NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

DAVID MIGUEL VAZ CARPINTEIRO
Bachelor of Computer Science

IMPROVING KEY-VALUE DATABASE SCALABILITY WITH LAZY STATE DETERMINATION

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2022

DEPARTMENT OF
COMPUTER SCIENCE

IMPROVING KEY-VALUE DATABASE SCALABILITY WITH LAZY STATE DETERMINATION

DAVID MIGUEL VAZ CARPINTEIRO

Bachelor of Computer Science

Adviser: João Lourenço
Associate Professor, FCT — NOVA University Lisbon

Examination Committee

Chair: Vasco Amaral
Associate Professor, FCT — NOVA University Lisbon

Rapporteur: Nuno Antunes
Assistant Professor, University of Coimbra

Adviser: João Lourenço
Associate Professor, FCT — NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2022

Improving Key-Value Database Scalability with Lazy State Determination

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.17) [27].

Acknowledgements

I would like to express my deepest gratitude and appreciation to my adviser, Professor João Lourenço, for his invaluable patience and feedback, his assistance and support at every stage of this project.

A special thanks to the Department of Computer Science of the NOVA School of Science and Technology, and its professors, for the experience they provided, as well as all the valuable knowledge shared with their students.

My thanks are also extended to the HiPSTr project (PTDC/CCI-COM/32456/2017 & LISBOA-01-0145-FEDER-032456) for providing financial support.

Finally, I would like to express my gratitude to my family, specially to my mother, Odete Vaz, for her unconditional and loving support.

Abstract

Applications keep demanding higher and higher throughput and lower response times from Database systems. Databases leverage concurrency, by using both multiple computer systems (nodes) and the multiple cores available in each node, to execute multiple requests (transactions) concurrently.

Executing multiple transactions concurrently requires coordination, which is ensured by the database concurrency control (CC) module. However, excessive control/limitation of concurrency by the CC module negatively impacts the overall performance (latency and throughput) of the database system. The performance limitations imposed by the database CC module can be addressed by exploring new hardware, or by leveraging software-based techniques such as futures and lazy evaluation of transactions.

This is where Lazy State Determination (LSD) shines [43, 42]. LSD proposes a new transactional API that decreases the conflicts between concurrent transactions by enabling the use of futures in both SQL and Key-Value database systems. The use of futures allows LSD to better capture the application semantics and to make more informed decisions on what really constitutes a conflict. These two key insights get together to create a system that provides high throughput in high contention scenarios.

Our work builds on top of a previous LSD prototype. We identified and diagnosed its shortcomings, and devised and implemented a new prototype that addressed them. We validated our new LSD system and evaluated its behaviour and performance by comparing and contrasting with the original prototype. Our evaluation showed that the throughput of the new LSD prototype is from 3.7× to 4.9× higher, in centralized and distributed settings respectively, while also reducing the latency up to 10 times.

With this work, we provide an LSD-based Key-Value Database System that has better vertical and horizontal scalability, and can take advantage of systems with higher core count or high number of nodes, in centralized and distributed settings, respectively.

Keywords: Concurrency Control, Transaction Processing, High Contention Transactions, Lazy evaluation, Key-Value Database System

Resumo

As aplicações continuam a exigir aos sistemas de base de dados (BD) débitos cada vez maiores e tempos de resposta cada vez menores. As BD respondem explorando a concorrência, usando múltiplos sistemas computacionais (nós) e os vários *cores* disponíveis em cada um desses nós, para executar vários pedidos (transações) simultaneamente.

A execução de múltiplas transações simultaneamente requer coordenação, assegurada pelo módulo de controlo de concorrência (CC) da BD. No entanto, o controlo/limitação excessiva de concorrência pelo módulo de CC impacta negativamente o desempenho geral (latência e débito) do sistema de BD. As limitações de desempenho impostas pelo módulo CC da BD podem ser abordadas tanto explorando novo *hardware* como recorrendo a técnicas baseadas em *software*, como futuros e avaliação diferida de transações.

É aqui que o *Lazy State Determination* (LSD) brilha [43, 42]. O LSD propõe uma nova API transacional que permite o uso de futuros em sistemas de BD *SQL* e *Chave-Valor*, diminuindo os conflitos entre transações concorrentes. O uso de futuros permite também que o LSD capture melhor a semântica da aplicação e tome decisões mais informadas sobre o que realmente constitui um conflito. Estes dois aspetos combinam-se para criar um sistema transacional que fornece elevado débito em cenários de alta contenção.

O nosso trabalho foi desenvolvido sobe um protótipo anterior de LSD. Identificamos e diagnosticamos as suas deficiências e limitações, e concebemos e implementamos um novo protótipo que as endereçou. Validamos o novo sistema LSD e avaliamos o seu comportamento e desempenho comparando e contrastando com o protótipo original. A nossa avaliação mostrou que o débito do novo protótipo LSD é de 3,7× a 4,9× maior, em configurações centralizadas e distribuídas, respetivamente, além de reduzir a latência até 10 vezes.

Com este trabalho, disponibilizamos um sistema de base de dados de *Chave-Valor* baseado em LSD que possui melhor escalabilidade vertical e horizontal, fazendo melhor uso de sistemas com múltiplos *cores* ou com elevado número de nós.

