = store.
                getOrDefault(key, null);
8           if (values != null) {
9               store.get(key).add(value);
10          } else {
11              values = new CopyOnWriteArrayList<>();
12              values.add(value);
13              store.put(key, values);
14          }
15      }
16  }
```

Listing 4.10: Map store commit algorithm

There is a potential issue that can occur with concurrent transactions: if two transactions are committing at the same time, they might attempt to create a new CopyOnWriteArrayList for the same key simultaneously. In this scenario, one of the CopyOnWriteArrayLists would be added to the store while the other would be discarded, causing the loss of some updates. To avoid this, the doCommit method is synchronized, i.a. only one transaction a time can call doCommit.

The doUpdate method is no-op, as the updates are not immediately applied to the map store. Instead, recall that the coordinator adds them to the *stateBuffer* and updates the *dirtySet*. The updates are applied to the store in doCommit. As discussed in the previous Section4.2.1 doUpdate being a no-op, simplifies the implementation and provides an efficient way to handle updates without affecting the map store's state during the transaction.

In an implementation without the Timestamp Generator, persisting the last commit timestamp would be useful as it would allow the store to avoid seeking through the entire store to find the most recent commit timestamp. This can be done by writing the commit timestamp to one of two on-disk locations, alternating between the two, and assuming that such a write is atomic and each location is monotonically increasing. When the store restarts after a power-down, it takes the highest of the two.

4.2.3 Journal store semantics

Our second variant is the journal-based store. In contrast to the map store, 1. a journal is accessed sequentially, 2. it contains general effects (not just assignments), and 3. it materialises values lazily on lookup.[3] Figure 5.1(b) provides an informal illustration.[4]

Figure 4.2 gives the formal semantics of a journal store. Function doUpdate appends an update record to the journal, with 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 any effects that precede the most recent assignment, and merging concurrent effects).

4.2.4 Journal store implementation

We provide an in-memory and an on-disk implementation of the journal. We make the following assumptions of the disk. Writes are sequentially ordered; 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 throughput, 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 optimizations, and checking premises with assertions. The in-memory implementation is an append-only data structure,

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

containing a sequence of records; the on-disk journal writes its records to a sequential file. There are three main types of records:

- 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.

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. After a crash if a commit record is found, we know all the update records preceding it are persisted.

```
1 public synchronized void doCommit(Transaction transaction,
    Timestamp commit) {
2   writeRecord(new Record(transaction.getTid(), commit, Record
      .Type.COMMIT));
3   flush();
4   super.doCommit(transaction, commit);
5 }
```

Listing 4.11: doCommit

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 materialize any version required, without any concurrency issues.

The records from multiple transactions may be interleaved in the journal. 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, i.e. transactions that are missing a commit or abort record, with an abort record, and compares the highest commit timestamp found with the timestamp server's current value. If the timestamp server's value is higher, we assume that a running transaction was interrupted by the crash, and revert the timestamp server's value to the highest commit timestamp found in the journal.

4.2.5 Discussion about design choices and performance

Timestamp Generator

We acknowledge that using a Timestamp Generator adds a point of failure to the system. Handling critical information such as $max\_allowed\_dt$ and $min\_allowed\_ct$ in a single separate component adds complexity to the system. However, one of our goals is to compare multiple stores to one another. Handling multiple timestamps in each would mean that during testing the stores would have to be aware of each other and require a certain amount of coordination to ensure that the *histories* of each store are equivalent.

The Journal store as it is implemented can compute using the timestamps in the journal the $max\_allowed\_dt$ and $min\_allowed\_ct$. For the Map store, we use discuss at the end the Section4.2.1 how to persist the timestamps in TimestampGeneratorless design.

Optimisation

We presented a number of design choices that can be adjusted to provide better performance. For the sake of simplicity, we privileged synchronization of methods like doCommit when fine-tune locks would lead to better performance. For the Journal we have a single writer thread, which is not optimal for performance. We could use multiple journal files, each managed by a dedicated writer thread. This approach would allow use to maintain the sequential nature of write. However, it requires careful management of the order in which these journal files are written and read to ensure data consistency.

Sharding and replication

One obvious way to improve the performance of both stores is to shard the data or to replicate the stores. These present a number of challenges, such as ensuring that the timestamps are consistent across the shards or replicas. We discuss these issues in Section8.1.

Using off the shelf components and libraries

In the context of this thesis, we chose to implement the stores from scratch. However, in a real-world scenario, we would probably use off-the-shelf components and libraries. One obvious example would be to combine multiple stores into a single store in order to leverage their strengths. We call it composing stores and discuss it in the next Chapter5.

5 Composing Stores

5.1 Composing Stores

The basic store variants of the previous section are simple but have performance issues, e.g., they grow without bound. Modern stores improve performance with 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 5.3.

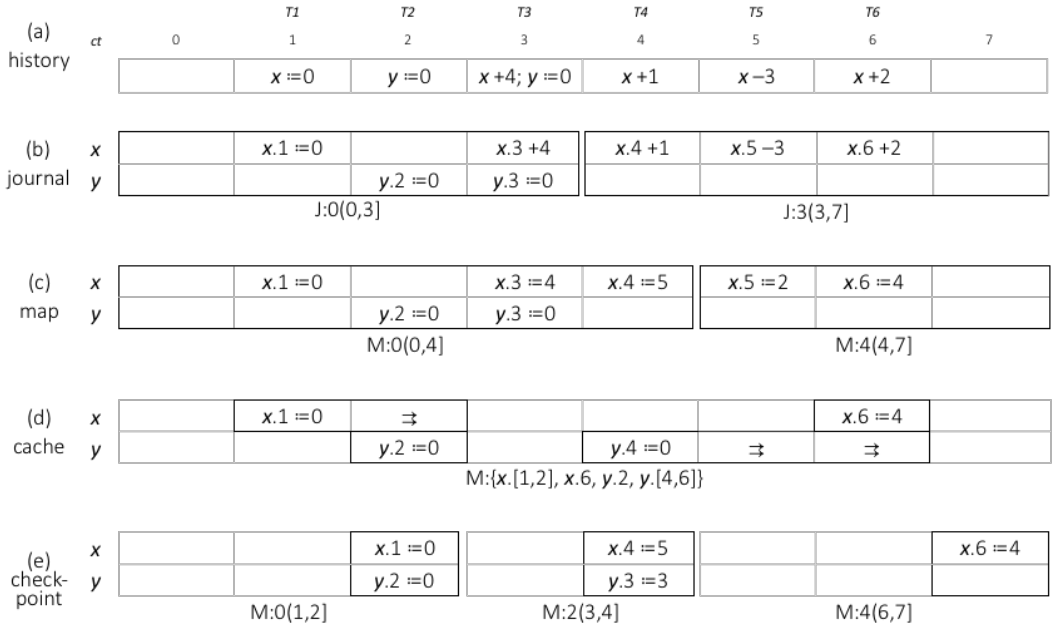

Fig. 5.1.: Store variants and store composition. Subfigure (a) represents a history; the following ones different variants and ranges. To simplify the figure, timestamps are assumed integer (scalar), and the history is represented as a sequence of transactions with strictly monotonic dependency and commit 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. $\rightrightarrows$ extends validity of a cache entry; in $J{:}1{:}(2,3]$: J = journal/M = map, $low\_history$=1, domain $(low\_lookup = 2, high\_lookup = 3]$.

