NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

RICARDO ALEXANDRE CRUZ MARGALHAU

Bachelor of Computer Science and Engineering

IMPROVING LAZY STATE DETERMINATION

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

April, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

IMPROVING LAZY STATE DETERMINATION

RICARDO ALEXANDRE CRUZ MARGALHAU

Bachelor of Computer Science and Engineering

Adviser: João Lourenço
Associate Professor, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa

Examination Committee

Chair: Nuno Correia
Full Professor, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa

Rapporteur: João Barreto
Associate Professor, Instituto Superior Técnico, Universidade de Lisboa

Adviser: João Lourenço
Associate Professor, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
April, 2025

Improving Lazy State Determination

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.3.9) [15].

Acknowledgements

Firstly, I would like to express my gratitude and thanks to my adviser, João Lourenço, which helped get this project to where it stands today, by not only advising me, but also other students who have worked on Lazy State Determination throughout the years. I also thank him for his incredible patience with me, as without it this would not have been possible.

I thank all the professors I have encountered during my academic path, as they played a fundamental role in getting where I am today. Special thanks go to the NOVA School of Science and Technology and the Department of Computer Science, for providing us, students, with the necessary tools to learn and create.

I would also like to express my immense gratitude to my team at Cloudflare for being awesome, and making going through this journey feel like a breeze. Without them, I probably would not have made where I am today, so my heartfelt thanks goes to them.

To all my friends who, even though they probably incentivized procrastination, made me not give up by making me laugh at countless moments and by being there when I needed them most.

Lastly, I want to express gratitude to my family, to my mom Rosa and dad Paulo, to my sister Catarina, to my grandmother Benedita, and to my late grandfather António, who always believed in me and provided me with care and love, so I would be able to pursue my dreams.

Abstract

High-performance transaction systems are used for online transaction processing in major platforms, like online shopping or social media platforms. It is important that these systems exhibit high throughput, with a low-latency, so that the final users do not experience unresponsiveness while browsing the platform. When under high contention, these systems often experience high abort rates or deadlocks, which is not ideal for the use-cases these systems are made for.

Lazy State Determination (**LSD**) helps by delaying, as much as possible, the need to retrieve concrete data from the database system, thus decreasing the window of opportunity for the data to be changed by other transactions, which improves transaction success rates. In order to create and execute transactions, **LSD** uses futures instead of concrete values, proving operations to compose these futures. When committing a transaction, **LSD** evaluates all futures by fetching concrete values, completing the transaction successfully if the conditions are maintained.

We propose a solution to improve the shortcomings of the **LSD** specification, as presented by Vale [25] and Carpinteiro [3].

Keywords: Transactions · Database Management Systems · Lazy State Determination · High Contention · Concurrency Management · High-performance Transactions · Online Transaction Processing

Resumo

Sistemas de alto desempenho transacional são usados para processamento de transações *online* em grandes plataformas, como plataformas de compra e venda *online*, ou redes sociais. Para que os utilizadores destas plataformas não tenham uma má experiência de utilização (e.g., tempos de carregamento longos), é necessário que estes sistemas exibam uma baixa latência e uma alta capacidade de processamento transacional. Durante períodos de alta contenção, estes sistemas experienciam regularmente taxas de aborto elevadas, ou *deadlocks*, o que não é ideal para os propósitos que estes sistemas foram construídos para.

Lazy State Determination (**LSD**) ajuda, adiando, o quão possível, a necessidade de obter dados concretos do sistema de bases de dados, assim diminuindo a janela de oportunidade em que os dados podem ser modificados por outras transações, melhorando a taxa de sucesso transacional. Para criar e a executar transações, **LSD** usa futuros em vez de valores concretos, providenciando operações para compor estes futuros. Aquando do *commit*, o **LSD** avalia todos os futuros, obtendo os valores concretos, e completando a transação com sucesso caso as condições observadas pelos futuros sejam mantidas.

Propomos uma solução para endereçar os problemas encontrados na especificação do **LSD** apresentada por Vale [25] e Carpinteiro [3].

Palavras-chave: Transações · Sistemas de Gestão de Bases de Dados · Lazy State Determination · Alta Contenção · Gestão de Concorrência · Alto Desempenho Transacional · Processamento de Transações Online

Contents

List of Figures

List of Tables

Listings

List of Algorithms

1 Introduction

1.1 Context

Database systems are used in every-day activities, such as social networking, online shopping, and productivity tasks. Humans are usually unaware of these systems, or how they work, but they can always perceive the effects of a bad (or good) database management system — the system they are using becomes slow and/or non-responsive.

Technology has evolved much in the past decade, and common users are always expecting more from the computational systems. In the context of database systems, this is often reflected in the number of transactions per second that the system can process, which has a considerable impact to the end-user in terms of response time(s) and productivity.

To run a successful business in the modern days of Internet, it is necessary to have good database management systems. For instance, to run a maps platform, where the users can find the quickest route between two points, a database system must be prepared to handle geographical and graph data very efficiently, otherwise users may find a competitor that provides a better service.

1.2 Motivation

The main motivation behind this work is that database systems need to be efficient and performant. These systems are used behind platforms that serve millions of users at the same time, like in social networking or online shopping platforms, where efficiency and performance is paramount to run a successful business. An efficient database system presents big monetary savings for large businesses (hardware resources, cluster maintenance, among other expenses), which can have an impact on its customers, whereas a performant database system provides a better user experience, that in turn can generate more money for the business (if a user has to wait multiple seconds to buy something, the user may give up and go to a competitor).

Picking a good starting database is essential for a business to grow, and when the

system that has been picked doesn't correspond to what we have been expecting, or when other database systems have found ways to address efficiency and performance issues significantly better, we may encounter ourselves in a situation where it is better to change, even if the cost is high. Discord [6], a popular social networking platform, had to recently make this change for one of its core systems — the messaging service — from **Cassandra** [1] to **ScyllaDB** [21], as the old solution became too expensive to maintain [7].

1.3 Problem

Modern database management systems use a client-server model, where we can have hundreds, or thousands, of clients accessing a database node at the same time (in reality, modern client implementations rely on connection pooling, to limit the number of clients for each node), but the number of database servers typically ranges from one (a single-node cluster), to about a few hundred (a large cluster of nodes). This poses a couple of problems for modern database systems:

1. How many nodes do we need to handle all the clients?
2. What is the impact of the network-layer implementation in throughput?

Current database concurrency controls tend to favor either low-contention systems, or high-contention systems, not both. This is not ideal due to the fact that, in reality, most database workloads are variable. For instance, during Christmas, an online shopping platform database system would observe more contention than during other times of the year. This means that during these peaks, database systems can observe a high number of conflicts (with optimistic approaches), or unacceptable slowdowns (in pessimistic approaches).

Testing and improving the existing prototype is our main focus with this work, and to achieve this end we must benchmark and test our current prototype with battle-tested technologies in order to assess where the bottlenecks are. To this end, we must modernize the toolchain used during development, as this is crucial to obtain accurate results. This performance testing will focus on three areas: the concurrency protocol, the network layer and the storage layer.

Our approach for this work is based on the existing work done for **LSD** in K-V stores [3] – the use of futures to reduce conflicts and improve performance under different workloads. Improving the existing futures will require us to take a closer look at the implementation of these operations, namely the **is-true** operation, where bottlenecks may be of significance.

1.4 Approach

The current prototype of Lazy State Determination has two implementations: the Optimistic Concurrency Control (OCC) version, and the Two-Phase Locking (2PL) version. With this

work, we expect to improve the existing OCC version, and to validate and improve the 2PL version, as the latter version has not been tested to the same degree as the OCC version.

Moreover, the existing toolchain used to compile, test and benchmark our implementation is old and difficult to use. With our work, we plan to improve this toolchain, leaving it in a more mature state for future work. We expect to introduce new benchmarks, allowing us to gather more data about the performance of our implementation. We also expect to make it more simple to compile, debug, test and benchmark the implementation, with comprehensive documentation on how to do each of these steps.

In order to improve the performance of our application, we will also need to evaluate the performance of our network and storage layers, so that we can determine if these layers have any significant impact over our ability to improve the overall performance of our prototype. This requires having a good toolchain in order to test and benchmark our implementation.

1.5 Document Organization

In Chapter 2 we present the background and related work that will enable us to develop this work. We give an overview over approaches to concurrency control in database systems, improvements that have been made by iterating over already existing approaches, or by using new approaches, and an introduction to what is Lazy State Determination [25].

Chapter 3 presents background on how Lazy State Determination works, and explores into what problems we have found with the original specification.

During Chapter 4, we focus on presenting the improvements we have made to the Lazy State Determination algorithms, in order to overcome the problems presented in the earlier chapter.

The work that was done while trying to evaluate our improvements to Lazy State Determination is presented in Chapter 5. In this chapter we explain how expanding the modular architecture allows us to create more comparisons and finding the source of performance issues. We also explain how we leveraged **TPC-C** [23] in order to conduct performance evaluations.

Finally, in Chapter 6 we discuss the work we have presented, what was done, and what future improvements could look like.

2

Background & Related Work

Transactional systems are widely used nowadays by, for example, online marketplaces and social networking platforms. These systems allow grouping a set of operations – a Transaction – and execute it as an atomic unit. In order to guarantee that the transactions are in fact executed as an atomic unit, a set of properties, known as **ACID** [12], has been defined [10]:

- Atomic — Either every operation present in the transaction is executed successfully, or none is;
- Consistent — Transactions must leave the database in a consistent state, where none of the imposed constraints are violated;
- Isolated — With parallel transactions, a transaction should not be able to interfere with other transactions executing simultaneously; and
- Durable — The effects of successful transactions should be persisted, even in the case of a system failure.

Isolation levels define the behavior a database system is expected to exhibit when multiple parallel transactions try to access the same data. The **SQL-92** [29] standard defines *read phenomena* that can happen when executing several transactions simultaneously:

- Dirty read — A transaction returns a row that has been updated by another transaction that has not yet committed;
- Non-repeatable reads — When a transaction fetches a row in two distict moments, but due to an update by another transaction, the second row fetch returns different data; and
- Phantom read — If a row is inserted or deleted by a transaction between two table row retrievals of another transaction;

The SQL standard also defines isolation levels. Isolation levels must guarantee that, at least, certain *phenomena* are prevented from appearing in them:

- Read uncommitted — No *read phenomena* are prevented;
- Read committed — Must prevent, at least, dirty reads;
- Repeatable read — Must prevent, at least, dirty reads and non-repeatable reads; and
- Serializable — Must prevent dirty reads, non-repeatable reads and phantom reads.

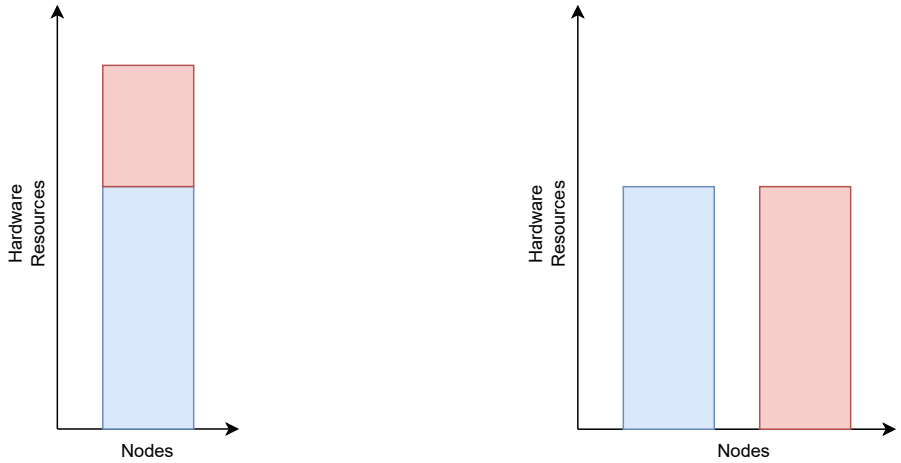
Having the ability to set different isolation levels for different transactions, enables database systems to increase transaction performance at the cost of *read phenomena*, which means that the stricter our transaction isolation, the bigger the performance penalty.