Palavras-chave: Controlo de Concorrência, Processamento de Transações, Avaliação Diferida (*Lazy*) Transações em Alta Contenção, Base de Dados tipo *Key-Value*

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

1.1 Context

We can no longer rely on the evolution of single core machines to scale our applications. Single threaded performance improvements are over and now the focus in on increasing the core count in a single chip [36], as Moore's law dictates that we will, most probably, continue to see exponential growth of the number of cores per chip [33].

Adding to this, the increased requirements of current Database Management Systems (DBMS), as applications become more complex, larger, and need faster executing transactions, known as On-Line Transaction Processing (OLTP), leads to a rise in demand for high throughput database systems.

With this in mind, we can see the need to improve the scalability of OLTP in DBMS. One way to achieve this is by running the DBMS in a parallel computing environment. A big challenge of this is concurrency control, as we need to ensure that concurrently executing transactions still provide Atomicity and Isolation, requirements of the ACID properties of a transaction.

In light of the popularity of Cloud Computing and Everything-as-a-Service (XaaS), more and more companies are deploying their services on large-scale clusters composed of a high number of machines. With multiple machines, all executing requests in parallel, they might need to access and modify data, in a database, concurrently. To ensure that the data integrity remains consistent throughout the evolution of the system, concurrency control mechanisms are used.

Concurrency control is not an easy task, and existing methods of dealing with it, through Two-Phase Locking or Timestamp Ordering, pessimist and optimist approaches, fail at providing high throughput on machines with hundreds of cores [46].

We want to improve the scalability of DBMS by implementing better concurrency control algorithms. A lot of research has been done on this topic, and many solutions already exist. Solutions like LSD [43, 42], Sloth [7], Faleiro [14], Version Boxes [6] and Bumper [10] that add futures and lazy evaluation to transactions. Others add replication to lower contention, Doppel [30], ROCOCO [29], Callas [44] and Salt [45]. Others redefine the Optimistic Concurrency Control to improve performance, Silo [41], TicToc [47]. Some utilize new hardware functionality, such as FaRM [11] with RDMA. And Sinfonia [1] that proposes microtransactions.

With all these solutions in mind, we choose to continue the work of Lazy State Determination. LSD proposes a new approach to the conflicting transactions' problem. First, by redefining what is a conflict, allowing the application to expose its semantics to the database systems, allowing it to make more informed decisions on what constitutes a conflict. And second, by reducing the conflict window between transactions by delaying the observation of the concrete database state to the end of the transaction and operating over an abstract state. The work already done shows an improvement of 6.5× more throughput, at 1K committed transactions per second, with 60% less latency, at 70ms, using the TPC-C benchmark [31] in high contention scenarios, with 60 clients.

We want to focus on the scalability of concurrent transactions in database systems, in particular Key-Value Stores, by implementing LSD. We will do this by analysing, testing and improving an already proposed solution, understanding its limitations and shortcomings.

1.2 Problem

The problem of lower performance and reduced throughput of existing concurrency control mechanisms for OLTP, that limit the available concurrency, stems from the conflict generated by concurrent transactions.

In relation to pessimistic concurrency control, where locking is used, transactions conflict when the locks owned by one transaction are required by another concurrent transaction. This interference on locks is called lock contention. Contention on locks increases when we have highly frequent access to a few data elements, where multiple concurrent transactions, trying to obtain a lock for an item, must wait for a single transaction to finish. With high contention on locks, we get reduced parallelism, which in turn leads to reduced throughput.

The same is true for optimistic concurrency control, where locks are not used, and instead exists a verification stage at the end of the transaction that confirms that there were no concurrent modifications to the data items accessed by the transaction. Without locks, concurrent transactions are free to modify data concurrently. While, in low contention scenarios, this method has better performance than the pessimistic approach, in high contention scenarios it suffers a lot. Here there is a very high probability that transactions modify the same data items, and when that happens the transaction needs to abort and restart. This leads to wasted computation and low throughput, as it is very difficult for transactions to complete and the system to make progress.

The LSD prototype [43, 42] addresses these problems by proposing the use of futures and lazy evaluation, that reduces conflict windows, and by exposing the application semantics to the DBMS, allowing for more informed decisions on what constitutes a conflict.

The problem with LSD, in its current state, is that it has several shortcomings that need to be addressed. The first one being the perceived low throughput achieved when it was evaluated using a python client and TPC-C [40] implementation and the use of RocksDB [13] as the low-level database, which is an in-disk database, that compared to other solutions running in memory, stands behind in the observed throughput results.

The leading solution, Silo [41], shows much better throughput. It accomplishes this by having an in-memory database and by having a highly optimized implementation for the TPC-C benchmark and running on the same machine, foregoing network latency.

On the server side, the biggest issue is the in-disk database used, which causes higher latency when retrieving stored data.

Another problem with the perceived low thought put is the fact that other solutions ran their tests in a singular process, where the client and server communicate thought function calls. This completely removes the communication latency between clients and server that a real word application would have.

1.3 Objectives and Approach

The main objective is to reduce contention to be able to increase parallelism between transactions. The first thing to consider is to reduce the conflict window of a transaction, decreasing the probability of two transactions requiring the same lock at the same time and, thus, reducing contention.