$$
\begin{aligned}
\mathsf{lookup}(\sigma, k, \mathtt{dt}) &= \begin{cases} \text{as in Fig. 4.1 or 4.2} & \text{if } (k,\mathtt{t}) \in \mathcal{D}_\sigma \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{doUpdate}(\sigma, \tau, k, \delta) &= \begin{cases} \text{as in Fig. 4.1 or 4.2} & \text{if } (k,\mathtt{t}) \in \mathcal{D}_\sigma \\ \sigma & \text{otherwise} \end{cases} \\
\mathsf{doCommit}(\sigma, \tau, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct}) &= \begin{cases} \text{as in Fig. 4.1 or 4.2} & \text{if } (k,\mathtt{t}) \in \mathcal{D}_\sigma \\ \sigma & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 5.2.: Store σ with domain $\mathcal{D}_\sigma$

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

Fig. 5.3.: Operations of composed store

ADD_MINISTORE

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

REMOVE_MINISTORE

$$
\frac{\begin{array}{ccc}\sigma = \{\sigma_0, \sigma_1, \sigma_2, \dots\} & & \mathcal{D}_\sigma = \mathcal{D}_{\sigma_0} \cup \mathcal{D}_{\sigma_1} \cup \mathcal{D}_{\sigma_2} \cup \dots \\ \sigma' = \sigma \setminus \sigma_0 & \mathcal{D}_{\sigma'} = \mathcal{D}_{\sigma_1} \cup \mathcal{D}_{\sigma_2} \cup \dots & \forall (k,\mathtt{t}) \in \mathcal{D}_{\sigma'} : \mathsf{lookup}(\sigma', k, \mathtt{t}) = \mathsf{lookup}(\sigma, k, \mathtt{t})\end{array}}{(\sigma, \mathcal{F}_\sigma, \mathcal{T}_\mathsf{a}, \mathcal{T}_\mathsf{c}, \mathcal{T}_\mathsf{r}) \xrightarrow{\mathsf{rmvmini}(\sigma, \mathcal{F}_\sigma, \sigma_0, \mathcal{F}_{\sigma_0})} (\sigma', \mathcal{F}_{\sigma'}, \mathcal{T}_\mathsf{a}, \mathcal{T}_\mathsf{c}, \mathcal{T}_\mathsf{r})}
$$

Fig. 5.4.: Operational semantics of store composition. σ is the composition of ministores σ_1, σ_2, ... $\mathcal{D}_\sigma$ is the domain component of $\mathcal{F}_\sigma$.

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 5.1(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 5.1(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 5.1(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 [O'N+96] 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 [Akk+16] by composition.

5.1.1 Field and domain of a store

We associate store σ with auxiliary information, its *field* $\mathcal{F}_\sigma$, 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 5.2.

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 that partition 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 $(low\_history, high\_lookup]$. A *checkpoint* $\sigma_{t_0} = C{:}0{:}(t_0 - 1, t_0]$ is a map store containing these values at time t_0: $\sigma_{t_0} \triangleq \{\mathtt{t} \mapsto \mathsf{safe}(\Theta, k, t_0)\}$. Then one can compute any later version of the key from the checkpoint and its later updates: $\mathsf{safe}(\Theta, k, \mathtt{t}) = \mathsf{lookup}(\sigma_{t_0} \sqcup \bigsqcup\{\delta_k^{t'} : \delta_k^{t'} \in \Theta \wedge t_0 < t' < \mathtt{t}\}$.

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 $\mathsf{lookup}(\sigma, k, \mathtt{t})$ is significant only if $20 < \mathtt{t} \leq 30$. Figure 5.1(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 $high\_lookup$).

5.1.2 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 5.3, 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 5.1 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,[1] 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 5.1, composing $M{:}0{:}(0,4]$ with checkpoint $M{:}4{:}(6,7]$ behaves like $M{:}0{:}(6,7]$.

Formally, if $HL_1 \not\geq LL_2 \wedge HL_1 \geq LH_2$, then the composition of $M{:}LH_1{:}(LL_1, HL_1]$ with $M{:}LH_2{:}(LL_2, HL_2]$ behaves like $M{:}LH_1{:}(LL_2, HL_2]$.

5.1.3 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, 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 5.4 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 $\mathcal{D}_{\sigma_0}$.

[1] 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.

Recall that, for any query $\mathsf{lookup}(\sigma, k, \mathtt{t})$, any store σ that is produced by applying the transaction semantics (Figure 3.1) 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 5.4. 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.

5.1.4 Total store

A total store is a segment store that represents the whole history, i.e., with $low\_history = 0$. To capture this, we add to the rules of Figure 5.4 the following for a total store: • A total store is created as a segment store with $low\_history = 0$. • In REMOVE_MINISTORE, the result domain $\mathcal{D}_{\sigma'}$ must be a segment store that satisfies $low\_history = 0$.

Removing a ministore from the total store should not interfere with the reads of a running transaction. Therefore, in Figure 3.1, we add the following premise to COMMIT for a total store: $\sigma.low\_lookup < \mathtt{dt}$.

5.1.5 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 $M''{:}0{:}(100, 200]$.

6 Conductor

6.1 Implementing a Write-ahead Log by Composition

To manage composition, we introduce a "conductor" store.[1] In order to compose dynamically, the conductor supports the addmini, rmvmini interface of Figure 5.4. 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. For each store the conductor maintains the list of *highLookups*, *lowLookups* and *lowHistories*. As any store, the conductor also supports the lookup, doUpdate, doCommit interface, which recurse through the ministores as specified in Figure 5.3, but follows the topology.

The addmini and rmvmini operations are implemented as *addSore* and *removeStore* operations in the conductor. The addStore is pretty straight forward, it just adds the ministore to the topology and initializes the domain of the ministore to the highest commit timestamp. The domain of the new store is]highlookup, highlookup]. Making it unqueryable until a transaction commits to this store.

The addmini operation is as follows:

```
1 public void addStore(KeyValueStore store) {
2   Timestamp initTimestamp = generalHighLookup;
3   lowHistories.add(initTimestamp);
4   lowLookups.add(initTimestamp);
5   highLookups.add(initTimestamp);
6   stores.add(store);
7 }
```

Listing 6.1: addStore operation

The removeStore operation is trickier. We implemented a naive version where a store σ_a is removed from the composition, if there exist a store σ_b such that the domain $\mathcal{D}_{\sigma a} \subseteq \mathcal{D}_{\sigma b}$. The ideal algorithm must check if there is a combination of store that covers the domain of the store to be removed. The removeStore operation is as follows:

[1] Unfortunately mixing musical metaphors.

```
public boolean removeStore(int index) {
      if (index >= 0 && index < stores.size()) {
          for (int i = 0; i < stores.size(); i++) {
              if (i != index && lowHistories.get(i).compareTo
                  (lowHistories.get(index)) <= 0 &&
                       highLookups.get(i).compareTo(
                           highLookups.get(index)) >= 0) {
                   stores.remove(index);
                   lowHistories.remove(index);
                   lowLookups.remove(index);
                   highLookups.remove(index);
                   return true;
              }
          }
      }
      return false;
  }
```

Listing 6.2: removeStore operation

The lookup algorithm is straightforward, 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-$\perp$. If no ministore returns a value, the conductor returns $\perp$. Function doUpdate applies to all ministores in range, but this generally reduces to the top-layer journal.

The lookup method is as follows:

```
public Value<?> lookup(Key key, Timestamp timestamp) {
    Value<?> result = null;
    for (int i = 0; i < stores.size(); i++) {
        if (timestamp.compareTo(lowLookups.get(i)) >= 0 &&
            timestamp.compareTo(highLookups.get(i)) <= 0) {
             result = stores.get(i).lookup(key, timestamp);
             if (result != null) {
                 break;
             }
        }
    }
    return result;
}
```

Listing 6.3: lookup operation

The preferred topology (ignoring caches) has a journal at the top layer, to leverage its write throughput and fault tolerance properties. Therefore in our implementation, the conductor chooses the top layer to write new transactions.