2.1 High-Contention Transactions

Performing database transactions in databases under high contention is a difficult task to accomplish, because with many users accessing the system at the same time, we observe higher abort rates (e.g., in the case of Optimistic Concurrency Control) or deadlocks when using locks to safely guarantee concurrent access to a tuple, which lead the performance of the database, and thus the performance of the application, to deteriorate.

To address the issue of performance and consistency under high contention, some databases have adopted less restrictive consistency models, that allow for more performance, with the promise that the database system will eventually evolve towards data consistency. However, using a looser consistency model is not ideal for every system. For instance, let us think about a system with monetary transactions, where there is a seller and a buyer. The buyer wants to buy a pair of shoes from the seller, at the price of 200u, and the seller has one unit of this type of shoes left in the warehouse. In a strict consistency scenario, the 200u would get removed from the buyer's bank account and deposited in the seller's account, with the shoes being transferred from the seller's inventory to the buyer's inventory, in a single atomic transaction. This requires either using 2PL, to acquire the necessary locks to make sure that the data is not modified by another transaction, or using an optimistic model such as the one implemented by Multi-Version Concurrency Control (**MVCC**) [28], reading the data at a specific version, and when committing checking if the version is still the same. With a less restrictive consistency model, we don't have the guarantees that another user won't be able to buy the shoes, even if the shoes were already, supposedly, bought by our first user. This would generate some problems for the seller, having to decide which user would get the shoes, and refund the money to the other one.

When planning a system we could decide the compromises we want to make in order to achieve more performance, but with a lot of systems, namely in systems that move money around, we want a strict consistency model, in order to avoid reaching an inconsistent state. A strict consistency makes it harder to achieve high performant systems when under high contention, because of the restrictions imposed to maintain this consistency level, however, we want the users to be happy while using our system, and keeping them waiting is not a solution.



(a) In vertical scalability, we add more resources (Red) to our existing node (Blue)

(b) In horizontal scalability, we add more nodes (Red) to the existing cluster (Blue)

Figure 2.1: Vertical vs. Horizontal Scalability

2.2 High-Performance Transaction Systems

The fast-paced environment in which the Internet operates led to the necessity of high-performance transaction systems. These systems have been improved throughout times, in order to execute these transactions faster than ever before. This required a myriad of engineering solutions, of which we will only present a selected few.

Initial high-performance transaction systems, focused on improving the performance using a single computer system resources, that is, the more resources we had in our computer system, the better the performance of our transaction system. This approach is known as **vertical scalability**, due to the fact that we are only scaling our single computer system. Scaling a computer system with this approach can significantly help our transaction system. As an example, the advent of high-capacity flash storage with Solid State Drives, led to more Throughput and I/O operations (IOPS), which have a significant impact on the performance of database systems, thus just by changing from a conventional hard-drive to flash storage, we can increase the performance of our database system.

Having multiple users spread across multiple points of the globe increased the necessity of having database systems in multiple points-of-presence, closer to the users of the systems, in order to reduce latency and improve user-experience. In response to this demand, replication techniques have been added to popular database systems, such as **PostgreSQL**. Replication allow us to effectively replicate data across multiple computer systems, providing **horizontal scalability**, where we can distribute the database load across multiple nodes, when **vertical scalability** becomes unfeasible (there is a limit on how much hardware we can provide a single node), or when we need multiple points-of-presence. Figure 2.1 shows a comparison between vertical and horizontal scalability. Replication can be used with two modes:

- Multi-primary replication — we have more than one primary node (i.e., node that accepts writes); and
- Primary-backup replication — only one node is the primary, and the others serve as backup nodes (i.e., only providing the ability to read data).

Primary-backup replication (shown in Figure 2.2) is the easiest mode to achieve, because we only have a single node controlling the writes to the database system, thus we do not need a distributed concurrency control scheme, as the backup nodes only serve as a mirror of the primary node.

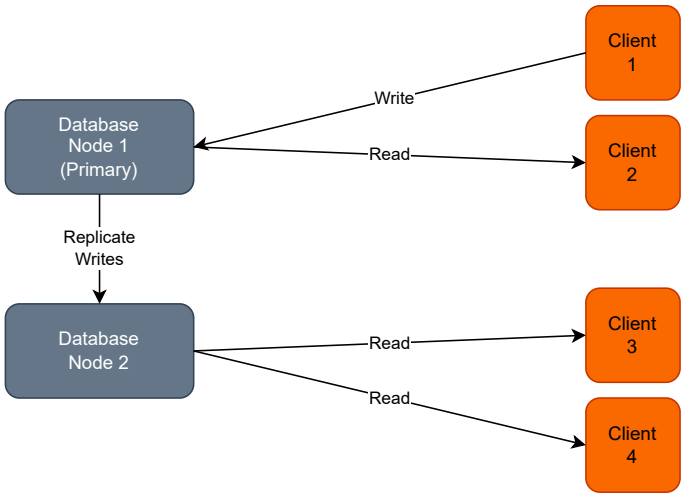


Figure 2.2: Primary-backup replication, where clients only write to primary, but can read from every node

Multi-primary replication systems (as shown in Figure 2.3), on the other hand, have to agree on writes across multiple nodes, possibly geographically distributed. The latter approach to replication has only been possible due to the research conducted on distributed systems, with the creation of protocols such as **Paxos** [13] and **Multi-Paxos** [26]. Modern database systems use **3-Phase Commit** [30] under the **Multi-primary** replication mode.

With larger databases, it is impractical having full replicated instances of the database across multiple nodes. This requires a significant investment in terms of storage capacity, and in most of the cases, this investment does not increase the gains obtained by having full replication. Instead, database systems present another approach called distributed data partitioning. Undistributed partitioning has been present in most popular database systems for several years now, where the idea is to partition data in multiple groups, each one identified by its partition key (e.g., the day at which a row has been inserted), increasing performance when we have to search a big table, since we now have to search only one partition (even though sometimes we may need to search multiple partitions for complex queries). Distributed partitioning of data takes it a step further, by storing partitions where they are most needed. As an example, a messaging service, can have a partition for each region (e.g., North America and Europe), and stores messages only

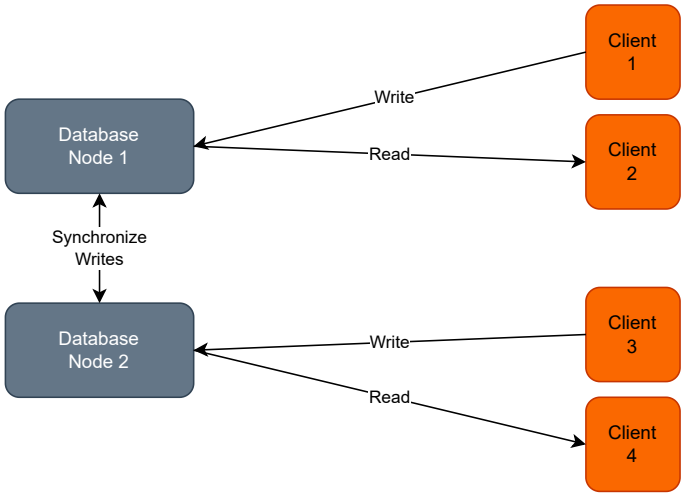


Figure 2.3: Multi-primary replication, where clients can write to multiple nodes

within the region they were created in. This approach is present in modern databases, such as **CockroachDB** [4], allowing enterprises to keep these systems cost-effective, and to also keep in compliance with data regulation laws. Figure 2.4 shows an example where users only read and write to their local region, only having full replication across regions for tables that are not partitioned by region.

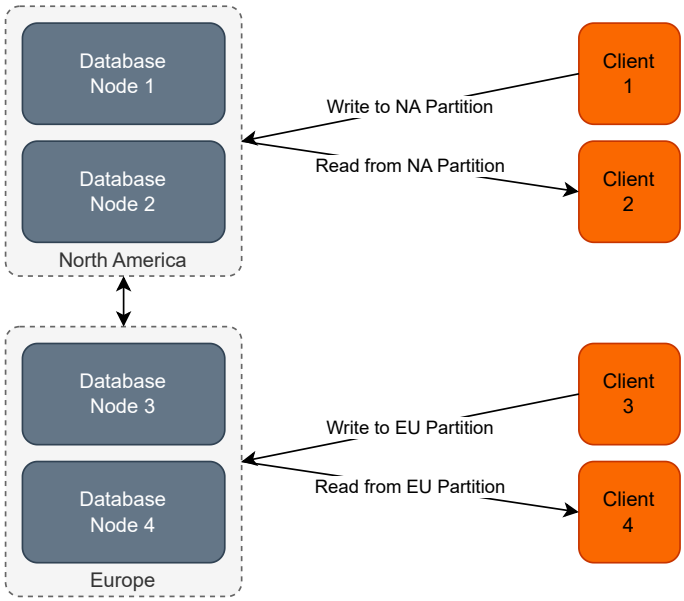


Figure 2.4: Distributed partitioning, with users reading and writing to their local region

2.3 Concurrency Control Strategies

Due to the need to support concurrency (i.e., many users accessing the database at the same time), we need some sort of concurrency control in order to make sure that concurrent operations to the database data are properly synchronized, or else we could reach a

state with inconsistent data. One approach would be to execute all transactions serially, however this scales very poorly with the number of concurrent users, thus we must use other solutions to guarantee that we can achieve a serial-equivalent execution, with a higher performance.

Locking-based mechanisms are a form of pessimistic concurrency controls, as we assume that every piece of data we want to lock will be used by concurrent transactions. The most famous locking mechanism is Two-Phase Locking [31], where we have two phases: the **locking phase**, and the **release phase**. During the **locking phase**, the transaction acquires all locks and no locks are released during this phase, whereas in the **release phase**, all locks are released and none can be acquired. This approach works well under high contention, but under low or even medium contention, it becomes expensive to maintain these locks. This protocol for acquiring and releasing locks is depicted in Figure 2.5. By using this approach, there is a risk of reaching a deadlock between two transactions, which needs to be handled as well.

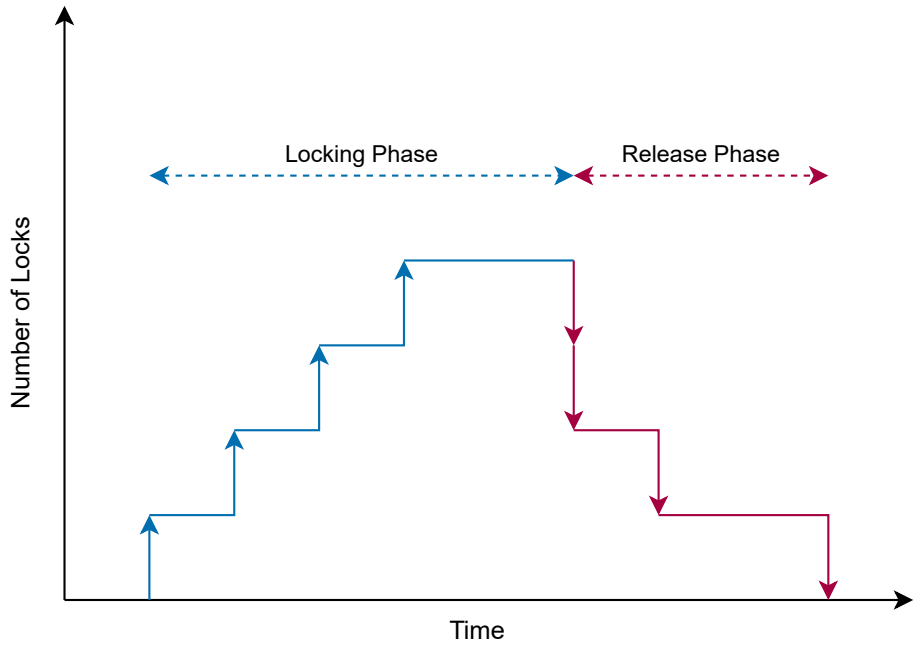


Figure 2.5: Phases of the Two-Phase Lock Protocol

Optimistic approaches come to the rescue in workloads that do not show a contention level that justifies the use of Two-Phase Locking. The optimistic approach works by maintaining a write set and a read set. These sets hold the data that has been written or read during the transaction, respectively, and when the transaction is committed, it goes through two phases: the **validation phase** and the **write phase**. The **validation phase** is responsible for guaranteeing that the data that has been read or written by this transaction does not conflict with data that has been modified by another concurrent transaction. If there is a conflict detected during this phase, the transaction must be aborted and retried again. The **write phase** is encompassed with the task of writing the data modified by the transaction (and held in the write set) to the storage mechanism, so it becomes the new

source of truth.