We achieved reduced contention by implementing LSD [43, 42]. The problem was that, although LSD exhibited considerable speedup improvements, the original baseline for throughput evaluation was very low and was still very low with LSD. Our focus is on improving the previous LSD prototype, which was accomplished with four major tasks.

First, contextualize the results of the existing LSD prototype with RocksDB, by re-running the tests and calculate the theoretical maximum throughput of the system used to run those tests. This required the updating of all the external libraries used, as the project is a few years old. This also required some minor API changes.

Second, on the server side, we improved performance by reducing the database access latency, by replacing RocksDB with a lightweight in-memory database based on a hashmap.

Third, looking at the client side, it was written in python, and used an existing TPC-C benchmark also written in python. We studied the impact of the client in the overall throughput by replacing the existing client in python with another client developed in C++, sharing the same server. In complement to the C++ client, we also used a TPC-C benchmark written in C++.

Finally, the new prototype was tested in an improved hardware setting, in a cluster with multiple nodes, each node with different logical partitions of the database. The tests evaluated the impact of the number of clients and the worload distribution across the multiple nodes in the cluster.

1.4 Contributions

Our contributions consist of the profiling and evaluation of the existing solution to discover possible areas of improvement. This involved updating the current prototype with more recent versions of libraries and performing the required API changes, as well as developing new benchmarks to test its limits. These benchmarks test the differences between running the client and server on the same node vs different nodes. The difference between running RocksDB in disk vs memory. The improvement of having a C++ client vs python. This also required the implementation of new operation modes for the server. The NO-OP mode and NO-TX mode.

After the evaluation of the existing prototype, we improved it, optimizing the code when possible. We implemented a way to execute the current RocksDB in memory, as well as finalizing the implementation of a hashmap based database.

Then we designed and implemented a new benchmark client and TPC-C driver, both written in C++, and to test the performance benefits of partitioning the server, we implemented database sharding, with a simple hash of the key, and a TPC-C aware sharding that partitions the keys based on the table they belong to.

Finally, we developed a new system to deploy, run and obtain revenant data for the tests and TPC-C benchmark in the new computing cluster. We followed by an analysis and evaluation of this new solution, to then compare with the previous python implementation. discussing the improvements observed and possible future work.

1.5 Document Organization

In this chapter, we introduced the motivation behind this work proposal, the previous work done its problems and the solutions we developed.

In Chapter 2, we go through the required background knowledge necessary to comprehend the problem at hand, as well as to implement and evaluate the solution. This includes the work done in Concurrency Control, that can be achieved with Locking Mechanisms, Transactions and Asynchronous Programming.

In Chapter 3, we present our approach to our work, the architecture behind our solution and the reasoning behind our decisions.

After, in Chapter 4 we go over our methodology for validating our solution, its correctness, and how we evaluated it, the environment setup and types of tests used to obtain our results as well as discussing these results.

Finally, in Chapter 5 we go over our discoveries and their importance and how they can be used for further work on LSD.

2

Background and Related Work

2.1 Background

For the correct function of a computer system, it is necessary that every operation is executed over a consistent system state, that, when terminated, leaves the system in a new and consistent state. In a sequential system, operations are executed one after the other, so there is no chance of interference between operations, and this rule is trivially true.

The same is not true for a parallel system [25], where the execution of operations intersect. This is problematic as in the middle of the execution of one complex operation, when the system state can be inconsistent, another operation can start executing in an inconsistent state and drive the system irreversibly inconsistent.

Concurrency control [4] is needed to maintain consistency on the presence of concurrent operations. It ensures that, even in the presence of intersecting operations, all operations begin seeing a consistent system state and terminate leaving the system in a consistent state.

In a parallel system, the risk of corruption is high, and therefore Concurrency Control mechanisms are generally complex, as they must deal with highly convoluted problems. Having complex mechanisms can mean an additional overhead in the system, leading to a decrease in performance.

For these reasons, Concurrency Control is a very complicated subject that as several implications in different areas of a computer system. Here we will discuss three main solutions to Concurrency Control. We will discuss **Locks** as a mean of synchronization between multiple threads in a parallel system, **Transactions** as a mean of dealing with concurrent access to a database system and **Asynchronous Programming** as a means of providing concurrency to a system without the use of locking strategies.

2.1.1 Consistency Models

A computer system can support multiple Consistency Models. By following a consistency model, we can define what kind of rules the system will follow. With a data consistency model, it is possible to make assumptions on the results of read and write operations.

The stronger the consistency model, the more restrains it requires from the system, leading to decreased performance. As that is not a desirable quality, weaker consistency models exist that provide better performance but with fewer guarantees.

Starting with the strongest of consistency models, we have linearisability [22]. Linearisability is a strong correction condition that ensures that every read on a shared object always returns the most recent write. With this model, we can attribute a linearization point to each operation, giving the appearance that the effects of an operation are instantaneous. For a system to provide linearisability, it must satisfy that [32]:

- Every operation must have a sequential equivalence. Meaning that concurrent access to a shared object must yield the same result if the operations were done sequentially, one after the other.

- An operation must complete before another can begin. No new operation can begin in the middle of an ongoing operation.