6.1.1 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 4.2.4 that this is atomic), the conductor identifies (if any) the first running transaction τ within M' that terminates beyond M', i.e., such that $\tau.\texttt{dt} \leq M'.HL' < \tau.\texttt{ct}$. 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 $LL' \leqq \tau.\texttt{dt}$. Following this ordering ensures that truncation remains transparent and crash-tolerant.

In our implementation, the journal has a ReadWriteLock to avoid concurrent checkpointing and truncation. CheckpointBegin writes a record to the journal, and materialize all update between LH and the specified timestamp. Then returns the values that are part of the checkpoint. CheckpointEnd writes a record to the journal. Finally, a Truncate function advances the LL of the journal and removes the records that are no longer needed.

```
public ConcurrentHashMap<Key, Value<?>> checkpointBegin(
    Timestamp checkpointTime) {
     return checkpointBegin(getlowLookup(), checkpointTime);
}

public ConcurrentHashMap<Key, Value<?>> checkpointBegin(
    Timestamp lowWatermark, Timestamp checkpointTime) {
     assert this.lowHistory.compareTo(checkpointTime) <= 0;
     ConcurrentHashMap<Key, Value<?>> updates = new
        ConcurrentHashMap<>();
     addRecord(new Record(checkpointTime, Record.Type.
        CHECKPOINTBEGIN));
     try {
          lowWatermarkReadLock.lock();
          int beginIndex = (lowWatermark.compareTo(this.
             lowHistory) == 0) ? lowWatermarkRecordId :
             getBeginIndex(lowWatermark);
          int commitIndex = getCommitRecordIdIndex(
             checkpointTime);
          if (beginIndex == -1 || commitIndex == -1) {
               throw new RuntimeException("No begin or commit
                  record found with the given timestamps");
          }
          for (int i = beginIndex; i <= commitIndex; i++) {
               Record record = journal.get(i);
               if (record.getType() == Record.Type.EFFECT) {
                    if (record.getDependency().compareTo(
                       lowWatermark) >= 0) {
                         updates.put(record.getKey(), record.
                            getValue());
                    }
               }
          }
          return updates;
     } finally {
          lowWatermarkReadLock.unlock();
     }
}
```

Listing 6.4: checkpointBegin

```
public void checkpointEnd(Timestamp LowWatermark) {
  addRecord(new Record(LowWatermark, Record.Type.
     CHECKPOINTEND));
}
```

Listing 6.5: checkpointEnd

```
public void truncate(Timestamp lowWatermark) {
    try {
        lowWatermarkWriteLock.lock();
        this.lowLookup = lowWatermark;
        int recordIdOfCommit = -1;
        for (int i = lowWatermarkRecordId; i < journal.size()
            ; i++) {
            Record record = journal.get(i);
            if (record.getType() == Record.Type.COMMIT) {
                if (record.getCommitTime().compareTo(
                    lowWatermark) == 0) {
                    recordIdOfCommit = record.getRecordId();
                    break;
                }
            }
        }
        if (recordIdOfCommit == -1) {
            throw new RuntimeException("No commit record
               found with commit time " + lowWatermark);
        }
        // Once we have the recordId of the commit record
        // with the low watermark commit time,
        // we need to find the first begin record with a
        // recordId lower than the recordId of the commit
           record
        // that has a commit record with a recordId higher
           than
        // the recordId of the commit record.
        ArrayList<Record> beginRecords = new ArrayList<>();
        ArrayList<Record> commitRecords = new ArrayList<>();
        System.out.println("Store " + journal.toString());
        for (int i = lowWatermarkRecordId; i <=
           recordIdOfCommit; i++) {
            System.out.println("i = " + i + " journal.size()
               = " + journal.size() + " recordIdOfCommit = "
                + recordIdOfCommit+ " lowWatermarkRecordId =
                " + lowWatermarkRecordId);
```

```
29              Record record = journal.get(i);
30              if (record.getType() == Record.Type.BEGIN) {
31                  beginRecords.add(record);
32              } else if (record.getType() == Record.Type.COMMIT
                    ) {
33                  commitRecords.add(record);
34              }
35          }
36          // Remove all the begin records from beginRecords
37          // that do not have a commit record with the same
38          // transaction id in the commitRecords list.
39          // Meaning running transactions that have not
                committed yet
40          // are not removed.
41          for (int i = 0; i < beginRecords.size(); i++) {
42              Record beginRecord = beginRecords.get(i);
43              boolean found = false;
44              for (Record commitRecord : commitRecords) {
45                  if (beginRecord.getTrId().equals(commitRecord
                        .getTrId())) {
46                      found = true;
47                      break;
48                  }
49              }
50              if (!found) {
51                  beginRecords.remove(i);
52                  i--;
53              }
54          }
55          // find the begin record with the lowest record id.
56          int lowestRecordId = Integer.MAX_VALUE;
57          for (Record beginRecord: beginRecords) {
58              if (beginRecord.getRecordId() < lowestRecordId) {
59                  lowestRecordId = beginRecord.getRecordId();
60              }
61          }
62          lowWatermarkRecordId = lowestRecordId;
63
64      } finally {
65          if (lowWatermarkWriteLock.isHeldByCurrentThread()) {
66              lowWatermarkWriteLock.unlock();
67          }
68      }
69  }
```

Listing 6.6: truncate

The WAL recovers as follows. The journal recovers as in Section 4.2.4. If it contains a CheckpointBegin record without the matching CheckpointEnd (a checkpoint might have not terminated), recovery restarts the checkpoint from the beginning using 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-initializes the domain of the checkpoint and the journal from the information stored in the journal.

Part III

Experimental Evaluation

7 Experimental evaluation

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. After some experiment on multiple machines, we ran our experiments 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. Our workload is I/O intensive and this machine had the best I/O performance. Run-time assertion checking is enabled.

7.0.1 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 7.1 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. As the journal grows in size, the read performance degrades degrades and with the INIT_KEY rule, the journal is forced to read before persisting a write operation to disk. Our crash-resistant WAL implementation (marked "Conductor" in the figure) suffers from the same issue as the Journal is the prefered layer for writing in our implementation. An obvious solution will be to compose the WAL with a cache for absorbing reads, and to allow blind writes.

7.0.2 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.

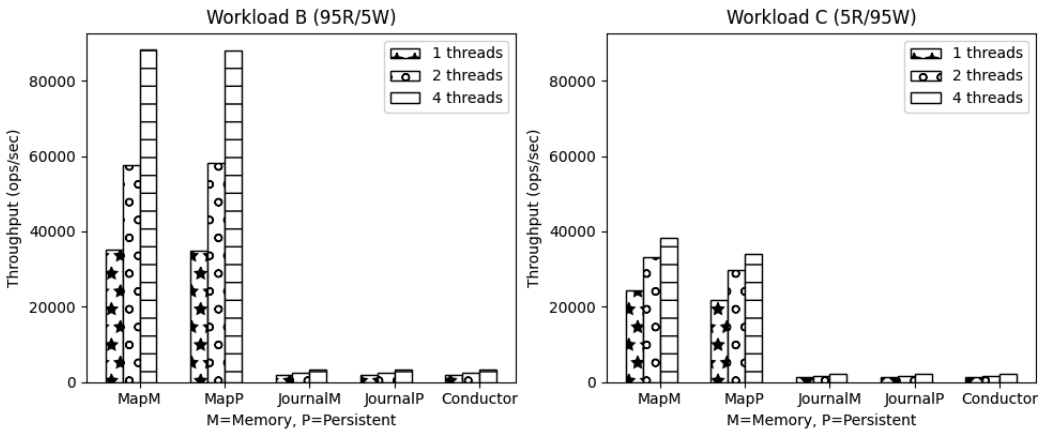

Fig. 7.1.: Average throughput of the different stores

Our second objective was to validate experimentally the theoretical result that stores are behaviourally equivalent. To this effect, we implemented a variant of the coordinator called emphMultiCoordinator. Like the Conductor the MultiCoordinator, has list of stores, and it executes the same transactions on all of them. For every read the MultiCoordinator waits for a response from all stores, and then compares the return values, see Listing 4.5. If there are any differences, it throws an exception.