Multi-Version Concurrency Control (**MVCC**) [28] is a type of optimistic concurrency control used in widely known databases, such as **PostgreSQL** [22]. This approach works by maintaining several versions of data, that can be used independently, and then reconciling the changes to the source of truth. When each transaction starts it acquires a snapshot (depending on the isolation level, the moment at which the snapshot was captured can actually be different) which is then used during the remainder of the transaction to perform read and write operations.

2.4 Improvements to Pessimistic Concurrency Control

One way to improve 2-Phase Locking, is by violating the protocol to be able to perform dirty reads on hotspots [11]. Reading dirty data is not allowed by the conventional 2PL definition (i.e., a transaction cannot acquire new locks after releasing locks to other transactions). The implementation of this is not compatible either with OCC, in which the data is only visible to other transactions after the transaction commits.

To enable dirty reads, Transaction chopping [32] has been explored extensively. This approach consists of dividing a transaction in smaller transactions, which allows the sub-transactions to make the updates immediately visible after it finishes, however this approach as a few limitations:

- We need to know our workload;
- Must follow specific criteria to avoid deadlocks and ensure serializability; and
- Conservative conflict detection could lead to unnecessary waiting.

2.5 Improvements in Optimistic Concurrency Control

OCC transactions with timestamps perform poorly on multicore systems, due to the fact that they use a centralized timestamp to resolve conflicts between different concurrent transactions at commit time. This timestamp is stored as an atomic value, which significantly limits parallelism, because this critical section cannot be accessed by two threads at the same time. There were a few attempts in the past to solve this issue, both at the software and hardware levels. Software solutions included for instance the one proposed by **Silo** [24] to make this timestamp coarse grained (i.e., each epoch represents 40ms), which significantly decreases the need to increment the atomic counter, and increases throughput. Hardware solutions include, for instance, shared clocks, which are hard and, more importantly, costly, to implement.

Tictoc [33] tackles this challenge by, instead of having a determined commit time timestamp, it has read and write timestamps for every tuple that the transaction read or has written to. This allows later at commit time to dynamically compute the commit

timestamp. Essentially, the commit timestamp of the transaction will be determined by Formula 2.1, which is defined as the maximum of two variables: the write timestamp of the read set and the next read timestamp of the write set, that is, the commit timestamp is the write timestamp of the read set (the latest modification that occurred), or the next read timestamp of the write set (we have modified the variable, and need to create a new version). This allows **Tictoc** to later check if Invariant 2.2 holds.

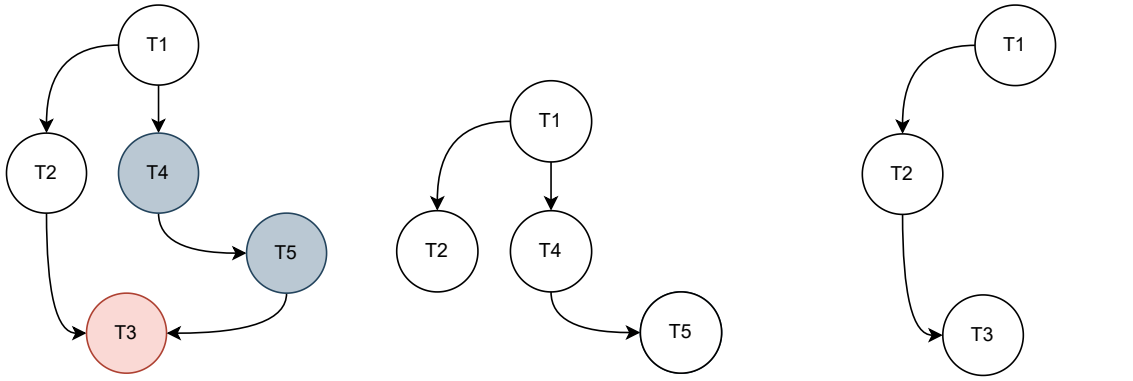
$$max(read\_set\_wts, write\_set\_rts + 1) \tag{2.1}$$

$$\begin{aligned} \exists commit\_ts, (\forall v \in \{\text{versions read by T}\}, v.wts \leq commit\_ts \leq v.rts) \\ \wedge (\forall v \in \{\text{versions written by T}\}, v.rts < commit\_ts) \end{aligned} \tag{2.2}$$

Invariant 2.2 checks if, for every object's version read during the transaction, the commit timestamp, as determined by Formula 2.1, is between the object's write timestamp and its read timestamp. The commit timestamp must be equal or greater than the write timestamp that happened to an object read in a transaction, otherwise we would read an outdated version. The commit timestamp must also be less or equal than the read timestamp of every object that was read, due to the fact that another transaction could have modified the tuple at a logical time between the local read timestamp of the tuple and the commit timestamp. The invariant also checks if, for every object written by the transaction, the commit timestamp of the transaction happened after the object's read timestamp, that is, every object write can only happen if the transaction possesses the latest version of the object.

MVCC improves the performance under high-contention by keeping track of multiple versions of the same data, increasing the space necessary to keep such data. This introduces a significant overhead when under a low-contention workload, due to the need to keep multiple versions, as well as to garbage collect them. **Cicada** [14] aims to provide a transactional protocol that globally improves transactions in workloads with different characteristics: contention level (low-contention, high-contention), operation intensity (read, write), record size (small, large) and under high-speed transactions. In order to improve transactions across these different types of workloads, **Cicada** uses a combination of features:

- Optimistic multi-version (high-contention level, read transactions, large records);
- Loosely synchronized clocks (high-speed transactions);
- Best-effort in-lining (low-contention level, read transactions, small records);
- Rapid garbage collection (both contention levels, write transactions, both record sizes); and
- Contention regulation (high-contention level, read and write transactions)



(a) The initial batch of transactions

(b) If we abort T3 we get an acyclic batch containing T1, T2, T4 and T5

(c) If we abort T4 and T5 we get an acyclic batch with T1, T2, T3

Figure 2.6: Aborting transactions in a batch, allowing for different criteria

Transaction batching and recording [5] can improve throughput by performing operations with smaller groups of transactions. This allows to reduce abort rates in transactions of the same batch and to remove unnecessary I/O operations, which are costly to database systems in general. Overall, the improvements introduced by this approach focus in three different areas:

- Communication Efficiency — by packing several transactions in one message;
- System Request Efficiency — we can avoid certain I/O calls if we know that these calls are duplicate or stale within a specific batch; and
- Abort Efficiency — we can improve abort rates by removing some transactions of a batch that conflict.

To improve the system request efficiency and abort efficiency, we can take a look at the read phase and write phase of a transaction. We can group read and write operations in a batch. Within this batch, we can first make sure that there are no duplicate operations, which already improves performance. Moreover, we can also guarantee when picking transactions for the batch that all read operations can come after write operations, which allows for lower abort rates, since all committed writes are observed by the reads in this batch.

Abort efficiency can be further improved at the validation phase. The validator needs to pick a serializable order of the transactions in a batch. To do this, we can selectively abort transactions in a batch according to some criteria (e.g., minimize number of aborts in a batch).

We can think of transactions in a batch as a directed graph of transactions, with each edge representing a read-after-write dependency between two different transactions. Our goal is to find a subset of all transactions in this batch, such that all transactions in this batch, except for the ones included in this subset, can be run in a serial order. That is, we

want to find a subset of transactions to abort so that our batch will validate successfully. This problem can be seen as the **feedback vortex graph problem** [27], which is NP-hard and APX-hard, as we want to find which transactions we should abort such that our graph becomes acyclic. This approach is demonstrated in Figure 2.6, where we have an initial batch with multiple transactions, and we can abort certain transactions (depending on the criteria we choose) in order to form an acyclic graph of transactions.

To solve this problem, we can divide the graph into Strongly Connected Components (components with a length of zero are not aborted, since they do not have dependencies). To decide which nodes should be eliminated first from an SCC, we can define a policy, that is represented by the node weight (the transaction weight), and we can adjust the algorithm to minimize the total weight in an SCC. For instance, the policy could be defined as the degree of the node, thus eliminating nodes with more dependencies first. We can also have more complex policies that can take into account how many times this transaction has been tried, so that we can guarantee liveness in cases of high-contention.

2.6 Deterministic Concurrency Control

Deterministic Concurrency Control allow database systems to use a predetermined serial order of execution for the transactions. This approach ensures serializable execution of the transaction, while avoiding deadlocks (e.g., from using 2-Phase Lock), or aborts (e.g., as in optimistic concurrency control models). This helps in scaling distributed transaction throughput, due to the reduced need for two-phase commit, and the simplification of the replication and failure recovery protocols, since transactions can be replayed deterministically.

However, skewed and contended transactions tend to perform poorly under deterministic concurrency control This is not desirable, since the most common usage for a database is in web-applications, which suffer for skewed and contented workloads due to spikes in usage (e.g., during Christmas, people can cause spikes in online e-commerce websites).

Carcaral [19] is a novel shared-memory database that implements deterministic concurrency control that runs transactions in multiple cores. Because transactions run in multiple cores, concurrency control is required to ensure that shared-memory data operations are correct. **Carcaral** batches transactions in epochs that run in parallel, and runs the critical section of concurrency control before each transaction in the batch, thus enabling batches to be executed concurrently.

Having concurrency control is not ideal, due to the fact that it limits parallelism, which leads to a lower throughput under high-contention workloads. To improve this, **Carcaral** proposes two optimizations:

- Batch Append — Each transaction's critical section happens in any order, when transactions are appended to the batch; and

- Split-on-demand — Detects contended transactions and splits these into contended and uncontended smaller transactions.

The Preordered Transactions – **POT** [25] algorithm was created in order to reduce the concurrency overhead, by using a pre-established order of serializability to run transactions in. **POT** works by keeping track of the next allowed transaction in the list, providing two modes of operation:

- Fast-mode — used by the allowed transactions (i.e., transactions that are have been scheduled to run at a specific time with pre-ordering), using virtually no concurrency control

- Speculative mode — used by transactions that are executing concurrently with an allowed transaction (i.e., a transaction that was not allowed by pre-ordering to run at that time), using appropriate concurrency control

With fast-mode transactions, it is possible to skip some concurrency control, which in turn increases the performance of these transactions. Transactions in speculative mode still require concurrency control, as they have not been scheduled as the primary transaction at that time, and they may be running concurrently with a fast-mode transaction, or other speculative mode transactions.

2.7 Lazy-State Determination

Lazy State Determination (**LSD**) [25] helps to improve database performance, with the insight of delaying, as much as possible, the need to read and write data to the database.

In a normal scenario with a strict consistency level, 2PL and a Read Committed isolation level, locks would be acquired on the data, on the first select statement. However, this data may require some transformations, or require that some conditions be satisfied in order for something to be written. Between the transaction begin and commit statements, a lot may happen, and that takes time. It takes CPU time to compute transformations and conditions, and it takes network time to communicate with the database, or even other external sources. **LSD** provides the ability to use futures in our transactions, instead of getting a concrete value (which can lock the data). The future is the promise that the database system will, on its discretion, resolve it to a concrete value. Because we are not using a concrete value, no lock happens right away, that is, other users can get a future to the same data, at the same time, thus increasing performance.

This future-based approach has some challenges:

- We want to provide the users with a similar programming interface to what we had before, so that it doesn't require a huge amount of codebase changes; and

- Because we are using futures instead of the concrete values, we must find a way so that users can perform transformations and conditions with these futures.

The first goal can be achieved by allowing users to do operations (i.e., transformations) between futures and static values, or even between futures and other futures. This is a feature that is already implemented in many languages, such as Java's **CompletableFuture** [18] or **ES6 Promises** [17], that allow for future composition.