By following these rules, we ensure that there is never an intersection of operations over the same shared object, which in turn means that there is never an inconsistent state in the system. The big disadvantage of this model it is the performance, as it turns a concurrent system into a sequential system, removing all the benefits of concurrent operations.

To obtain better performance, the consistency model must be relaxed. This takes us to Weak Consistency Models. Here the two main models are Eventual Consistency and Causal Consistency.

Eventual Consistency states that, when the system stops receiving updates, eventually the system converges, and all reads return the same, most recent, value. This means that, while the system is receiving updates, a read operation can return any previously written value.

With Causal Consistency, a read operation always observes a state that follows the causality relation between writes, this means that only causal operations appear in order, while other operations can be observed with different orders. This is a stronger model compared to Eventual Consistency, that does not impose any order on the operations. Causal Consistency can be achieved by providing the following four session guarantees [37]:

- Read Your Writes. Within a session, a read operation must always return the most recent written value by that client.

- Monotonic Reads. Within a session, subsequent reads must always return the same value or a more recent value. Never can a read operation return a value older than one already observed.

- Monotonic Writes. Multiple writes within a session, i.e., by the same client, must be observed respecting that order by other clients.

- Writes Follows Reads. If a read operation is followed by a write operation on the same item, the write must be ordered after the previously read value.

Following these guarantees means we get a stronger weaker consistency model, the only downside compared to Eventual Consistency being that there is no guarantee of convergence.

2.1.2 Locks

Locking [32] is a mechanism used to restrict the access to a resource that is being accessed by multiple threads, making it possible to achieve mutual exclusion. Mutual exclusion being the preventative measure put in place to prevent race conditions and create a critical section, where only one thread is allowed at a time. Race conditions occur when the value of a shared resource is dependable on non-deterministic concurrent operations, non-deterministic because the order in which different threads access or modify the value cannot be controlled and is dictated by a scheduling algorithm.

To prevent race conditions, a critical section is used, where shared resources can be accessed and modified safely. Code inside a critical section behaves as an Atomic Operation. The critical section synchronizes the different processes and must follow three conditions:

- Mutual Exclusion, only one process is allowed in the critical section at a time.
- Progress, a process can always enter the critical section if it is free.
- Bounded Waiting, processes have a limited waiting time to enter the critical section.

2.1.2.1 Types and Uses of Locks

A Mutual Exclusion Object (mutex) is a locking mechanism implemented as an object. Only one thread is allowed to own a lock on this object at a time, and only the thread that acquired the lock can release it. Any other thread that wants to acquire the lock must wait in a queue. This gives the necessary tools to create a critical section.

Next, we have Reentrant Locks. Reentrant, meaning that a process can claim the lock on a resource multiple times, without releasing it. This is, when a process obtains a lock for a shared resource, it can lock it multiple times after, while the system keeps track of the number of requested locks. The lock is only released when there is the same amount of unlocks as locks. This type of lock is useful in situations where it is difficult/impossible to know if the process already obtained the lock on an object (keep track of owned locks), allowing said process to reclaim the lock without blocking itself. If we were using a simple lock/mutex, a process would block when requesting the same lock twice, without unlocking first. This type of lock also offers a fairness condition, giving the lock to the thread that waited the longest.

Next, we have Read-Write Locks. While the two previously discussed locks block all types of access to the locked object, a read-write lock allows for more granular control.

This lock allows for concurrent access, from multiple threads, to read only operations over the lock object, while imposing exclusive access to write operations. This means that multiple threads can read data in parallel, leading to an increased level of concurrency in the system. For write operations, an exclusive lock is still required, which also blocks read operations. To obtain these two types of locks, certain conditions need to be meet:

- To obtain a Read Lock no other thread must own a write lock.
- To obtain a Write Lock no other thread must own a read or write lock.

From these conditions, we can conclude that, if we don't prioritize write locks, starvation can occur, as the write lock request would block until all readers unlock the read lock. With this in mind, a read lock can only be given when there are no write locks and no write lock requests are pending.

Finally, we have Multiple Granularity Locking, which is used in database management systems. Granularity, meaning the size of data that will be locked. Here the database can be divided into Multiple hierarchy levels, has in a tree structure. In this structure, the root node is the database, and its children are the nodes called areas, areas then have files which in turn have records. When a lock is explicitly applied on a node, it's implicitly applied on all its descendants, without the necessity to lock all its descendants individually. Here, locks are based on non-strict two-phase locking.

Summarizing, we have two main levels of granularity.

Coarse, where we lock large amounts of data, leading to a reduce number of locks. By using fewer locks, we reduce the overhead of supporting multiple locks, and in a scenario of low contention, better performance. But if we have multiple threads trying to access the same shared objects, we will get worse performance because of the lock contention derived from the large window of conflict.

To solve this, we can use **Fine** granularity. By locking a smaller amount of data and using more locks, we increase the lock overhead, but reduce the lock contention by reducing the conflict window.

2.1.2.2 Disadvantages of Locks

Lock overhead refers to the increase in resources needed to use locks. This includes the memory allocated, the CPU time needed to create and destroy locks, and the time spent waiting to acquire locks. The more locks used, the worse is the performance.