```
1   public void init_key(Key key) {
2     assert (!transaction.getStateBuffer().containsKey(key));
3     ArrayList<Value<?>> values = new ArrayList<>();
4     // Read from each store and ensure that the values are
          the same
5     for (KeyValueStore store : listOfStore) {
6         Value<?> value = store.lookup(key, transaction.getDt
              ());
7         values.add(value);
8     }
9     if (values.size() > 1) {
10        for (int i = 1; i < values.size(); i++) {
11            if (values.get(i) == null && values.get(i-1) ==
                  null) {
12                continue;
13            } else if (values.get(i) == null || values.get(i
                  -1) == null) {
14                System.out.println(values.get(i-1) + " " +
                      values.get(i));
15                assert (false);
16            } else {
```

```
17                  assert(values.get(i).isEquals(values.get(i-1)
                        ));
18              }
19          }
20      }
21      transaction.addValueToStateBuffer(key, values.get(0));
22  }
```

Listing 7.1: MultiCoordinator, Init_Key

Thanks to early tests using this experiment, we were able to find several small divergence bugs in our implementation of the journal-store and the map-based store.

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.

7.0.3 Code coverage

During our development, we tried to maximize code coverage. The stores are still in active development, and we are still adding tests. Code coverage is not a good metric for correctness, but it is a good indicator of the testing effort. Java is a verbose language making it hard to cover we do not cover all the setter and getter methods, as well as the exception handling. In the current state of the code we show that we have decent coverage, as shown in Table 7.1.

7.0.4 Lessons learned

Although we strived to translate the specification verbatim, the asserts were invaluable for detecting bugs early. Consider for instance enforcing the visibility premise (Section 4.1.4). Our initial implementation used a single atomic counter to generate the commit timestamp and to assign a dependency to a transaction. While this appears correct in a sequential execution, assert checking identified a bug, whereby the dependency was assigned before a concurrent transaction had fully committed.

Tab. 7.1.: Code coverage

Element	Method, %	Line, %
Key	100% (5/5)	100%
Record	100% (18/18)	95%
Timestamp	83% (5/6)	46%
Transaction	100% (8/8)	75%
Conductor	61% (13/21)	50%
LWW	100% (4/4)	83%
PNCounter	60% (6/10)	78%
JournalMemory	77% (17/22)	73%
Journal Persistent	100 (11/11)	61%
MapMemory	88% (8/9)	97%
MapPersistent	85% (6/7)	49%
TimestampGenerator	100% (12/12)	91%
Coordinator	76% (10/13)	40%
MultiCoordinator	100% (9/9)	100%

Hence our move to the current design with the two atomic counters $max\_allowed\_dt$ and $min\_allowed\_ct$. Following this change, we found two more cascading bugs.

The specification is invaluable to help reason about the behaviour. Direct translation of the transitions and the invariants in the code is a practical way to ensure correctness. In case the specification changes, the assertions are quick to catch any incorrect side effects of the changes, and help solve the resulting issues.

7.0.5 Note

More experiments are on the way, but we are currently changing the conductor implementation to support a cache. We are also planning on running the same YSCB benchmark on other databases to compare the performance of our WAL with other systems.

Part IV

Current and future work

8 Current and future work

8.1 Advancing the Conductor

The current implementation of the conductor is a Write-Ahead Log (WAL), which leaves room for enhancements and future research opportunities. This section discusses a few avenues that warrant further exploration.

8.1.1 Incorporating Cache for Improved Performance

Ongoing improvement to the conductor is the integration of a cache to enhance overall performance. The cache constitutes a map-store containing arbitrary key-value pairs, aimed at minimizing the overhead associated with frequently accessing the underlying storage layers.

8.1.2 Authorize blind

Another ongoing improvement to the conductor is the implementation of blindwrites to the journal. Write heavy performance is heavily impacted by the need to call Init_Key on the journal for every write.

8.1.3 Implementing an LSM-Tree through Composition

Following the successful integration of cache, the next stage of conductor development involves the implementation of an LSM-Tree using composition. The current conductor design does not perform periodic checkpointing and pruning of the journal. LSM-Tree-like compaction through composition would allow the conductor to periodically merge multiple stores.

8.1.4 Exploring Sharding Mechanisms through Composition

We believe the composition theory presented in this thesis offers a foundation for justifying sharding. In this thesis we have mainly explored composition through temporal domain using timestamps to define the domain of each ministore. Sharding is a way to partition the data domain and use it to provide additional mechanisms to improve performance and scalability.

8.1.5 Dynamic Addition and Removal of Ministores during Execution

Another research direction to be considered for the future development of the conductor is support for dynamic addition and removal of ministores. This capability will allow the conductor to dynamically scale. One example would be to add additional nodes for shading and using the different domain information to correctly reshard existing data.

8.1.6 Creating Adapters for Existing Databases

An interesting approach to composition would to create adapters for existing databases, enabling them to be incorporated into the composition framework. These adapters would act as an intermediary layer, exposing the store interface to the conductor. The adapter would be responsible for translating the store interface to the underlying database interface. This work is not trivial as it would require to compensate for missing features and consistency guarantees. For example, a database that does not support transactions would require the adapter to implement a transactional interface and ensure that transitions are visible atomically.

8.2 Formal Verification

As part of an ongoing collaboration, we aim to formally verify the equivalence between different store variants using Coq. Given any two implementations of the store, we will prove that they 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.

The proof goals for the composition include:

1. Show that when composing a journal and a checkpoint, where the lower bound of the journal overlaps the upper bound of the checkpoint, the resulting system is equivalent to the full store. This would confirm the correctness of the composition-based approach when merging journal and checkpoint data.
2. Prove that composing a cache with either a map or a journal results in a system equivalent to a cacheless one. This implies that the presence of a cache does not affect the system's correctness, and the cache only serves as an optimization mechanism.
3. Establish that checkpointing and truncation can be executed concurrently with transactions without compromising correctness. This would demonstrate the ability of the conductor to maintain data consistency in a concurrent environment.
4. Demonstrate that chunked (incremental) checkpoints are equivalent to non-chunked (monolithic) checkpoints in terms of functionality and correctness.

8.3 Checkpointing distributed database

In the beginning of this thesis we started working on a way to correctly prune AntidoteDB. Our goal was to define the portion of the history that can be safely removed.

8.3.1 AntidoteDB

AntidoteDB is a distributed Journal-store. AntidoteDB supports concurrent updates occurring in geo-distributed, highly-available Data centres (replicas); each DC originates its own set of updates. In turn, a DC is partitioned into shards, that are coordinated by Transaction Managers. The latter is responsible for assigning clients to transaction coordinators, which are responsible for executing transactions much like our implementation. Thus, in every shard the journal here is not a single sequential data structure, but is logically the union of a number of sequential *journal streams*, one per shard per DC. The originating shard is the single writer of a journal stream; all other replicas are readers. Furthermore, a stream originating in some shard in some DC is replicated to the same shard in all other DCs.[1]

Zooming into a given shard at a given DC, the former contains a *local* journal that contains the effects to this shard originating from the latter, and a set of remote journal streams, each one replicating the effects to this same shard at some other DC.