In Kotlin, we could achieve this behavior elegantly by implementing a plus operation for our Future type, as demonstrated in Listing 2.1. Other languages would probably require us to have a method to create a function that allow us to subtract a concrete value from a future (e.g. $SUB(Future, 2)$).

Listing 2.1: Example of operator overloading in Kotlin

```
1    val a: Future<Int> = read(A)
2    val b: Future<String> = a + "␣-␣Hello␣World."
```

In order to achieve our second goal, we need some way to evaluate if a certain future respects a certain condition. The tricky part here is that we do not have access to the future, nor do we want to get the future's current value in our application. What we can do is ask the database whether the value that our future represents upholds a certain condition. The database checks if the current value of the Future (an observation) respects the condition, returning true or false to the application. Note that even if we are getting a concrete boolean value in our application, this value is made from the condition that we have specified, so the only thing that must be guaranteed by the database is that the condition stays with the same logical value, which means, we can have concurrent writes as long as the condition returns the same value, thus optimizing performance. Using the optimistic concurrency control to maintain our conditions (we want to maintain our invariants), is just like we would implement any algorithm with lock-free concurrency: we observe the current state without locking, and when we are ready to commit, we must check that the observed state is maintained and if so we can change the value (our compare-and-set).

2.8 Lazy-State Determination for K-V

Key-Value stores are a simpler method of storing data than row or column based databases. Whereas in a typical row-based database system we would have multiple columns, identified by a primary key (a subset of those columns), in a Key-Value store we simply have a key mapped (which can be seen as our primary key) to a value. These storage databases are designed to be very simple and performant, due to the fact that they are mainly used as cache, to optimize load times for users. Modern K-V stores also provide special support for value types such as JSON, where querying a **JSONPath** [9] stored in a specific key is optimized by the database system.

The current prototype for **LSD** in Key-Value stores [3] has three main components:

- Client — the client that submits transactions to be executed in the servers;
- Barrier — used when starting up the system and before running transactions; and
- Server — handles incoming transactions, and stores the changes, using an appropriate storage driver.

Figure 2.7 shows an overview of the prototyped solution for key-value stores.

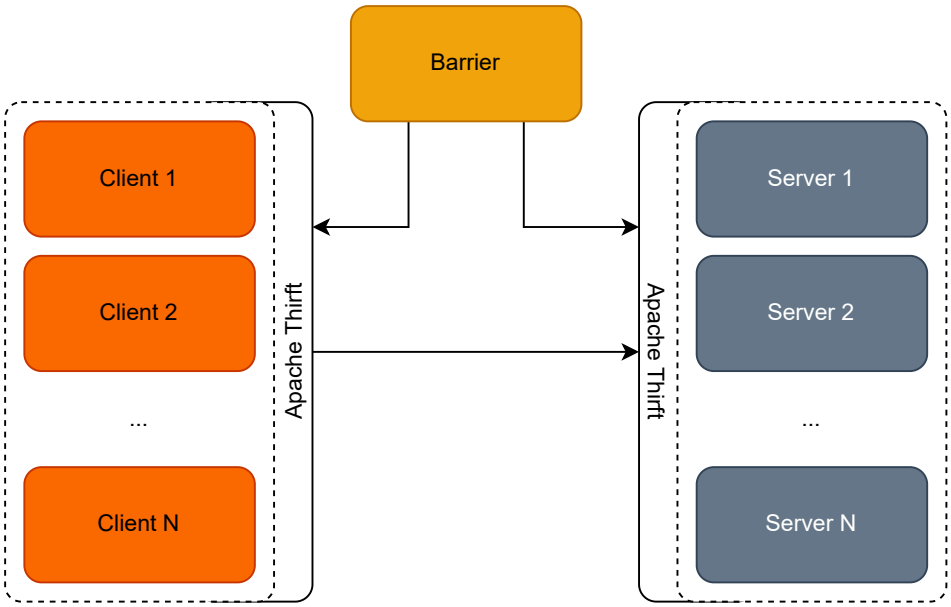


Figure 2.7: Overview of the prototype architecture

In the current prototype [3], the different components use **Apache Thrift** [2]. This prototype does not abstract the network layer, thus it is not possible to use another network protocol.

For the storage layer implementation, the current prototype abstracts a common interface that each storage driver must observe, which allows for different drivers to be compared in experimental evaluation. The implemented storage drivers are:

- **RocksDB** [16] — an embedded Key-Value storage developed at **Meta**; and
- A simple unordered hash map implementation.

The benchmarking component for our prototype consists of a **TPC-C** [23] fork, adapted to key-value stores, which can be run by executing a bash script, already present in the codebase.

Documentation for this prototype is scarce, or wrong, which presents trouble for beginners trying to benchmark or test the application. The prototype has still some old benchmark implementations, which makes it difficult to figure out how to properly run the latest benchmark implementation.

3

Lazy State Determination

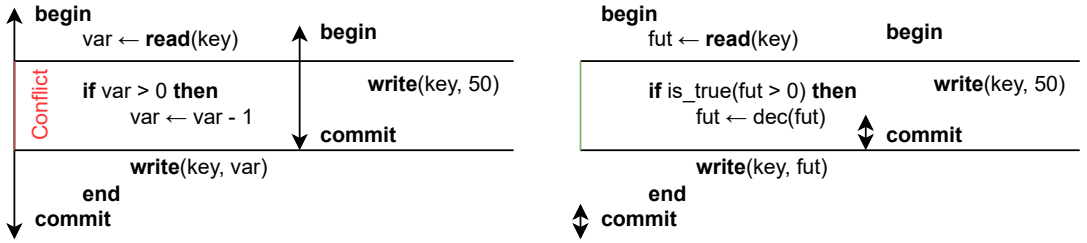
This chapter aims to give an introduction to the concepts of **LSD**, such as the existence of Futures and how they can be leveraged to our use-case. It also covers what problems we found with the original algorithm of **LSD**, why those problems matter, and what was the approach we took to solve those problems.

Lazy State Determination is a novel algorithm for handling transactions under high contention workloads. The algorithm works by delaying the fetch of concrete values from the underlying storage system, avoiding the need for locking during the transaction or the need for aborting when a value that was fetched during the transaction is no longer the same at commit time.

In order for **LSD** to delay getting the real value from the storage layer, and locking (or remembering) that value during the whole transaction window, we need to create a new entity that holds a promise to a value that lives (or not) in storage. As such, we present an abstraction layer called **Future**. A Future is nothing more than a promise to an entry in our storage layer. Whenever we want to get the concrete value that the Future holds, we can just follow through with that promise, that is, we can resolve our Future.

As with **LSD** we only need to get the concrete value when we really need it, we can reduce the transactional conflict window to the commit phase, as only during this phase we do actually need the real value. This however, introduces a problem, as we can no longer use values to perform decision branching or simple arithmetic and logic operations, since the only data we have available to us during a transaction are the futures we have read. As such, futures need a way to compose themselves, in order to allow users to use a future as they would use a concrete value (e.g. an integer). Moreover, performing decision branching also requires us to implement some sort of future speculation, to the likes of Optimistic Concurrency Control, where we read values from storage without locking in the transactional context, but when we commit the transaction we actually need to check that those values are still the same. Figure 3.1 shows how **LSD** reduces the conflict window of a transaction versus other approaches to concurrency control.

Since **LSD** uses futures to represent a promise to the actual data, we are no longer able to do certain operations like we used to in the past. For instance, we can no longer



(a) Conflict with normal transaction

(b) No conflicts by using LSD

Figure 3.1: Reducing the conflict window with Lazy State Determination

write a value to the storage using the storage API, as we don't have the concrete value in our possession, but rather a promise for that value. To counter this, the **LSD** specification defines special functions, or methods, that must be implemented in order to make use of these promises, as seen in Table 3.1.

Table 3.1: Lazy State Determination Interface

Operation	**Description**
begin()	Starts a new transaction
read(key) → Δ	Reads a key (concrete value), returning a Future
read(Δ) → □	Reads a key Δ that is a future, returning another Future □
write(key, Δ)	Writes the value represented by the future Δ to a concrete key
write(Δ, □)	Writes the value represented by the future □ to the key represented by Future □
is_true(Δ) → boolean	Evaluates if the future represented by Δ represents a boolean set to true
commit() → boolean	Commits the current transaction
abort()	Aborts the current transaction

The first thing we notice when looking at the original Lazy State Determination algorithm is that there are multiple methods to handle the combinations of futures and concrete values, for instance, with the `read` and `write` functions. The original algorithm also has no way to write a concrete value to a concrete key, as we always need at least one Future (the value).

One other thing that we can notice immediately looking at the original specification is that we have a special function called `is_true`. The purpose of this function is so that users can perform decision branching in a transactional context. Due to the fact that every concrete value is handled behind our Future abstraction, we need a way to check if a condition is true or not. To this end, this function speculates the current value of the condition, and as long as the condition upholds during the commit phase, the transaction does not need to abort. An advantage of `is_true` is that only the value of the condition needs to be maintained for the duration of the transaction for the transaction to proceed, and not the concrete values. Even if a value has changed while the transaction was ongoing, if the condition still resolves to the same value (i.e. true or false), the transaction

can proceed.

During the course of our work we have observed several pitfalls of the original Lazy State Determination algorithm. Examples of such pitfalls include, for instance, the inability to perform read-after-writes in the same transaction. With the old algorithm, a key that was written in one transaction does not affect subsequent reads of the same key, in the same transaction. These drawbacks prevent **LSD** from being used in any real world application, as it completely misses out on certain functionality that a real world user would require for their transactional workload. Additionally, one of the ideals of **LSD** is to provide users with a similar interface, so that migrating from a traditional transaction to Lazy State Determination is easy. As such, **LSD** should offer everything that traditional transaction systems already offer.

3.1 Future Composition

Future composition is an essential part of Lazy State Determination which allows its users to actually make use of Futures and concrete values together. If we think about **LSD** and its operations, as depicted on Table 3.1, without having any sort of support for composition, we can see we are only able to create new Futures through writing concrete values, and reading them later on. As such, if we want to create something new, using concrete values or other futures as sources, we must provide a way for users to compose futures.

We can imagine a transaction like the one depicted in Figure 3.2a, where we want to read a future from our storage layer, using the provided **LSD** operations, and we want to increment that future and put it back in our storage layer. To perform this operation we require composition, as incrementing a Future cannot be performed like incrementing a primitive integer. Figure 3.2b shows that, within our Future, we must store the operations that affect that Future so that a composed future can be resolved later on during the commit phase, or even during a speculation.

```
begin
 Δ ← read(key)
 Δ ← Δ + 1
 write(key, Δ)
commit
```

(a) Example of transaction that composes a future with a concrete value

```
Δ {
 key,
 operation: {
  type: +,
  operand: 1
 }
}
```

(b) Future composition created by the transaction

Figure 3.2: Example of Future composition in Lazy State Determination

As we can observe in Figure 3.2b, in order to support composition, our Future holds both the operation, in this case an addition, and the other operand it was composed

with, which for the transaction shown in Figure 3.2a, that operand is a concrete value representing the integer one.

Additionally, by allowing our Futures to contain references to other Futures during composition, we are able to compose multiple futures together, just like we can compose primitive values. The ability to perform composition is fundamental to make **LSD** viable in a real world scenario, where it is often that we get a value from our database, we apply some transformation over that value, and then we write the result back to the database.

In order to support the resolution of a composed future, the resolution algorithm must, for each future to be resolved, also resolve the composition by applying the operation and the operands to obtain the resulting value. The resolution for composed futures is done in two steps:

1. Resolve both operands of the composed Future; and
2. Apply the operation between the resolved operands concrete values.

This means that, for composed futures, the resolution is recursive, calling the resolution of each operand until the maximum depth of the future is reached.

3.2 Reads After Writes

As stated previously, the current commit algorithm for Lazy State Determination presented by Vale [25] does not allow transactional users to read a key that was previously written in the same transaction. In this section we will demonstrate where the issue lies, so that it can be fixed later on.