Lock contention occurs when a thread tries to acquire a lock on an object that is already locked by another thread, resulting in the thread having to wait until the lock is released. More contention leads to reduced concurrency in the system. Contention can be minimized by using more fine-grained locks.

Deadlock occurs when there are at least two processes requesting a lock that the other holds. In this scenario, no progress can be made, as there is no solution that can make the threads continue execution, making them wait forever. For a Deadlock to occurs three conditions must be meet, known as the Coffman conditions [8]:

- Mutual Exclusion, at least one of the resources is held in an exclusive mode.
- Hold and Wait, a process must be holding a resource and requesting additional resources.
- No Preemption, a resource can only be released by the process that holds it.
- Circular Wait, a process 1 is waiting for a resource held by a process 2, which is waiting for a resource held by a process X, which in turn, is waiting for the resource held by process 1.

In the presence of a Deadlock, there are three main approaches to dealing with it:

- Ignoring, where we assume there is the possibility of a deadlock if null.
- Detection, an algorithm is used to detect the presence of a deadlock so that the situation can be corrected by: (1) Process Termination, terminating one or more processes and checking if the system is no longer in a Deadlock state. This method has the possibility of loss of data. (2) Resource Preemption, allocate the requested resources to another thread until the deadlock is resolved.
- Prevention, to prevent a deadlock one must prevent one of the Coffman conditions from being true:
  - Removing Mutual Exclusion, by using non-blocking algorithms
  - Removing Hold and Wait, by requiring all threads to request all locks at once. Problematic because it can lead to resource starvation.
  - Removing No Preemption, this requires that a roll-back option must be implemented to avoid data corruption, this is used in lock and wait free algorithms and OCC.
  - Remove Circular Wait, for this it is necessary to determine the ordering of resources and conclude that no hierarchy exists.

It could happen that the policy used to decide what process is allowed access to a resource, i.e., having its request fulfilled, could lead to a process never gaining access to the resource, or the resource being unavailable for long periods of time. In this situation, we say that the process is suffering from starvation. While the process is not in a deadlock, it also cannot progress, as it is waiting for a resource that it will receive. While a mutual exclusion algorithm that arbitrarily chooses a process to enter a critical region can be called deadlock-free, a much stronger guarantee is needed for and the algorithm to be starvation-free. Starvation can be caused by errors in the scheduling algorithm, mutual exclusion algorithm or by resource leaks.

2.1.2.3 Final Thoughts

Locks gives us a safe way to synchronize concurrent access to data, but the problem with using locks as a concurrency control mechanism is that it's a pessimistic solution. It assumes that conflict is always a possibility, and therefore takes the most extreme stance to ensure that no problems arise in that event. Using locks means we decrement concurrency a lot, creating a system that, despite secure, is under performing.

2.1.3 Transactions

2.1.3.1 Concept

A transaction [18] is a container for a set of actions that behaves as a single unit of work, that is used to maintain a set of properties in a system, in the presence of concurrent operations. If a transaction does not corrupt the system, then all actions in it are applied, else no action is executed.

2.1.3.2 Uses

There are two main types of transaction models. Transactional memory and database transactions.

Transactional memory [21] is a mechanism to control the access of a shared memory resource and synchronize processes that execute in parallel. In case of locks, we would need to define a critical section of code that cannot be parallelized, i.e., not be executed simultaneously by several threads. Only the thread that has a lock on the critical section can enter it, while other threads must wait for it to complete and release the lock. In turn, transactional memory allows these critical sections to be defined as transactions, which them get managed by a transactional memory system that ensures that the result of multiple threads accessing the critical section in parallel will be the equivalent to a serial execution, where each thread executed isolated from the others. This transactional memory system can be implemented in software, Software Transactional Memory (STM), or in hardware, Hardware Transactional Memory (HTM). This system monitors the accesses of parallel transactions and detects the presence of a conflict, in which case it aborts the responsible transaction. This transaction is therefore forced to stop and restart.

The concept of transactional memory is analogous to a database transaction, which we will now present.

In databases, transactions [12] represent a system state change, that begins in a consistent state and ends in a consistent state. Each transaction appears to execute isolated from other transactions. The modifications of a transaction are all applied, or none are, and if successful, the changes will persist in the system. These properties are known are the ACID [19] properties, which we will discuss later.

2.1.3.3 Properties

The main four properties of a transaction are known as the *ACID* properties.

Atomicity A transaction behaves as an atomic unit, meaning that or all operations are executed, or none are. No transaction is ever partially completed.

Consistency A transaction must ensure that it begins in a consistent state and terminates, leaving the system in a consistent state. A consistent state is one that does not break any database constraint.

Isolation In the case of parallel transaction, the system must ensure that all transaction execute isolated, with no interference from other transactions executing simultaneously.

Durability If a transaction succeeds, its effects on the database should be durable, and persist even in the case of a system failure.

By following these properties, we can build a reliable and safe transactional system.