Another source of complexity is the Transactional Causal Consistency (TCC) model. Each DC is logically sequential, based on a variant of Snapshot Isolation. However, two DCs may update the database concurrently, and effects are related by the *happened-before* partial order (sometimes called causal order).

To keep track of happened-before, Antidote uses vector timestamps with one entry per DC. Every event in the database is tagged by the corresponding vector timestamp. A *consistent vector timestamp* (CVT) marks a *transactionally- and causally-consistent cut*. This means that, if a CVT contains some effect δ, then it also contains all effects in the same transaction as δ, as well as those that happened-before δ. The snapshot time and commit are different, they are represented by CVTs.

Recall that a given DC-shard has a local journal stream and one journal stream per remote DC. Similarly, a CVT has a timestamp entry per DC. We can now map each

[1] We ignore here the fact that a shard is itself replicated in its DC for fault tolerance, because this does not change the fact that each journal stream is sequential.

entry in a CVT (for some DC) to the prefix of the journal stream (of the same DC) that happened-before it. This cut forms a *transactionally- and causally-consistent snapshot*.

8.4 Consistent cuts of interest

The causal ordering of events implies that their vector timestamps are ordered in the same way: If Event 1 is causally-before Event 2, their vector timestamps *vt1* and *vt2* are such that $vc1 < vc2$. Note that the converse is not true in Antidote. The timestamps of two concurrent events may be either incomparable or arbitrarily ordered.

The order between vector timestamps *vt1* and *vt2* is defined as follows:

- $vt1 = vt2$ if every entry of *vt1* is equal to the corresponding entry of *vt2*. They represent the same event.
- $vt1 \leq vt2$ if every entry of *vt1* is less or equal to the corresponding entry of *vt2*.
- $vt1 < vt2$ if $vt1 \leq vt2$ and $vt1 \neq vt2$. Event 1 being causally before Event 2 implies $vt1 < vt2$, but the converse is not guaranteed.
- *vt1* is incomparable to *vt2* if $vt1 \neg \leq vt2 \wedge vt1 \neg \leq vt2$. There is some entry in *vt1* that is lower than the corresponding entry in *vt2*, and vice-versa. Events 1 and 2 are necessarily concurrent, but the converse is not true; that is, if two events are concurrent, this does not guarantee that their vector timestamps are incomparable.

A vector timestamp represents a *cut* or time of the data store. We are interested only in *consistent cut* as they have properties that are useful for maintaining information about the database.

8.4.1 Checkpoint Time (CT)

Every time a checkpoint is created, we persist a checkpoint record in the journal.

The variable *Checkpoint Time* designates the *oldest* available checkpoint, i.e., lowest *CVT* that includes, for every object, a checkpoint whose state includes all the update records committed at any time $\leq CheckpointTime$. State prior to *CT* cannot be safely recovered.

Checkpoint Time $= -\infty$, implies that the journal cannot be trimmed. Note that, while recovering from *CT* is safe, typically recovery will proceed from the *most recent* available checkpoint for performance reasons. To save space, a system typically stores only a few numbers of checkpoints, preferably only one. If the checkpoint store does not support versioning (i.e., each object has a single version), then there is a single checkpoint, identified by *Checkpoint Time*. For the rest of this section, we assume that the checkpoint store has a single version.

8.4.2 DC-Wide Causal Safe Point (DCSf)

Each shard in a DC is replicated to all other DCs. All effects that originate in some DC are sent asynchronously to the corresponding shard in other DCs. Although the shard-to-shard connection is FIFO, the storage state of different shards in the same DC is not causally consistent. Without extra care, a transaction that reads from multiple shards might be unsafe. To avoid this, a transaction should wait if a shard is missing effects with respect to another.

The Cure protocol [Akk+16] is what ensures the TCC properties. Cure has two main objectives:

1. Ensure that transactions in a DC commit atomically, and in a total order across all shards of that DC. It uses the Clock-SI design for this purpose [DEZ13].

2. Ensure that effects are observed in a causally-consistent order within a DC. To this effect, each shard continuously computes a safe lower bound for that DC, called its DC-Wide Causal Safe Point (DCSf).[2] The DCSf is a *CVT* across all shards of the DC that is *causally safe*, i.e., the corresponding updates, and their causal predecessors, have been received and persisted by all shards of this DC. States that are above the DCSf are not *visible*. A transaction whose snapshot time is not earlier than the DCSf must be blocked until the DCSf advances beyond it.

8.4.3 Global Causal Stable Point (GCSt)

In our new design, we will also leverage the concept of a *Global Causal Stable point* (GCSt). A GCSt is a state where all concurrent operations have been received and resolved. Formally, any updates that are delivered after the GCSt is computed

[2] Called the Global Stable Snapshot (GSS) in the Cure paper [Akk+16].

will have a higher timestamp than the GCSt [BAS17, Definition 5.1]. To simplify the logic, we assume that successive computations of a GCSt at some shard are monotonically non-decreasing (this is always possible). To compute the GCSt each shard shares its DCSf periodically with their counterparts in other DCs. The vague GCSt is computed as the lower bound of all known DCSf.

State that is earlier than the GCSt, can be stored using its sequential representation, avoiding any concurrency-related metadata such as vector clocks or tombstones. The transitions between successive GCSt's can be explained as sequential updates. This makes the representation simpler and more compact and enables the use of any sequential database as a checkpoint store. We leverage this fact by choosing checkpoint states to be earlier than GCSt whenever possible, this makes the sequentially consistent.

One issue with GCSt is that it makes progress only when every single DC is available. It stops advancing as soon as any single DC does not regularly communicate its metadata.

8.4.4 Min_dependency and Max_committed

In order to satisfy the properties of checkpoints, of the DCSf and the GCSt we keep track of all the dependencies of running transactions but also all the commit times of finished transactions.

Among those we single out those who are the most important.

min_dependency represents the oldest snapshot any running transaction is reading from. Because a checkpoints is sequentially consistent, there should be no in-flight transactions at Checkpoint-Time. To ensure that this property holds true min_dependency is used to track the point beyond which sequentiality is not guaranteed. When a transaction is finished, committed or aborted, the min_dependency advances to the next Dependency. One issue is that while a transaction is running, min_dependency will not advance, and consequently checkpointing will be paused.

max_committed is the last commit time recorded in the Journal. DCSf's advancement is bounded by max_committed, making sure that a new transaction does not read unsafe updates. Every time a transaction commits, its commit time become the new max_committed. Similarly to min_dependency, if no transaction commits, max_committed does not advances, nor does DCSf.

8.4.5 Low-Watermark and High-Watermark

To represent the persistent portion of the log we use Low-Watermark and High-Watermark. With *Low-Watermark* representing the lower bound and *High-Watermark* the higher bound. Records that precede *Low-Watermark* may be deleted and records that postdates *High-Watermark* might be volatile.

8.4.6 Invariants

Our goal is to have no perceivable loss of information, meaning that records that have not been checkpointed must not be deleted. Hence, our first invariant is $Low - Watermark \leq CheckpointTime$. Checkpoints should be sequentially consistent and stable across all DCs. Hence, the invariants $CheckpointTime \leq GCSt$ and $CheckpointTime \leq$ *min_dependency*. By construction the GCSt is computed as the lower bound of all known DCSfs across so this gives us the following invariant $GCSt \leq DCSf$. DCSf represents the point of safety in a DC using shared information between shards about their registered *commit times* therefore $DCSf \leq$ *max_committed*. These relations control the behaviour of each shard (*figure 8.1*).