Algorithm 1 Lazy State Determination Commit Algorithm

$rvalues \leftarrow \emptyset$
$result \leftarrow \top$
for $\langle key, - \rangle \in wset$ **do**
    $lock(key)$
end for
for $\square \in frset$ **do**
    $key \leftarrow key(\square)$
    lock(key)
    $rvalues \leftarrow rvalues \cup \{\langle key, get(key) \rangle\}$
end for
for $\langle \Delta, \square \rangle \in fwset$ **do**
    $key \leftarrow resolve(\Delta, rvalues)$
    $lock(key)$
    $wset \leftarrow wset \cup \{\langle key, \square \rangle\}$
end for

```
for ⟨key, version⟩ ∈ rset do
    if version ≠ version(key) then
        result ← ⊥
    end if
end for
for ⟨□, expected⟩ ∈ cset do
    value ← resolve(□, rvalues)
    if value ≠ expected then
        result ← ⊥
    end if
end for
if result = ⊤ then
    for ⟨key, □⟩ ∈ wset do
        value ← resolve(□, rvalues)
        version ← nextversion(key)
        put(key, value, version)
    end for
end if
for key ∈ wset ∪ keys(frset) do
    unlock(key)
end for
return ⟨result, rvalues⟩
```

Algorithm 1 presents the commit-phase algorithm of the **LSD** specification using Optimistic Concurrency Control. This algorithm can be divided in 7 sections:

1. It locks the keys present in the write set ($wset$). The write set represents the writes which were made to concrete value key with a future value;

2. For each future value in the future read set ($frset$) we obtain the key for which the future was created for, we lock this key and then add the key-value pair to the read values ($rvalues$), obtaining the value associated with this key from the storage layer;

3. For each future key-value pair in the future write set ($fwset$) we must resolve the future key using the read values ($rvalues$). These $rvalues$ contain all previously read values in the same transaction, thus if a future key was passed, it must have been read in the same transaction before the write operation. With our key resolved, we can lock it, and add the concrete key and the future value to the write set ($wset$) to be handled later on;

4. For each key read from the storage during the transaction ($rset$) we check that the version we got initially is still the latest version for this key. Otherwise, if there is a version mismatch, we cannot continue this transaction, and we must abort;

5. For all future value-boolean pairs in the compare set ($cset$), we must resolve the future value with $rvalues$, using the same strategy as we did for resolving the $fwset$. Once we resolved the value, we must check that it is the same as the value we first observed during our transaction, and, if not, we must abort.

6. If we observe that the transaction can continue, we can take all our writes present in the write set ($wset$), and for each of them we can resolve the future value using the values we have read during our transaction ($rvalues$), and insert into our storage the key-value pair.

7. We finish our transaction always by unlocking all the keys we have locked previously, even in the case of a transaction abort.

This means that, for a value written during our transaction, it will end up eventually in the write set used by the **LSD** commit algorithm. Unfortunately, this write set is only observed when writing keys to the storage layer, and not when reading keys from storage during the transaction. As can be seen in Algorithm 1, the reads are resolved, and obtained from the storage layer in the second phase of the algorithm, however the writes in $wset$ (and by extent, $fwset$) are only observed well after that, when we confirm that our transaction is not to be aborted. This means that if we try to do a read-after-write in the course of a transactional context, our read will only ever observe values that have been already materialized to a storage layer.

By not allowing read-after-writes in transactions, we seriously hinder our Lazy State Determination algorithm from ever being used in a real-world scenario, where workloads that perform this type of operation are common.

4

Improving Lazy State Determination

The **LSD** algorithm presented in [25] has some caveats that prevent it from being applicable in a real world scenario. As stated before, we can immediately identify some of these:

- We require multiple functions to combine futures and concrete values.
- There is a need for complementary functions to perform operations over futures. If we want to perform a boolean check over a future we need to call `is_true`, or, if we want to increment an integer future, we need to have a function to perform that, as we cannot do `value + 1`.
- Currently, **LSD** does not support read-after-write operations, therefore, if we write a value in a transaction, our next read of that same write in the same transaction will return the data at the start of that transaction (i.e., before the write was made).

As such, improving the current Lazy State Determination prototype also requires us to pick a language that supports solving those problems. Even though we could implement a Lazy State Determination prototype in Go for example, it would not be possible for us to eliminate the need for complementary functions, as this language does not offer support for operator overloading.

The current implementation language C++, which offers great performance, and also supports operator overloading. However, the drawbacks are that it is not an easy language to learn, with multiple pitfalls, namely regarding memory management, that can difficult and compromise the development of projects that must leverage concurrency. Due to these reasons, we have chosen **Rust** as the language for improving the current prototype of Lazy State Determinations.

Even though the learning curve of **Rust** is still high, it offers more safeties in terms of memory management and sharing data across multiple threads (or tasks). For instance, **Rust** natively implements traits such as `Sync` and `Send`, that prevent us from sending or sharing non-thread safe data across threads. Moreover, this language also offers the

performance that is necessary for this type of application, operator overloading, a feature-rich macro system, and a thriving ecosystem, that have contributed to improving the current prototype.

In the next sections we present how we have improved the existing **LSD** prototype, by looking at every aspect that we improved on.

4.1 Future

A **Future** represents a promise of a value that is to be resolved, or an already resolved value. By this definition we can already solve one of the problems stated above: we don't need multiple methods to handle combinations of futures and concrete values, since concrete values can live inside futures.

Our futures can be resolved, and provider a mechanism to trigger the resolution Resolving a **Future** is the operation of asking the provider of the promise to fulfill that promise, returning a value. Futures can also be temporarily resolved, that is, the value of the **Future** can be **speculated**. Speculating a Future value is different from resolving it, as there's no guarantee by the resolver that the speculated value will be the same for the whole duration of the transactional context, as another transaction might win the rights to change the underlying concrete value first.

Futures must also provide its users a mechanism for composition, allowing users to perform arithmetic or logic operation between two futures, or even between a future and a concrete value. To this end, a Future must also store a operator and another operand. Hence, we can think of Futures as a data structure like the one represented in listing 4.1.

Listing 4.1: Representation of a Future in **Rust**

```
struct Future {
    lhs: FutureValue
    rhs: Option<Operation>
}

enum FutureValue {
    Resolved(Value),
    Future(Key)
}

enum Operation {
    Add(FutureValue),
    Sub(FutureValue),
    ...
}
```

A future is a simple data structure that only holds information related to how the future can be solved. The resolution of the future is dependent to the system where the future is used. For **LSD** the future is solved by the transaction on commit, and can be speculated at any time by calling a speculate operation during a transactional context, that will interrogate the underlying storage regarding the value of the future at that point in the transaction. This completely decouples the data and logic functions, as Futures are nothing more than a data abstraction.

This representation of a Future also provides a significant advantage over the original specification — we no longer need extra functions to be able to mix futures and concrete values, as a Future can already be resolved to a concrete value. It massively simplifies the user interface, and provides the user with possible combinations that could not be achieved with the original implementation, such as writing a concrete value to a concrete key.

The resolution of the future is identical to the speculation of the value of a future, except that for speculated values, the concurrency control mechanism needs to keep the value that was speculated so that if this value is later used for decision branching, the concurrency control mechanism can abort the transaction if there is a drift that prevents the transaction from continuing.

4.2 Using Concrete Values

As we have stated before, one of the drawbacks of the old **LSD** implementation was combining operations containing both concretes and futures, which would require us to implement and use the most appropriate function for the combination we required.

In order to avoid having multiple combinations of our functions, we can simply use our created Future data structure to store concrete values as well. If we pretend that a concrete value is nothing but an already resolved Future, we can store our concrete values as resolved futures. When resolving this particular future, the resolver will not have to do anything, besides returning the already resolved value that is contained inside that future.

This however, poses a new problem, because we need a way to convert a concrete value into a Future data structure. In **Rust**, the best solution that can be achieved for this problem is through the **From/Into** traits, that allow us to convert one type to another. By implementing the `From<i32>` trait for our Future struct, we can do `Future::from(value)` where the value is a 32-bit integer. Additionally, because all **From** traits also implement **Into**, we can do `value.into()` when passing the value as a Future parameter.

The approach we have chosen allow us to convert between types with ease, however implementing the actual conversions can become a bit tiresome for maintainers. Just accounting for integers, in **Rust** there are 12 types that can represent an integer, ranging from 8-bit to 128-bit integers, and signed versus unsigned. To solve this problem we have looked into **Rust**'s macros to make these implementations more automatized for us. If we

look into an actual implementation for converting signed 64-bit integers to Futures, all we have to do is make that value the resolved value of the future, as shown in Listing 4.2. To simplify the implementation for all types we can implement a macro that creates these common implementations for every type we want, as specified in Listing 4.3.

Listing 4.2: Converting concrete values using From in **Rust**

```
impl From<u64> for Future<u64> {
    fn from(value: u64) -> Future<u64> {
        Future::from(crate::types::proto::Future {
            fut_id: 0,
            rhs: Some(Box::from(FutureStatus {
                status: Some(Status::Resolved(Value {
                    r#type: ValueType::U64 as i32,
                    data: value.encode_to_vec(),
                })),
            })),
            operation: None,
        })
    }
}
```

Listing 4.3: Using macros in **Rust** to simplify implementing conversions for all types

```
macro_rules! into_future {
    ($(($t:ty,$vt:ident))*) => (
        $(
            impl From<$t> for Future<$t> {
                fn from(value: $t) -> Future<$t> {
                    Future::from(crate::types::proto::Future {
                        fut_id: 0,
                        rhs: Some(Box::from(FutureStatus {
                            status: Some(Status::Resolved(Value {
                                r#type: ValueType::$vt as i32,
                                data: value.encode_to_vec(),
                            })),
                        })),
                        operation: None,
                    })
                }
            }
        )*
    );
```

```
20      }
21
22      # Using it is as simple as
23      into_future! {
24          (u8, U8)
25          (u16, U16)
26          (u32, U32)
27          (u64, U64)
28      }
```

4.3 Resolving Futures

The resolution of a Future is a responsibility of the system that is accepting those futures. In our use-case, the one responsible for solving all futures is the **LSD** transactional context. We need to be able to resolve futures in two different cases:

1. Speculating the current value of a Future; and
2. Resolving the Future to its final value in a particular transactional context.

Future resolution can be achieved by performing at most three operations:

1. Resolving the value of the future; and
2. If the future has an operation and another operand:
    a) Resolving the other operand (a future); and
    b) Computing the final result of the operation, by applying the operation on both of the resolved concrete values.

In order to resolve a future, all futures must take into account its dependencies, which are always other futures. When resolving a future we start by resolving its dependencies using a depth-first approach, that is, we start by solving the last dependency in our chain, and start applying operations and solving the dependencies so that we are able to solve the future we want. Figure 4.1 demonstrates how this future resolution works using the depth-first approach.

In Figure 4.1, when we want to solve Future `0`, we need to resolve the dependencies first, and, in this case, we start by resolving Future `2`. Since Future `2` also has dependencies, we must solve them first, which means we need to resolve Future `3` and Future `1`, and apply the operation between these two futures to compute Future `2`. With Future `2` computed, we can go back to Future `0` and use the resolution we had already done for Future `1` to do the operation between Future `1` and Future `2`, that leads to Future `0`.

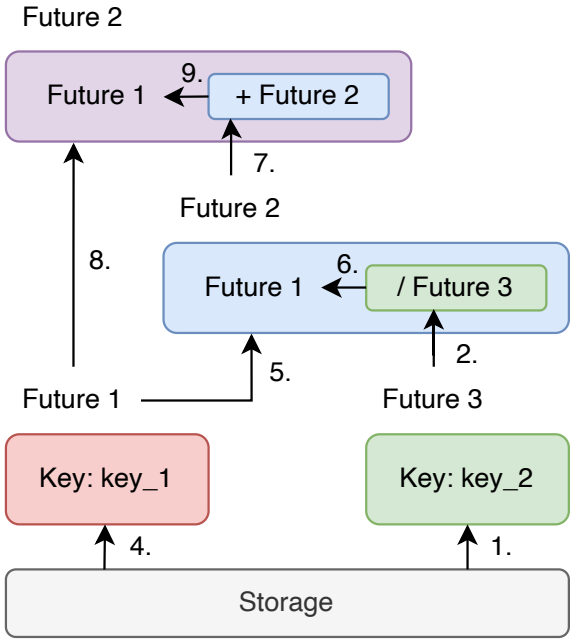


Figure 4.1: Depth-First resolution of Futures

In Algorithm 2 we propose a solution for resolving **LSD** futures. This solution implements a depth-first approach using recursion, where it undergoes, at most, three phases:

1. Resolve the right-hand side of the Future, using a depth-first approach. This side of the future always exists, even if there is no operation (i.e., this future was not composed with another). If we find that the right-hand side is already resolved, we can use the resolved value directly, skipping the need for the depth-first resolution.
2. If the future contains an operation:
   a) We solve the left-hand side of the future, using the same approach we used for the right-hand side.
   b) Apply the operation between the two operands.