The ANSI/ISO standard introduces *read phenomena*, which happen at when we use different isolation levels and a transaction can read data that another transaction might have changed. First, we have **dirt reads**, which happen we use the **read uncommitted** level and transaction can read uncommitted data that was modified by another concurrent transaction. Another similar phenomenon is **non-repeatable reads**. This happens at the **read committed** and **read uncommitted** levels when reading an item multiple times during a transaction results in different values, due to another concurrent transaction modifying the value and successfully committing the changes in between reads of the other transaction. The final phenomena are **phantom reads** which are a special case of **non-repeatable reads**. Here, new values are added or removed during the transaction, leading to repeated range queries returning different amounts of items. This can happen in the **Repeatable Reads** isolation level. While the phenomena introduced are part of a standard, more have been defined, including Lost Updates, Read Skew and Write Skew.

With the *read phenomena* in mind, the ANSI/ISO standard also defines four isolation levels [3] that prevent them and provide a compromise between performance and safety.

Serializable It is the strongest isolation level. Serializability can be achieved by using a locking or non-locking concurrency control protocol, and ensures that there is no collision between data accesses and modifications in the presence of concurrent transactions. It avoids the three phenomena previously described.

Repeatable Reads Here only **phantom reads** can occur because in the case of a lock-based concurrency control protocol, only the read and write sets are locks, with no range locks.

Read Committed At this isolation, write locks are kept until the end of the transaction, but the same is not true for read locks, because of this **non-repeatable reads** can occur.

Read Uncommitted It is the lowest form of isolation, therefore all the *read phenomena* are possible.

2.1.3.4 Concurrency Control of Transactions

The serial execution of transactions is slow, because of the added overhead, therefore, to benefit from their use, they are executed concurrently. To maintain all the ACID properties of transactions in a parallel system, a concurrency control protocol is used to ensure isolation.

We will first introduce the concept of schedules [12]. A schedule is any sequence of operations from a set of transactions that preserves the order in which operations appear in each transaction. A serial schedule is when a set of transactions are executed one after the other, with no interleaving of operations. It's the simplest form of isolation, where no parallelism is allowed. When we want to have concurrent executing transactions, we use serializability. A serializable schedule is one that whose output is equal to a serial schedule, meaning no conflicts arose from the interleaving of operations from multiple concurrent transaction and no insistent state is observed by them.

Concurrency control mechanisms are used to preserve the integrity of the database in the event of concurrent access of data. Two transactions conflict when their concurrent execution causes an inconsistent state. To prevent this, we can use two main concurrency control protocols, a pessimistic one and an optimistic one [28], which we will now present.

Optimistic concurrency control uses conflict detection to preserve isolation, while pessimistic concurrency control uses locking the means of obtaining serialization.

For the pessimistic concurrency control protocol, we have two phase locking (2PL) [39]. Here, a transaction needs to obtain a lock before it can execute a read or write operation on an object. Two transactions cannot hold a conflicting lock, i.e., lock on the same object, at the same time. Also, as soon as a transaction releases a lock, it can no longer request a new lock. These two insights help define the two phases of the protocol. In the first phase, the growing phase, the transaction acquires all the locks it will require for the rest of the execution, not releasing any. The second phase, the shrinking phase, starts when the transaction first releases a lock, not being allowed to obtain any more locks. Once a transaction commits or aborts, all locks are automatically released. In the case of strict two-phase locking (S2PL) the write locks can only be released at commit or abort. For conservative two-phase locking (C2PL), a transaction needs to acquire all the required locks before its initialization.

2PL is considered pessimistic because it expects that conflict will occur, leading to all transactions having to acquire a lock for any object they access or modify. When a transaction cannot acquire a lock, they need to wait until the lock is released and the object becomes available. The waiting time needs to be controlled to avoid the occurrence of deadlocks. This can be dealt with by using deadlock detection, that monitors an await-for graph for the creation of cycles, and in the event of a deadlock, aborts, and restarts the required transaction to break the cycle. Another method is non-waiting deadlock prevention, in which a transaction is not allowed to wait for the release of a lock, by being aborted.

For the optimistic protocol, we have timestamp ordering (T/O). Here, the serialization of transactions is done before they execute. When a transaction is created, it gets assigned a unique timestamp, which is then used to resolve conflicts. Next, we will present three main T/O protocols.

In the basic T/O protocol [4], for every object accessed by a transaction, its timestamp, given by the timestamp of the last transaction that modify it, is compared with the timestamp of the transaction. The transaction aborts if its timestamp is lower than the timestamp of the object needed for a read or write operation. This happens when the object was modified after the transaction tries to access it. When aborted, the transaction is restarted and given a new timestamp.

In multi-version concurrency control (MVCC) [5], the database maintains a list of versions for every object. Every time an object is modified, a new version is added to the list. When a transaction tries to read an object, the database selects each version it should return, ensuring a serializable order of operations. In contrast with the basic T/O protocol, MVCC allows a read operation on an object that was modified after the transaction started, returning a previous version.

Finally, the Optimistic Concurrency Control (OCC) [26], creates a read and write set for each transaction, keeping all the write operations local until the commit phase. On commit, the database checks if the read set does not intersect with a write set of other concurrent executing transactions, and then applies the writes that were stored locally.

Comparing the two protocols, 2PL performs better in high contention scenarios, where there is a high number of concurrent conflicting transactions. In this scenario with T/O, transactions will constantly abort, with little progress being made. On the other hand, T/O performs better in low contention scenarios, where the risk of conflict is low, allowing it to reap better performance in comparison to 2PL, that suffers from the overhead of using locks.