Intuitively, using these relations we deduce the portion of the journal that is safe to be pruned, represented in orange in *figure 8.1*. It represents a CVT that is the lower bound of the GCSt and min_dependency. Meaning it only contains effects that are stable across all DCs and that have no concurrent transactions running. The next step of this work was proving that AntidoteDB with our new design was behaviourally equivalent to the original design. After trying to specify AntidoteDB we decided to start with a non distributed database which lead to the work presented in this thesis.

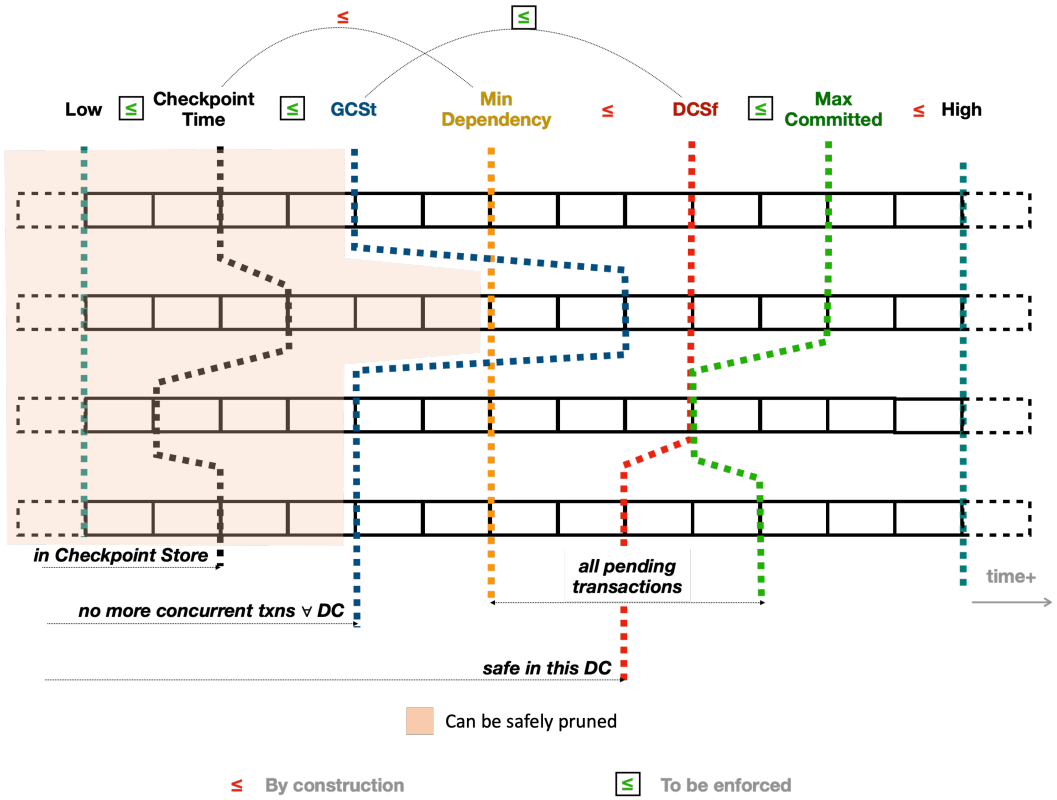

Fig. 8.1.: Relevant system states and their relations (for a given shard, at a given DC). Each horizontal tape (one per DC) depicts a journal stream for this shard. Each vertical line depicts a cut of interest and its vector timestamp. Over time, each journal stream grows to the right (and is trimmed to the left), and each state of interest advances monotonically to the right. As this happens, the causal precedence invariants, denoted $\leq$, must be maintained. To enforce $\leq$ requires synchronization between the corresponding processes.

Part V

Conclusion

9 Related Work

There is a large body of recent literature on specifying, verifying, and/or correctly implementing concurrent data structures and algorithms, including distributed systems and storage and file systems. As far as we are aware, none of the related work addresses the complexity of a modern storage backend.

In contrast to previous work, our focus is not to prove a particular algorithm, but to implement a correct system building on first principles. We show that a rigorous approach is not incompatible with performance.

Our approach of composing simple modules into a more complex structure is the standard approach in software engineering. In our case, each basic variant is sufficiently simple to be convinced of its correctness almost by inspection. We formalize the correctness rules for composition, which we leverage to conclude to correctness of the composed system. We believe this is novel in the case of a modern storage system design.

9.0.1 Formal specification of transactions and isolation models

Cerone et al. [CBG15] provide a detailed formalization framework for reasoning about the properties, invariants, and constraints of a concurrent transactional system. The focus of their work is to specify the different consistency/isolation models (e.g., serialisability, snapshot isolation, etc.) in a declarative fashion, and to prove their formalization equivalent to a more traditional operational specification. Our transaction semantics leverages their framework while simplifying it. Our current work focuses on causal consistency, but we believe we can extend it to stronger levels, by borrowing from their results. Their framework is monolithic, does not provide insight on the storage subsystem, and does not consider composition.

Sharing some similarities with the above, Crooks et al. [Cro+17] provide a state-based formalization of storage systems. They describe the storage system as a black box, aiming to characterize correct answers, whereas our approach is to derive an implementation.

9.0.2 Using lightweight formal methods to validate storage systems

Hance et al. [Han+23] applies lightweight formal methods to validate a new key-value storage node. Their approach uses formal specifications, which they translate into reference executables. Correcteness properties are extracted from the reference model and used to develop automated tools to test ongoing developement. In contrast, we start from a full formal specification of both transactions and the store. We implement them manually, paying special attention to efficient synchronization and crash atomicity. The use of property-based testing is an interesting avenue for future work.

9.0.3 Verified implementations

Chajed et al. [Cha+22] verify the crash safety of a file system implemented in Go against a formal specification. The implementation adds two-phase locking on top of the previously-verified implementation of a crash-safe journal. Having proved the correctness of two-phase locking, the file system behaves equivalently to a sequential system, enabling the use of a sequential proof system.

In contrast, we derive the Java implementation manually from the specification and reason only informally about the crash safety of our journal, our focus being on the higher levels. We leverage the explicit versioning of our store interface to show (in a future publication) that all stores are behaviourally equivalent; this obviates the need to specifically prove equivalence to a sequential execution.

Similarly, Chen et al. [Che+15] produce a machine-checked proof that the implementation of a simple file system recovers correctly from crashes. The paper extends Hoare logic with specific rules to describe crashing and recovery. As explained above, we do not formalize crash resistance, and use standard Hoare-style logic.

Malecha et al. [Mal+10] prove the correctness of a relational database management system, including B-Tree storage and query optimization.

In a related vein, Hawblitzel et al. [Haw+15] prove correct a sharded, distributed key-value store that synchronizes with Paxos. Our specification allows more behaviours as it does not assume consensus, but does enforce transactional semantics. Fleshing out distributed sharding is future work.

An alternative approach is to use a formal specification to generate tests. For instance, Ridge et al. [Rid+15] derive a test oracle for file systems from a formal specification. This is an interesting direction for future work.

9.0.4 Compiling specifications to executable code

A recent approach is to compile an implementation directly from the formal specification Hackett et al. [Hac+23] and Costa [Cos19]. While attractive, we did not attempt this avenue, which we believe is still out of reach pragmatically. We believe our approach will be educational and appealing to software engineers.

10 Conclusion

Throughout this thesis, we have delved into the formal specification of a database backend. From the system model to the formal model of transactions, our goal was to provide a rigorous ground for implementing a correct database backend. We then presented our implementation of multiple variants of a store based on different data structures.