Algorithm 2 New Lazy State Determination Algorithm for Resolving Futures

```
function RESOLVE(Δ, resolved)
    if Δ ∈ resolved then
        return resolved[Δ]
    end if
    result ← ⊥
    rhs ← ⊥
    if state(RightHandSide(Δ)) = resolved then
        rhs ← RightHandSide(Δ)
    else
```

```
        rhs ← Resolve(RightHandSide(Δ), resolved)
    end if
    if HasOperation(Δ) then
        operation ← operation(Δ)
        lhs ← ⊥
        if state(LeftHandSide(Δ)) = resolved then
            lhs ← LeftHandSide(Δ)
        else
            lhs ← Resolve(LeftHandSide(Δ), resolved)
        end if
        result ← ApplyOperation(lhs, operation, rhs)
    else
        result ← rhs
    end if
    return result
end function
```

4.4 Allowing for Read After Writes

To allow for read after writes, we need a commit algorithm that is able to perform all the writes to a temporary environment, instead of immediately materializing the writes at the end, like the original **LSD** algorithm. For this temporary environment, we can hold our key-value pairs in a local HashMap. Since this HashMap is local to the commit algorithm, we don't incur in additional concurrency overhead.

When we want to commit a transaction, we can simply push our temporary environment, which contains all the writes that were performed during the transaction to the underlying storage layer. To make sure that we have an atomic guarantee for our writes (i.e., all writes succeed or none do) we can put all writes in a batch, and send this batch to the storage layer. That way, it is the responsability of the storage layer to guarantee that a batch is always handled atomically, since the actual implementation could change depending on the underlying storage.

This temporary environment is implemented as an operations list. Every operation we perform during a transaction (e.g., *read*, *write*, ...) is recorded in an ordered list of operations that belongs to the transaction where they were performed. To find the current state of our environment, that is, to find the writes that have been performed by the current transaction, we must compile all the operations that have been added to the list, by the order they were added to this list. Compiling the operations will provide information about:

- If the transaction is to be aborted due to a drift in a speculated value.
- What are the writes that must be materialized to storage at the end of the transaction.

- What futures were resolved during our compilation.

An ordered list is required to hold all the operations performed during a transaction, as if we start storing operations in an arbitrary order we lose the information required to perform read-after-write operations and future speculation.

Figure 4.2 demonstrates the concept behind this list. When new operations are performed, these are pushed to the end of the list to be processed later on. Whenever we want to figure out the current state of the transaction, we can iterate over this list, and compute the current state by applying in a temporary environment the operations by the order they were executed.

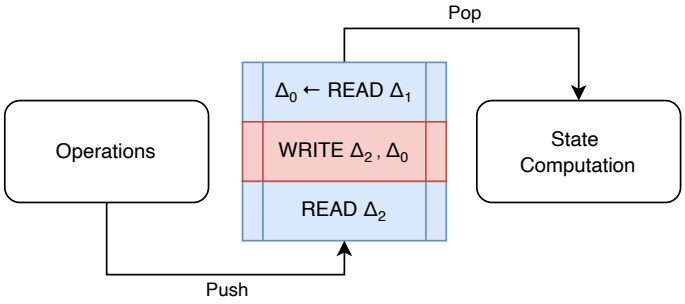


Figure 4.2: Operations list as the transaction state holder

The process of analyzing the list of operations and figuring out what the current state is, is explained in Algorithm 3, which specifies how, for a list of operations, they should be transformed to find the current state of the transaction. The specific process for each type of operation is explained as follows:

- For a *read* operation we resolve the future key that was provided when executing this operation. The resolution of the future key is done via the algorithm presented previously for future resolution. Once we have the concrete value for the key we want to read, we must lock it, to guarantee that no one is reading from it at the same time. Once locked, we must evaluate if the key is in the list of writes we computed previously with our operations list, and, if it is, it means we are under a read-after-write scenario, and we should read the value from the previous write in the same transaction. If we are not under a read-after-write scenario, we check if this key has been read before in the same transaction, so that we can return the same value if it is, which guarantees *Repeatable Reads*. As a last resort, if the key was never encountered before, we must retrieve it from storage, and add it to the reads and resolved futures map.

- For a *write* operation we must resolve the key and value futures that were provided by this operation. Once resolved, we must lock the key, so that no one else can access it. After this, we can simply add our key to the writes map, so that it can be used to solve for read-after-write scenarios in the same transaction. The reads map is

also returned if this transaction is allowed to commit, after all operations have been applied.

- For $speculate$ operations, we need to check if the speculated value is still the current one in storage. In order to do this, we resolve the value that was speculated previously, and we check whether the return value is still the same as the one that was speculated. If not, it returns an error indicating that the transaction is to be aborted.

Algorithm 3 New Lazy State Determination Algorithm for Computing Operations

function ComputeOperations
  $writes \leftarrow \emptyset$
  $reads \leftarrow \emptyset$
  $resolved \leftarrow \emptyset$
  for $operation \in operations$ **do**
    if $type(operation) = read$ **then**
      $fut\_key \leftarrow get\_deferred(future(operation))$
      $key \leftarrow resolve(fut\_key, resolved)$
      $resolved \leftarrow resolved \cup \{\langle fut\_key, key\rangle\}$
      $lock(key)$
      if $key \in writes$ **then**
        $value \leftarrow writes[key]$
        $resolved \leftarrow resolved \cup \{\langle future(operation), value\rangle\}$
      else
        if $key \in reads$ **then**
          $value \leftarrow reads[key]$
          $resolved \leftarrow resolved \cup \{\langle future(operation), value\rangle\}$
        else
          $value \leftarrow get(key)$
          $reads \leftarrow reads \cup \{\langle key, value\rangle\}$
          $resolved \leftarrow resolved \cup \{\langle future(operation), value\rangle\}$
        end if
      end if
    else if $type(operation) = write$ **then**
      $key \leftarrow resolve(future\_key(operation), resolved)$
      $value \leftarrow resolve(future\_value(operation), resolved)$
      $resolved \leftarrow resolved \cup \{\langle future\_key(operation), key\rangle\}$
      $resolved \leftarrow resolved \cup \{\langle future\_value(operation), value\rangle\}$
      $lock(key)$
      $writes \leftarrow writes \cup \{\langle key, value\rangle\}$
    else if $type(operation) = speculate$ **then**

```
            value ← resolve(future_value(operation), resolved)
            resolved ← resolved ∪ {⟨future_value(operation), value⟩}
            if speculated(operation) ≠ value then
                return Error(Aborted)
            end if
        end if
    end for
    return Ok(⟨writes, resolved⟩)
end function
```

4.5 Operations

For the new **LSD** specification we improved the operations, reducing the need for having multiple operations to handle combinations of futures and concrete values. Due to the fact that we have established that a concrete value can be wrapped in a resolved Future, we can cut operations from the original **LSD** specification that were used solely for the purpose of handling concrete values. This is shown in Table 4.1.

Table 4.1: New Lazy State Determination Interface

Operation	Description
begin()	Starts a new transaction
read(Δ) → □	Reads a key Δ that is a future, returning another Future □
write(Δ, □)	Writes the value represented by the future □ to the key represented by Future □
speculate(Δ) → Value	Speculates the current value of the future represented by Δ
commit() → boolean	Commits the current transaction
abort()	Aborts the current transaction

Another difference in our new operations present in Table 4.1, comparing to the original **LSD** specification, is the fact that there is no longer the *is_true* operation that used to exist. Instead, this operation has been replaced by the *speculate* operation, which is more flexible than its previous counterpart as it is now usable for every data type supported by Lazy State Determination, and not just booleans.

We will cover the *read*, *write*, *speculate* and *commit* operations in more depth, as these operations provide the backbone for Lazy State Determination.

4.5.1 Read Operation

The read operation allows users to read a key from the storage layer during the course of a transaction. The algorithm for this operation is shown in Algorithm 4, where we can observe that this operation does two things:

1. It creates a Future that consumes from our Future Key (the future that was provided by the user to the operation). As stated before, Futures are either Resolved, or they are Deferred. When the futures are Deferred, they take a Future that acts as the key from where to get (i.e., resolve) the value from the storage layer.
2. We insert the future to our operations map, where the type of the operation corresponds to *read*.

Algorithm 4 New Lazy State Determination Algorithm for Reading Data

function READ(Δ)
  $\square \leftarrow FutureFromKey(\Delta)$
  $operations \leftarrow operations \cup \{\langle "read", \square \rangle\}$
end function

4.5.2 Write Operation

The write operation is the simplest operation in **LSD**, as all we have to do is to insert the key and values futures provided by the users to the operations map, as demonstrated in Algorithm 5.

Algorithm 5 New Lazy State Determination Algorithm for Writing

function WRITE($\Delta, \square$)
  $operations \leftarrow operations \cup \{\langle "write", \Delta, \square \rangle\}$
end function

4.5.3 Speculate Operation

For speculating the current value of a future during a transactional context, the users must provide **LSD** a Future to speculate on. With this operation we must resolve the future, just like we have to do during the commit phase using the *Resolve* and *ComputeOperations* functions presented in Algorithms Algorithms 2 and 3, respectively. The difference between a speculation and a commit lies in the fact that the speculation does not materialize anything, nor does it abort a ongoing transaction in any circumstance. It only makes use of the current state of the transaction, that is, the current operations that have been performed in this transaction, as a way to properly compute what the concrete value promised by the future is.

The algorithm for the speculation of futures is shown in Algorithm 6, where we can observe that, in a first step, we resolve the future that was provided to us, and in a second phase we add the result of the speculation, along with the future that we speculated on, to our list of operations. Adding these results to the list of operations is essential to determine whether to allow a transaction to proceed, or abort it, during the commit phase. If the speculated value mismatches with the value computed during the commit phase, the transaction has drifted from the current state of the database, and so it must be aborted.

Algorithm 6 New Lazy State Determination Algorithm for Speculating Futures

function SPECULATE(Δ)
  $resolved \leftarrow \emptyset$
  $speculated \leftarrow Resolve(\Delta, resolved)$
  $operations \leftarrow operations \cup \{\langle "speculate", \Delta, speculated \rangle\}$
end function

4.5.4 Commit Operation

The commit operation has the goal of taking every operation that was performed during a transaction, and materialize that transaction if possible to our storage media. As such, the commit operation also makes the use of the $ComputeOperations$ algorithm, as presented in Algorithm 3, to compute the final state of the operations performed during the transaction determining either:

- If the operation was successful, which writes must we materialize to our storage layer; or
- If the operation was aborted, why was it aborted.

If our operations can proceed to their materialization, we must observe the Atomicity property of the transaction as specified by the ACID Model [12], which we are observing for Lazy State Determination. To guarantee atomicity for the writes performed during a transactional context, we rely on the storage layer, as this layer is the one responsible for sending the writes we have done to the storage media, either with being a disk, an API, or another one. Only the storage layer is capable of understanding how atomicity works for the underlying storage media. Due to this fact, every storage layer must provide an interface to perform the insertion of batches of data. Every $Batch$ is guaranteed, by the storage layer, to obey to the atomicity and durability aspects of the ACID specification.

Algorithm 7 proposes a new algorithm for performing commits in a Lazy State Determination environment. This commit starts by computing the necessary operations, evaluating if it must abort or proceed. If the commit proceeds the next step it must perform is to go through every written key and find what is the next version for that key on the storage layer. Once it has the version for every key we want to write to, we can add the $\langle key, value, version \rangle$ batch to our list of batches, to be handled by the storage layer.