2.1.3.5 Partitioning

The use of partitioning allows for more parallelism when executing transactions, providing better performance and availability, while also reducing contention. Data divided in partitions can be managed and accessed separately, while also allowing the division to be done by some criteria that allows the data to be store by the type of usage. Using cheaper storage for infrequently accessed data.

Partitioning can improve performance because it allows the database to be divided into logical partitions, proving better load balancing and access over a smaller volume of data. If a query runs over more than one partition, in can be executed in parallel. It also allows for better scalability because it removes the physical hardware limit of a single database, by having multiple servers running with different partitions. Scale out. Availability, because it allows for a partition to exist in multiple nodes, avoid a single point of failure

Partitioning can be done horizontally, vertically and functionally.

- When done horizontally, also known as sharding, it means that rows from a single table can be separated into multiple tables following a condition. The original table becomes a view that is a union of the different tables.
- When done vertically, the columns in a table are separated into multiple tables that can be store in different physical locations, prioritizing frequently accessed data.
- With functional partitioning, data is aggregated by its context in the system, putting the relevant data for one operation in the same partition.

Partition also follows some criteria to split the database. These criteria are applied to the partition key of the table.

- Range Partition, here the partition key is assigned to a partition based on a certain range.
- List Partition, here the partition key is assigned to a partition if it is contained in a list.
- Composite Partition, here the partition key, is assigned to a partition based on a combination of the two types above.
- Round-robin Partition, here the partition key is assigned to a partition based on a uniformed distribution, allowing access to sequential data to be executed in parallel.
- Hash Partition, here the partition key is assigned to a partition based on the hash of the key, giving it a direct match to a partition. Allows for direct match queries.

A partition scheme needs to be designed before it's used and cannot be modified while the system is running. To take advantage of partitioning, we need to increase granularity, leading to the organization of data into small sets that increase performance by allowing for parallel operations.

2.1.3.6 Distributed Transactions

Simply put, a distributed transaction [4] is a transaction that spans across multiple nodes of a distribute database. This is the case for a partitioned database, where each partition can be located in separated physical nodes.

To ensure the correct execution of distributed transactions, including the ACID properties, the two-phase commit algorithm (2PC) is used. In the 2PC algorithm, the processes in nodes are coordinated into deciding whether to commit or abort the proposed transaction. This is done in two phases. First is the commit-request phase, where all nodes receive a request to vote on whether to commit or abort the current transaction, depending on the success of the transaction in a particular node. Second is the commit phase, where all votes are gathered, and if all decide to commit, then all nodes are notified of the decision. Every node them commits or aborts the current transaction depending on the result. This way all nodes either commits or aborts.

2.1.3.7 Final Thoughts

With transactions, we can group multiple operations into a single unit of work, that represent a stat change in a database, ensuring the four ACID properties. To allow for the safe execution of concurrent operations, concurrency control protocols are used to provide safe ordering of operations. Specific protocols can be used to provide an optimized execution for specific scenarios, in order to improve performance. Transactions can also span multiple nodes in a distribute setting, where data can be partitioned to further optimize the execution of transactions. In this case, an algorithm like 2PC is used to synchronize the different nodes.

2.1.4 Asynchronous Programming

In a synchronous system, only one operation is executed at a time, blocking the execution of anything else until it completes, with no parallelism between operations. With asynchronous programming, we can have multiple operations executing at the same time. An operation only needs to be started and can then run in the background while other operations to continue running. This type of operation is called non-blocking.

2.1.4.1 Coroutines

We will first discuss subroutines [9]. Subroutines, also called functions or procedures, are used to organize a set of operations into a block, allowing for the organization and reusability of code. This block can be invoked multiple times from anywhere in a program, receiving inputs and producing results. A subroutine can also call other subroutines, but by doing this means it loses control until the other subroutines terminate.

A coroutine [24], co for cooperative, does not have this restriction. When a coroutine calls another, it does not lose control, instead it can continue to execute and pass values to the other coroutines. This allows for concurrent executing of operations. Compared with threads, coroutines are simpler to implement, not requiring synchronization, but do not offer parallelism, as only one coroutine executes at a time, meaning only concurrency is possible.

2.1.4.2 Future

To be able to synchronize an asynchronous program, a common solution is the use of futures [17]. Simply put, they represent an object whose value is still unknown, as the computation is still being made, but whose value will eventually be resolved, behaving like a placeholder for the actual value. A future object provides a read only view of the value. This value can be obtained in a blocking manner, by explicitly waiting for it to be resolved. Or in a non-blocking manner, by way of a notification, in which case it will be accessed by the main program.

Lazy or delayed evaluation [20], contrary to normal eager evaluation, uses futures to improve performance by delaying the evaluation of an expression until it is needed. By doing this, we can discard expressions that are never used and save time by eliminating temporary computations. It is also used to avoid repeated evaluations of the same expression, by sharing the results when needed. When an expression is assigned to a variable, the expression is not evaluated, instead a future is created and the evaluation is delayed until the moment the actual/concrete value is needed. Lazy evaluation also leads to reduced space complexity by only having to compute the necessary values.