Building on our model, we implemented multiple variants. Using the variants we presented the Conductor, an approach to compose multiple stores, such as map-based and journal-based store into a single system. We presented the semantic for the composition and presented an implementation of the Conductor. We tested through testing the behavioural equivalence between our stores and show the performance of the implementation are decent by leave room for improvement and future work.

Finally, we talk about the current and future work that can improve on the building blocks presented in this thesis. We hope the work presented in this thesis will lead to a better understanding of the complexity of a database backend and help in the design of future database backend and make them safer and more reliable.

Bibliography

[Aha+95] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. “Causal Memory: Definitions, Implementation, and Programming”. In: *Distrib. Comput.* 9.1 (1995), 37–49 (cit. on pp. 16, 17).

[Akk+16] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, et al. “Cure: Strong semantics meets high availability and low latency”. In: *Int. Conf. on Distributed Comp. Sys. (ICDCS)*. Nara, Japan, June 2016, pp. 405–414 (cit. on pp. 68, 94).

[ASB15] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. “Efficient State-Based CRDTs by Delta-Mutation”. In: *Int. Conf. on Networked Systems (NETYS)*. Vol. 9466. Lecture Notes in Comp. Sc. (LNCS). Agadir, Morocco: Springer-Verlag, May 2015, pp. 62–76 (cit. on p. 20).

[BAS17] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. *Pure Operation-Based Replicated Data Types*. ArXiv e-print 1710.04469. arXiv Computing Research Repository (CoRR), Oct. 2017 (cit. on p. 95).

[BM99] Carlos Baquero and Francisco Moura. “Using structural characteristics for autonomous operation”. In: *Operating Systems Review* 33.4 (1999), pp. 90–96 (cit. on p. 20).

[BM70] R. Bayer and E. McCreight. “Organization and Maintenance of Large Ordered Indices”. In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET ’70. Houston, Texas: Association for Computing Machinery, 1970, 107–141 (cit. on p. 9).

[Ben75] Jon Louis Bentley. “Multidimensional Binary Search Trees Used for Associative Searching”. In: *Commun. ACM* 18.9 (1975), 509–517 (cit. on p. 11).

[Ber+95] Hal Berenson, Phil Bernstein, Jim Gray, et al. “A Critique of ANSI SQL Isolation Levels”. In: *SIGMOD Rec.* 24.2 (May 1995), pp. 1–10 (cit. on p. 15).

[Ber] Bernhardsson. *GitHub - spotify/annoy: Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk — github.com*. `https://github.com/spotify/annoy`. [Accessed 29-Apr-2023] (cit. on p. 11).

[BG84] Philip A. Bernstein and Nathan Goodman. “An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases”. In: *ACM Trans. on Database Systems (TODS)* 9.4 (Dec. 1984), pp. 596–615 (cit. on p. 13).

[BG81] Philip A. Bernstein and Nathan Goodman. “Concurrency Control in Distributed Database Systems”. In: *ACM Comput. Surv.* 13.2 (1981), 185–221 (cit. on p. 7).

[Bey+99] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. “When Is ”Nearest Neighbor” Meaningful?” In: *Proceedings of the 7th International Conference on Database Theory*. ICDT ’99. Berlin, Heidelberg: Springer-Verlag, 1999, 217–235 (cit. on p. 10).

[Blo70] Burton H. Bloom. “Space/Time Trade-Offs in Hash Coding with Allowable Errors”. In: *Commun. ACM* 13.7 (1970), 422–426 (cit. on p. 23).

[Bre12] Eric Brewer. “CAP Twelve Years Later: How the “Rules” Have Changed”. In: *IEEE Computer* 45.2 (Feb. 2012), pp. 23–29 (cit. on p. 25).

[Bur+14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. “Replicated Data Types: Specification, Verification, Optimality”. In: *Symp. on Principles of Prog. Lang. (POPL)*. San Diego, CA, USA, Jan. 2014, pp. 271–284 (cit. on p. 34).

[CBG15] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. “A Framework for Transactional Consistency Models with Atomic Visibility”. In: *Int. Conf. on Concurrency Theory (CONCUR)*. Ed. by Luca Aceto and David de Frutos Escrig. Vol. 42. Leibniz Int. Proc. in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015, pp. 58–71 (cit. on pp. 14, 101).

[Cha+22] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. “Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning”. In: *Symp. on Op. Sys. Design and Implementation (OSDI)*. Ed. by Marcos K. Aguilera and Hakim Weatherspoon. Carlsbad, CA, USA: Usenix, July 2022, pp. 447–463 (cit. on p. 102).

[Che+15] Haogang Chen, Daniel Ziegler, Tej Chajed, et al. “Using Crash Hoare Logic for Certifying the FSCQ File System”. In: *Symp. on Op. Sys. Principles (SOSP)*. SOSP ’15. Monterey, CA, USA: Assoc. for Computing Machinery, Oct. 2015, pp. 18–37 (cit. on p. 102).

[Cod70] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Commun. ACM* 13.6 (1970), 377–387 (cit. on p. 7).

[Cos19] Renato Mascarenhas Costa. “Compiling distributed system specifications into implementations”. MA thesis. Vancouver, BC, Canada: U. of British Columbia, May 2019 (cit. on p. 103).

[Cro+17] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. “Seeing is Believing: A Client-Centric Specification of Database Isolation”. In: *Symp. on Principles of Dist. Comp. (PODC)*. PODC ’17. Washington, DC, USA: Assoc. for Computing Machinery, 2017, pp. 73–82 (cit. on p. 101).

[Dat03] Chris J. Date. *An Introduction to Database Systems*. 8th ed. Boston, MA: Addison-Wesley, 2003 (cit. on p. 7).

[DEZ13] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. “Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks”. In: *Symp. on Reliable Dist. Sys. (SRDS)*. Braga, Portugal: IEEE Comp. Society, Oct. 2013, pp. 173–184 (cit. on p. 94).

[Fid88] C. J. Fidge. “Timestamps in message-passing systems that preserve the partial ordering”. In: *11th Australian Computer Science Conference*. U. of Queensland, Australia, 1988, pp. 55–66 (cit. on p. 33).

[GL02] Seth Gilbert and Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *SIGACT News* 33.2 (2002), pp. 51–59 (cit. on p. 25).

[GIM99] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. “Similarity Search in High Dimensions via Hashing”. In: *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB ’99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, 518–529 (cit. on p. 11).

[Fai] *GitHub - facebookresearch/faiss: A library for efficient similarity search and clustering of dense vectors. — github.com*. `https://github.com/facebookresearch/faiss`. [Accessed 29-Apr-2023] (cit. on p. 11).

[Roc] *GitHub - facebook/rocksdb: A library that provides an embeddable, persistent key-value store for fast storage. — github.com*. `https://github.com/facebook/rocksdb/`. [Accessed 29-Apr-2023] (cit. on p. 9).

[Lev] *GitHub - google/leveldb: LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values. — github.com*. `https://github.com/google/leveldb`. [Accessed 29-Apr-2023] (cit. on p. 9).

[Nms] *GitHub - nmslib/nmslib: Non-Metric Space Library (NMSLIB): An efficient similarity search library and a toolkit for evaluation of k-NN methods for generic non-metric spaces. — github.com*. `https://github.com/nmslib/nmslib`. [Accessed 29-Apr-2023] (cit. on p. 11).

[GR92a] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992 (cit. on p. 7).

[GR92b] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992 (cit. on p. 13).