Algorithm 7 New Lazy State Determination Algorithm for Committing Transactions

function COMMIT
  $result \leftarrow ComputeOperations()$
  if $type(result) = Error$ **then**
    $UnlockAll()$
    return result
  end if

```
    ⟨writes, −⟩ ← result
    batch ← ∅
    for ⟨key, value⟩ ∈ writes do
        version ← version(key)
        batch ← batch ∪ {⟨key, value, version⟩}
    end for
    response ← InsertBatch(batch)
    UnlockAll()
    return response
end function
```

5 Evaluation

In this chapter we will mention what was done to try to evaluate the proposed implementation of **LSD**.

To guarantee that we have representative values, we must analyze our implementation under various circumstances. As such, we explore how we created abstractions to deal with different storage layers and different networking protocols.

We will also present on how we planned to measure the performance of our system using the **TPC-C** specification. This specification was used in previous versions of the **LSD** prototype, which allows us to have a base of comparison with those implementations.

5.1 Storage

As with the previous prototype of **LSD**, we still want to provide users with the ability to choose and pick which storage layer they want. In order to do this, we must provide an abstraction that enables **LSD** users to create a new storage backend if they wish.

A storage driver must allow for users to:

1. Get and Modify a specific key, guaranteeing that the key can be only modified from one thread at a time.

2. Performing writes on multiple keys at a time, atomically.

Firstly, we need to define what operations should a normal, non-thread safe key should support. To this end, we have defined a `StoredKey` trait, that driver implementations must observe, as defined in Listing 5.1.

Listing 5.1: The StoredKey trait for **LSD** in **Rust**

```
1    pub trait StoredKey: Debug + Send + Clone + 'static {
2        fn get_key(&self) -> String;
3        fn get(&self) -> LsdResult<(Value, u64)>;
4        fn get_version(&self) -> u64;
5        fn put(&mut self, value: Value) -> LsdResult<()>;
```

```
    }
```

With a trait for key-value pairs that are to be stored on the storage backend, we defined a common struct that implements a locked version of that key, signalling that the key is currently being used by a thread or another process. As such, we have defined the `LockedKey` struct, that is nothing but a wrapper around **Rust**'s `MutexGuards` for `StoredKeys`, as shown in Listing 5.2. A `MutexGuard` is the **Rust** accessor for the data behind a locked `Mutex`. When locked a `Mutex` returns a guard that allow us to access the data that lives inside the `Mutex`, and, when the guard variable is dropped from the current execution, the `Mutex` is automatically unlocked again, due to the automatic `Drop` trait implemented for `MutexGuards`. By using a wrapper around a `MutexGuard` we can guarantee that:

1. The guard is always created from a `Mutex` that represents a `StoredKey`; and
2. When the `LockedKey` is dropped out of execution (e.g., the variable is no longer used), the underlying `MutexGuard` is also dropped automatically, releasing the lock on the `Mutex`.

Listing 5.2: The LockedKey trait for **LSD** in **Rust**

```
pub struct LockedKey<T>
where
    T: StoredKey,
{
    inner: MutexGuard<'static, T>,
}
```

It should be noted that the static lifetime present in Listing 5.2, actually represents the lifetime of the reference that the `MutexGuard` has to the originating `Mutex`. Since we will keep our `Mutexes` alive for the remainder of execution, as our storage layer doesn't implement a mechanism for key eviction, we can safely use a static lifetime.

With these data structure and traits, a storage driver ultimately only needs to implement the `StorageDriver` trait, as it is defined in Listing 5.3. This trait requires drivers to implement two operations: `lock` that tries to obtain the lock on a `StoredKey`, returning the `LockedKey`, and a `send_batch` operation, that allows users of the storage layer to send a batch of modifications to be performed atomically to a number of `LockedKeys`.

Listing 5.3: The StorageDriver trait for **LSD** in **Rust**

```
#[async_trait]
pub trait StorageDriver<T>
where
    T: StoredKey,
{
    async fn lock(&mut self, key: String)
```

```
7               -> LsdResult<LockedKey<T>>;
8           async fn send_batch(&self, batch: Batch<T>)
9               -> LsdResult<()>;
10      }
```

In order to obtain a better comparison across the experimental results, we implemented two different storage mechanisms for the **LSD** prototype, using the modular storage architecture that was previously presented. The two storage providers are:

1. A **Rust HashMap** [20]. This provider is in-memory only and does not keep any data on the disk.

2. **RocksDB** [16], which was the storage solution already in use by the previous implementation. We decided to keep **RocksDB** as it allow us to have a measure of comparison with how previous implementations performed.

By having two different storage implementations, one that stores data in-memory, and another one that uses a disk, we can measure how I/O impacts the workloads using **LSD**. By having a disk provider we are also able to test the performance impact that using a Hard Disk Drive brings, in comparison to using a Solid State Drive.

5.2 Network Protocol

The original prototype for **LSD** Key-Value [3] only supports **Apache Thrift** as its networking layer, and it doesn't introduce any abstraction to allow developers to implement any other network layer they prefer. In our implementation we aim to provide an abstraction over the network layer. This abstraction should allow any developer to implement their network layer, using the core components of **LSD** K-V for the managing each transactions and its associated futures.

For our prototype we have opted to use **gRPC** to demonstrated how a network layer could be implemented on top of the core **LSD** components. **gRPC** was chosen because of its ease of use, when compared to **Apache Thrift**, the use of Google's Protobuf's [8], and its popularity, which makes it easy to search for online resources regarding **gRPC**. This network layer has been implemented in the `lsd-grpc` package, which makes use of the `lsd-core` package to manage the transactions.

The **gRPC** protocol is defined in the `lsd-grpc` package and is used to generate the client and server types and abstractions, that let us implement a compatible **gRPC** clients and servers. These implementations are generated using the **tonic Rust** crate (using the **prost**), and are generated by the means of a compile time hook, that when building the application, generates the necessary **Rust** code from the defined Protobuf's file, as we can observe in Listing 5.4.

Listing 5.4: Protobuf Code Generation at Compile-Time

```rust
fn main() -> Result<(), Box<dyn std::error::Error>> {
    tonic_build::compile_protos("proto/lsd.proto")?;
    Ok(())
}
```

We have implemented four different **LSD** servers, with different combinations of the storage layers and the concurrency control mechanisms (e.g., **LSD** with Optimistic Concurrency Control, Only Optimistic Concurrency Control, ...). We also provide a client library, that is capable of communicating with any server that uses the **gRPC** protocol. The clients are responsible to execute operations, as defined for our Key-Value store. These operations, when requested by the client, are executed on the server by using a Remote Procedure Call, using **gRPC**. For each operation, the common protocol defines the list of necessary parameters for each RPC, and it's up to the client application to pass these parameters to the server. On the other hand, the servers are responsible to manage transactions, keeping a record of the active transactions, so that whenever a client interacts with a specific transaction, the server can look up whether the transaction that the client passed is valid or not. If the transaction is still active, the server will execute the requested operation on behalf of the user, for the transaction that was specified in the requests. Figure 5.1 depicts the communication that happens when a client tries to execute a simple transaction.

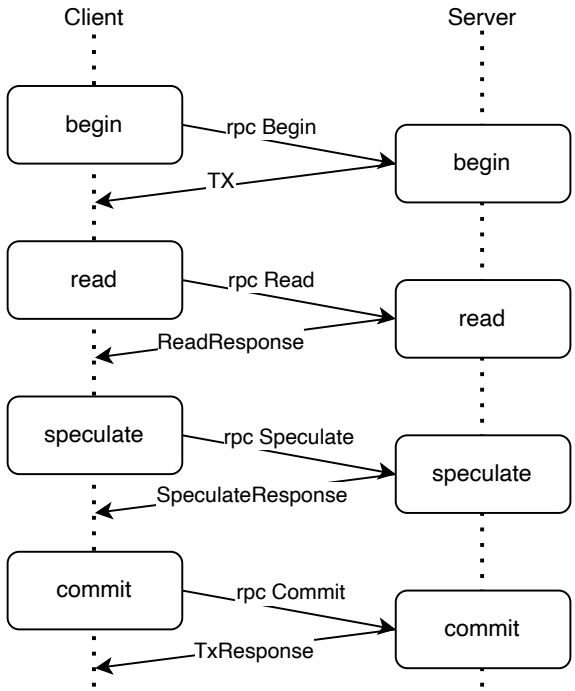


Figure 5.1: Communication Diagram between Client and Server using **gRPC**

5.3 TPC-C Benchmarks

In order to obtain experimental results for our work, we have chosen the **TPC-C** [23] benchmark. This benchmark was already used for work done previously by Vale [25] and Carpinteiro [3], so it also provides a way for us to compare what has improved, and what still needs improving, between these implementations of **LSD**, and between the modules that make this implementation.

Since we chose **Rust** as the language of choice for our implementation, we were unable to use previously chosen **TPC-C** implementations. Unfortunately, as the **Rust** ecosystem did not have any **TPC-C** implementation available at the time of our work, we were left with two options: implement the **TPC-C** specification in **Rust** or implement a foreign function interface that would let us leverage the C++ implementation. Since the C++ implementation would require some tweaks to be able to work with the new Futures we have introduced in our implementation, we have chosen to implement the **TPC-C** specification in **Rust** ourselves.

The **TPC-C** benchmarks are based upon a real-world scenario of a company working in the wholesaler market. This company has several warehouses around the globe to serve its customers. Each warehouse can be responsible for multiple districts around its location. All the products that this company sells to customers are stored in their warehouses, so the company must keep track of the stock of each product, given the number of orders for each product. The database entities and relationships are depicted in Figure 5.2.

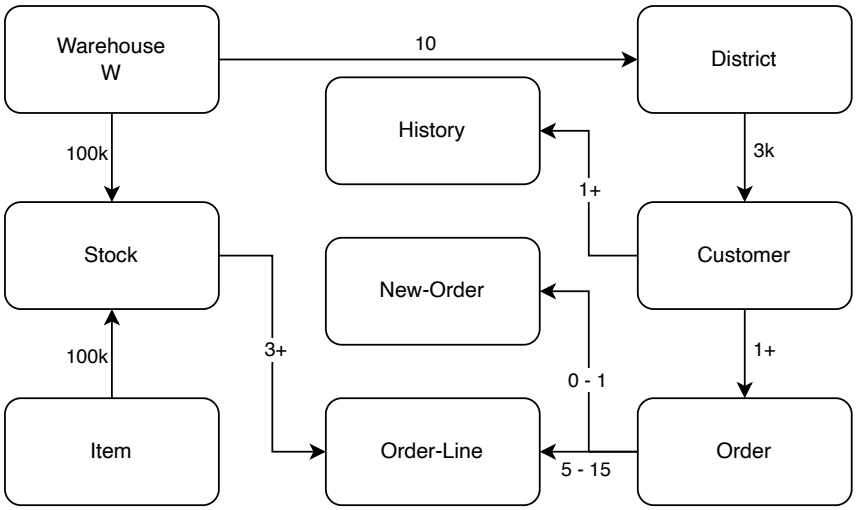


Figure 5.2: Diagram of the TPC-C database structure

Moreover, for the tables shown in Figure 5.2, **TPC-C** also defines what their data volumes must be, in order for the benchmarks to be conducted according to specification. These volumes are based on the number of warehouses W and are depicted in Table 5.1.

In order to implement the **TPC-C** specification, we must implement the same workloads that this specification defines. **TPC-C** defines 5 transactional workloads:

Table 5.1: Data volumes of TPC-C tables, according to specification

Table	Number of Rows
Warehouse	W
Stock	$W * 100000$
Item	$W * 100000$
District	$W * 10$
Customer	$W * 30000$
Order	At least $W * 30000$
History	At least $W * 30000$
New-Order	At least $W * 9000$
Order-Line	At least $W * 300000$

1. The **New-Order** Transaction — which represents the main transaction we want to measure. This transaction consists of creating a new order, reading and writing data from several tables of our database system;

2. The **Payment** Transaction — which when executed reflects a payment made by a customer to the company. This transaction affects the sales statistics for warehouses and districts;

3. The **Order-Status** Transaction — obtains the order status of a customer's last order;

4. The **Delivery** Transaction — processes a batch of 10 orders that are not marked as delivered yet;

5. The **Stock-Level** Transaction — obtains the number of products that have their stock levels below a defined threshold.

TPC-C also defines, that while during the execution of performance testing, each transaction must maintain its appropriate mix. The mix of a transaction serves to guarantee that transactions which require more work are executed more often, so that every transaction approximately executes the same number of times in the duration of the test. Since the transaction where performance is being measured is the **New-Order** transaction, we do not impose this mixing constraint upon this transaction, as we want to find what is the maximum number of new orders we can submit during our evaluation. The mixing constraints are defined, for each transaction, in Table 5.2.

Table 5.2: Mixing constraints for TPC-C transactions

Transaction	Minimum % of mix
Payment	43%
Order-Status	4%
Delivery	4%
Stock-Level	4%

For the implementation of the **TPC-C** benchmark we defined a common interface that all users can follow to test their implementations. This interface is represented by the `Tpcc` trait in **Rust**, which defines functions that must be implemented so that each transaction is executed on the destination system, and functions to populate data in the database. Having a future agnostic interface for **TPC-C** allows any database implementation in **Rust** to re-utilize this work to measure the performance its implementation.

Additionally, **TPC-C** can also be used to test whether our proposed solution complies with the ACID properties, however, we did not perform these tests, as they would require further modifications to our **TPC-C** implementation.

6

Conclusions

This chapter focuses on analyzing the work that was done, its results and whatever limitations there may exist, where future work could be done to improve them.

During the course of this document we have discussed the shortcomings of the original **LSD** specification, as initially proposed by Vale [25]. In our opinion, these shortcomings prevented Lazy State Determination from ever being useful to a real-world user, as it did not provide many of the tools that are required in a productionized environment. Building upon this work, we have presented a new specification for **LSD** that aims at overcoming these shortcomings, making **LSD** more viable in a real environment.

We started by proposing to treat every concrete value as a resolved Future. This meant that, for passing combinations of concrete values and Futures to the functions defined in the **LSD** specification, we could simply only provide Futures to these functions. By doing this, we demonstrated that we could reduce the number of required functions in the **LSD** specification, saving precious time to developers who want to implement Lazy State Determination and improving developer experience, by creating a streamlined interface.

Another issue we noticed was that, for certain types of operations such as checking if a boolean was true or not during the context of a transaction, there was a need to introduce special operations in the **LSD** specification. Although we were not able to eliminate this extra operation in the specification, we were able to make it much more flexible. By allowing not just booleans, but all futures, to be speculated, we provide users with a new feature that was not possible before.

As we analyzed the specification for the original **LSD** commit algorithm, we noticed that it could not handle certain transactional behaviors. By splitting the commit algorithm in two phases: Read and Write; we are essentially ignoring the ordering of operations in a transaction, which means that behaviors such as read-after-writes are not desirable, as they would produce an unexpected outcome for most users. In this documented we proposed a new algorithm for the commit phase of Lazy State Determination, which fixes these issues by respecting the ordering of operations in a transaction. This is done by keeping all the operations performed during a transaction in an ordered list, and iterating over these operations by the order they were executed to compute the state of the transaction,

effectively solving the issue that was encountered.

During the course of our work, we also took the advantage of **Rust** to give the **LSD** prototype a new beginning. While improving the original C++ prototype we quickly noticed that performing operations such as adding two futures together, adding a future with a concrete value, or simply introducing a new data type, were not obvious tasks, requiring excessive work for what they provided. With the transaction to **Rust** we managed to simplify the task of performing operations between futures and other futures (which can now also represent concrete values as a resolved Future) by improving upon the operator overloading that was already in use for the C++ prototype. By using **Rust** macros to generate a lot of repetitive code for different data types, we were able to significantly improve the developer experience, allowing us to add support to any data type more easily.

The prototype initially implemented by Vale [25] and later improved by Carpinteiro [3] also did not allow for testing of network layer, as these implementations only relied on **Apache Thrift**. As databases are distributed systems (at least in productionized environments), they need a networking layer that is capable to deliver high throughput and low latency, so that they do not represent a significant bottleneck in the performance of our transaction workloads. To this end, we proposed a solution to modularize the **LSD** prototype, by keeping the core implementation, which contains most of the server logic, in its own package. This package can be later used by any server implementation, either it being **Apache Thrift** or **gRPC** (or even another one), improving the confidence in our experimental results and giving more flexibility to developers. Unfortunately, during our testing, we were only able to implement the **gRPC** version, however, implementing a **Apache Thrift** version should be simple, now that we have provided this modular architecture.

6.1 Future Work

As with any piece of work, we believe that what we presented in here has a lot of potential to be improved upon in the future.

As we mentioned previously, the current implementation does not offer an alternative networking layer. This prevents us from testing what is the best networking approach, and under what conditions does it perform well. This means that, for effectively testing this layer, we need another implementation. We believe that this work lays the groundwork for adding new networking modules through its proposed modular architecture, so developers should feel comfortable when implementing a new network protocol.

Regarding the storage layer, we re-implemented two different providers that were already in use in previous prototypes: an in-memory HashMap and **RocksDB**; which were already introduced in Carpinteiro [3]. We believe there is potential to test more how each of these providers fair under various circumstances, and to also implement other storage providers that show promise. With the modular architecture we provide, any

developer could easily plug in any storage provider, given that the provider supports operations for retrieving a key, writing a key and sending writes in atomic batches.

One area where the work we presented today is lacking in is experimentation and validation. We have not been able to gather enough data to create good comparisons between the work we proposed in this document and the work done in previous instances. Even though we managed to increase the unit testing for **LSD**, we lack essential performance testing that is required to evaluate the areas of improvement for future work. In this area, we have implemented our own version of TPC-C in **Rust**, since one was not available at the time, however, this same implementation lacks theoretical and practical validation that makes sure the benchmark implementation is actually correct. If we cannot guarantee the correctness of the benchmark specification, then we are also unable to guarantee the correctness of the benchmarking results we get using those workloads. As such, we propose that future work is done in this area, to validate and evaluate the performance and correctness guarantees of the work we proposed in this document.

Bibliography

[1] Apache Foundation. *Apache Cassandra*. URL: https://cassandra.apache.org/ (visited on 2023-07-10) (cit. on p. 2).

[2] Apache Foundation. *Apache Thrift*. URL: https://thrift.apache.org/ (visited on 2023-07-10) (cit. on p. 16).

[3] D. M. V. Carpinteiro. "Improving Key-Value Database Scalability with Lazy State Determination". NOVA University Lisbon, 2022. URL: http://hdl.handle.net/10362/151154 (cit. on pp. iii, iv, 2, 16, 38, 40, 44).

[4] Cockroach Labs, Inc. *How we built easy row-level data homing in CockroachDB with REGIONAL BY ROW*. URL: https://www.cockroachlabs.com/blog/regional-by-row/ (visited on 2023-07-10) (cit. on p. 8).

[5] B. Ding, L. Kot, and J. Gehrke. "Improving Optimistic Concurrency Control through Transaction Batching and Operation Reordering". In: *Proc. VLDB Endow.* 12.2 (2018-10), pp. 169–182. ISSN: 2150-8097. DOI: 10.14778/3282495.3282502 (cit. on p. 12).

[6] Discord Inc. *Discord*. URL: https://discord.com/ (visited on 2023-07-10) (cit. on p. 2).

[7] Discord Inc. *How Discord Stores Trillion of Messages*. URL: https://discord.com/blog/how-discord-stores-trillions-of-messages (visited on 2023-07-10) (cit. on p. 2).

[8] Google, LLC. *Protocol Buffers*. URL: https://protobuf.dev/ (visited on 2025-03-30) (cit. on p. 38).

[9] S. Gössner. *JSONPath - XPath for JSON*. URL: https://goessner.net/articles/JsonPath/ (visited on 2023-07-10) (cit. on p. 15).

[10] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992 (cit. on p. 4).

[11] Z. Guo et al. "Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 658–670. ISBN: 9781450383431. DOI: 10.1145/3448016.3457294 (cit. on p. 10).

[12] T. Haerder and A. Reuter. "Principles of transaction-oriented database recovery". In: *ACM Comput. Surv.* 15.4 (1983-12), pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291 (cit. on pp. 4, 34).

[13] L. Lamport. "The Part-time Parliament.(1998)". In: *ACM Transactions on Computer Systems* (1998) (cit. on p. 7).

[14] H. Lim, M. Kaminsky, and D. G. Andersen. "Cicada: Dependably Fast Multi-Core In-Memory Transactions". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 21–35. ISBN: 9781450341974. DOI: 10.1145/3035918.3064015 (cit. on p. 11).

[15] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[16] Meta Platforms, Inc. *RocksDB | A persistent key-value store*. URL: https://rocksdb.org/ (visited on 2023-07-10) (cit. on pp. 16, 38).

[17] Mozilla Foundation. *Promise - Javascript*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise (visited on 2023-07-10) (cit. on p. 15).

[18] Oracle Corporation. *CompletableFuture (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html (visited on 2023-07-10) (cit. on p. 15).

[19] D. Qin, A. D. Brown, and A. Goel. "Caracal: Contention Management with Deterministic Concurrency Control". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 180–194. ISBN: 9781450387095. DOI: 10.1145/3477132.3483591 (cit. on p. 13).

[20] Rust Contributors. *Rust HashMap Documentation*. URL: https://doc.rust-lang.org/std/collections/struct.HashMap.html (visited on 2025-03-30) (cit. on p. 38).

[21] ScyllaDB Inc. *ScyllaDB*. URL: https://www.scylladb.com/ (visited on 2023-07-10) (cit. on p. 2).

[22] The PostgreSQL Global Development Group. *PostgreSQL*. URL: https://www.postgresql.org/ (visited on 2023-07-10) (cit. on p. 10).

[23] TPC. *TPC-C Homepage*. URL: https://www.tpc.org/tpcc/ (visited on 2023-07-10) (cit. on pp. 3, 16, 40).

[24] S. Tu et al. "Speedy Transactions in Multicore In-Memory Databases". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 18–32. ISBN: 9781450323888. DOI: 10.1145/2517349.2522713 (cit. on p. 10).

[25] T. M. d. Vale. "Executing requests concurrently in state machine replication". NOVA University Lisbon, 2019. URL: http://hdl.handle.net/10362/71218 (cit. on pp. iii, iv, 3, 14, 20, 23, 40, 43, 44).

[26] R. Van Renesse and D. Altinbuken. "Paxos Made Moderately Complex". In: *ACM Comput. Surv.* 47.3 (2015-02). ISSN: 0360-0300. DOI: 10.1145/2673577 (cit. on p. 7).

[27] Wikipedia Contributors. *Feedback vertex set*. URL: https://en.wikipedia.org/wiki/Feedback_vertex_set (visited on 2023-07-10) (cit. on p. 13).

[28] Wikipedia Contributors. *Multiversion Concurrency Control*. URL: https://en.wikipedia.org/wiki/Multiversion_concurrency_control (visited on 2023-07-10) (cit. on pp. 5, 10).

[29] Wikipedia Contributors. *SQL-92*. URL: https://en.wikipedia.org/wiki/SQL-92 (visited on 2023-07-10) (cit. on p. 4).

[30] Wikipedia Contributors. *Three-phase commit*. URL: https://en.wikipedia.org/wiki/Three-phase_commit_protocol (visited on 2023-07-10) (cit. on p. 7).

[31] Wikipedia Contributors. *Two-phase locking*. URL: https://en.wikipedia.org/wiki/Two-phase_locking (visited on 2023-07-10) (cit. on p. 9).

[32] C. Xie et al. "High-Performance ACID via Modular Concurrency Control". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: Association for Computing Machinery, 2015, pp. 279–294. ISBN: 9781450338349. DOI: 10.1145/2815400.2815430 (cit. on p. 10).

[33] X. Yu et al. "TicToc: Time Traveling Optimistic Concurrency Control". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1629–1642. ISBN: 9781450335317. DOI: 10.1145/2882903.2882935 (cit. on p. 10).