[Hac+23] Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastnikh. “Compiling Distributed System Models with PGo”. In: *Int. Conf. on Archi. Support for Prog. Lang. and Systems (ASPLOS)*. ASPLOS 2023. Vancouver, BC, Canada: Assoc. for Computing Machinery, 2023, pp. 159–175 (cit. on p. 103).

[HR83] Theo Haerder and Andreas Reuter. “Principles of Transaction-Oriented Database Recovery”. In: *ACM Comput. Surv.* 15.4 (1983), 287–317 (cit. on p. 12).

[Han+23] Travis Hance, Yi Zhou, Andrea Lattuada, et al. “Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems”. In: *Symp. on Op. Sys. Design and Implementation (OSDI)*. Usenix. Boston, MA, USA, July 2023, TBD (cit. on p. 102).

[HS20] Saalik Hatia and Marc Shapiro. *Specification of a Transactionally and Causally-Consistent (TCC) database*. Research Report RR-9355. DELYS ; LIP6, Sorbonne Université, Inria, Paris, France, July 2020 (cit. on p. 3).

[Haw+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, et al. "IronFleet: Proving Practical Distributed Systems Correct". In: *Symp. on Op. Sys. Principles (SOSP)*. SOSP '15. Monterey, CA, USA: Assoc. for Computing Machinery, Oct. 2015, pp. 1–17 (cit. on p. 102).

[HSH07] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. "Architecture of a Database System". In: *Found. Trends Databases* 1.2 (2007), 141–259 (cit. on pp. 12, 13).

[HW90] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), 463–492 (cit. on p. 15).

[Lam79] L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Trans. Comput.* 28.9 (1979), 690–691 (cit. on p. 14).

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565 (cit. on p. 17).

[Mal+10] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. "Toward a Verified Relational Database Management System". In: *Symp. on Principles of Prog. Lang. (POPL)*. POPL '10. Madrid, Spain: Assoc. for Computing Machinery, 2010, pp. 237–248 (cit. on p. 102).

[MY18] Yu. A. Malkov and D. A. Yashunin. *Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs*. 2018. arXiv: `1603.09320 [cs.DS]` (cit. on p. 11).

[Mat88] Friedmann Mattern. "Virtual Time and Global States of Distributed Systems". In: *Int. W. on Parallel and Distributed Algorithms*. Bonas, France: Elsevier Science Publishers B.V. (North-Holland), Oct. 1988, pp. 215–226 (cit. on p. 33).

[Moh+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging". In: *ACM Trans. Database Syst.* 17.1 (1992), 94–162 (cit. on p. 21).

[Omo89] Stephen M. Omohundro. *Five Balltree Construction Algorithms*. Tech. rep. TR-89-063. International Computer Science Institute, 1989 (cit. on p. 11).

[O'N+96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The log-structured merge-tree (LSM-tree)". In: *Acta Informatica* 33.4 (June 1996), pp. 351–385 (cit. on pp. 9, 23, 68).

[Pap79] Christos H. Papadimitriou. "The Serializability of Concurrent Database Updates". In: *J. ACM* 26.4 (1979), 631–653 (cit. on p. 13).

[Ree78] D. P. Reed. *NAMING AND SYNCHRONIZATION IN A DECENTRALIZED COMPUTER SYSTEM*. Tech. rep. USA, 1978 (cit. on p. 21).

[Ree83] David P. Reed. “Implementing Atomic Actions on Decentralized Data”. In: *ACM Trans. Comput. Syst.* 1.1 (1983), 3–23 (cit. on p. 12).

[Rid+15] Tom Ridge, David Sheets, Thomas Tuerk, et al. “SibylFS: Formal Specification and Oracle-Based Testing for POSIX and Real-World File Systems”. In: *Symp. on Op. Sys. Principles (SOSP)*. Monterey, CA, USA: Assoc. for Computing Machinery, Oct. 2015, pp. 38–53 (cit. on p. 102).

[RO92] Mendel Rosenblum and John K. Ousterhout. “The Design and Implementation of a Log-Structured File System”. In: *ACM Trans. on Computer Systems (TOCS)* 10.1 (Feb. 1992), pp. 26–52 (cit. on p. 9).

[Sha+11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. “Conflict-free Replicated Data Types”. In: *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS)*. Ed. by Xavier Défago, Franck Petit, and V. Villain. Vol. 6976. Lecture Notes in Comp. Sc. (LNCS). Grenoble, France: Springer-Verlag, Oct. 2011, pp. 386–400 (cit. on pp. 19, 34).

[Sto02] Michael Stonebraker, ed. *Readings in Database Systems, Second Edition*. Morgan Kaufmann, Jan. 3, 2002 (cit. on p. 7).

[VV16] Paolo Viotti and Marko Vukolić. *Consistency in Non-Transactional Distributed Storage Systems*. 2016. arXiv: `1512.00168 [cs.DC]` (cit. on pp. 14, 18).

[Yia93] Peter N. Yianilos. “Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces”. In: *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’93. Austin, Texas, USA: Society for Industrial and Applied Mathematics, 1993, 311–321 (cit. on p. 11).

List of Figures

List of Tables

List of Listings

A

Notations

$k, \mathtt{t}, v$	$\in Key, \in \mathtt{TS}, \in Value$	Primitive Key, Timestamp, Value
δ	$\in \mathcal{E}\mathit{ff} = Value_\perp \to Value_\perp$	Effect
$:= v$ or v	$\in \mathcal{A}\mathit{ss} \subset \mathcal{E}\mathit{ff}$	Assignment
Op	$\in = \{:= v, := v\circ := v, \dots\}$	Operation, non-assignement
$\perp$	$\notin \mathcal{E}\mathit{ff}$	Absence of effect
$\delta_k^{\mathtt{t}}$	$\in \mathcal{TE}\mathit{ff} = \mathcal{E}\mathit{ff} \times Key \times \mathtt{TS}$	Effect to key k with commit timestamp $\mathtt{t}$
$\delta_1 \overset{vis}{\to} \delta_2$		δ_1 visible to δ_2
σ	$\in \Sigma = Key \times \mathtt{TS} \to \mathcal{E}\mathit{ff}_\perp$	Store
$\mathcal{F} = (low\_history, \mathcal{D})$	$\in \mathtt{TS} \times 2^{Key \times \mathtt{TS}}$	Field of store
$\{\ \sigma_1, \sigma_2, \dots\ \}$		Composition of stores
Θ	$\in (\mathcal{TE}\mathit{ff}, \overset{vis}{\to})$	Trace
τ	$\in \mathcal{T}_{ID}$	Transaction identifier
$\mathtt{dt}$	$\in \mathtt{TS}$	Dependency timestamp (of transaction)
$\mathcal{R}$	$\in \mathcal{P}(Key)$	Read set = set of keys read in transaction
$\mathcal{W}$	$\in \mathcal{P}(Key)$	Dirty set = set of keys modified in transaction
$\mathcal{B}$	$\in Sbuf = Key \to \mathcal{E}\mathit{ff}_\perp$	State buffer (of transaction)
$\mathtt{ct}$	$\in \mathtt{TS}$	Commit timestamp (of transaction)
$(\tau, \mathtt{dt}, \mathcal{R}, \mathcal{W}, \mathcal{B}, \mathtt{ct})$	$\in \mathcal{T}_{DESC} = \mathcal{T}_{ID} \times \mathtt{TS}$	
	$\times \mathcal{P}(Key) \times \mathcal{P}(Key) \times Sbuf \times \mathtt{TS}$	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
LH		Low history
LL		Low lookup
HL		High lookup
$\sigma, \mathcal{T}_\mathrm{r}, \dots$		Before transition
$\sigma', \mathcal{T}_\mathrm{r}', \dots$		After transition