2.1.4.3 Actor Model

Actors can be seen as isolated objects that can receive, process and send messages. They operate in an asynchronous system, have their own private internal state, where they can store received messages for later processing to operate sequentially. Actors can also create new actors and send messages to other actors. They can be running all in the same machine or be physically separated.

They enable the use of asynchronous communication between components and can be used to build concurrent systems, allowing for ease of scalability. This is due to not having to deal with concurrency problems at the language level, in a single machine, and in a distributed setting. By using actors, these problems are dealt in the same way, as the physical location of actors is not a requirement to the way concurrency control is applied. All of this leads to a more consistent system, where every component is represented by an actor, making communication and execution simpler.

2.1.4.4 Final Thoughts

Asynchronous programming provides us with tools to create concurrent programs and optimize their execution in multithreaded environments. But, by itself, does not provide a way of dealing with the problem of safety when executing concurrent operations, this is left to the programmer. The concepts presented can still be used, in conjunction with other mechanisms, to developed higher performance algorithms to deal with such problems.

2.2 Related Work

In this chapter, in Section 2.3, we present some of the more popular work done to improve the throughput of high contention concurrent transactions that provide serializability. We will combine similar solutions into sections where we discuss how they make use of the particular technique and the improvements they were able to obtain, as well as why LSD can provide a better solution. In Section 2.4 we present LSD, how it works and its current status.

2.3 Previous Work

2.3.1 Lazy Evaluation

Sloth [7] uses lazy evaluation to delay computation in the database. It uses dynamic analysis of the application to obtain information about its state, using that information to batch the queries issued by the application. It batches the queries, in the application, until a value derived from those queries is needed. This allows Sloth to have lower latency of database operations by reducing the number of round trips between the application and the database. While it shows good results, reducing latency by 1.5 times and a medium speed-up between 1.3× and 2.2×, their results for the **TPC-C** benchmark show a medium speed down of 8.52%, leading to slower execution times.

LSD, while still using lazy evaluation, it goes further, by sharing application semantics with the database allowing the concurrency control protocol to extract more parallelism.

Faleiro [14] uses a lazy transaction execution engine, deferring the execution of the transaction's logic to a later time in the future. It does this while not modifying the semantics of the transaction and maintaining the ACID properties. It achieves better temporal locality and reduces contention footprint of concurrent transactions. But it also shows increased latency for read intensive transactions. In terms of implementation, it requires transactions to be stored procedures and their write sets to be known a priori. It uses a contention footprint to decide which query execution leads to the least contention. In relation to the **TPC-C** benchmark, with a high contention, running with only one warehouse, they were able to achieve an 4× improvement in throughput. But they reach a bottleneck, as the size of the database scales and more warehouses were added (and contention decreases), leading to reduced throughput cause by high overhead.

LSD is not limited to store procedures, supporting a client-server execution model, and does not require the write set to be known beforehand. It also offers the added benefit of providing conditional validation and locking, allowing for more parallelism.

Versioned Boxes [6] propose the use of per-transaction boxes, in which private values of a transaction are kept, and then merged with shared values in the commit phase, and restartable transactions, that, in case of conflict, only have to re-execute the part of the transaction that caused the conflict. It indicates that it can improve concurrency in long, high on reads, transactions, by having no conflict between read only transactions and delaying computations over high contention objects until the commit phase, after validation. But the delaying is only possible if the value of the object is not used elsewhere during the transaction, e.g., a read-write transaction.

Bumper [10] main objective is to shelter transactions from conflicts, by minimize aborts in high contention transactions. They propose the use of time-warping, that allows a transaction that observes a stale value to be serialized before the concurrent transaction that cause the stale value. And using delayed actions that allow the serialization of conflicting transactions. Once again, this delaying is only possible if the object updated is not required for the remainder of the transaction. In their evaluation, they achieved a 3× higher throughput in a high conflict scenario, with only an 2% overhead in a scenario of no contention.

LSD does not require such restrictions on the delaying of computations, as it utilizes futures that can be used throughout the transaction and then resolved at the commit phase. Occasionally, it can hold the evaluation of the future past its use in the application logic, adding the operation where the future is used into a list of pending operations, i.e., the write-set contains not only values but operations as well.

2.3.2 Redefined OCC

Silo [41] is an optimistic concurrency control algorithm that provides better performance over traditional OCC protocols. It uses a global timestamp that is used to dictate the serial order of transactions and detect conflicts. It's implemented using a shared-memory store, taking advantage of system memory and caches. In the **TPC-C** benchmark, they show near linear scalability with very high throughput and minimal overhead. Their results are not very interesting in the context of high contention transactions, as they utilized a single thread client to make requests to the database, not generating concurrent transactions, that access the same warehouse, in the context of the TPC-C benchmark.

TicToc [47] introduces a new optimistic concurrency control algorithm that reduces the bottlenecks derived from increasing scalability and concurrency. It proposes a data-driven timestamp management protocol that assigns timestamps to data items, instead of transactions, and uses those timestamps to lazily calculate a valid timestamp for each transaction, producing a serial order of transactions. In relation to the **TPC-C** benchmark, it achieves 3× the throughput of 2PL with non-waiting deadlock prevention, in a low contention scenario of 4 warehouses, 1.8× better throughput than **Silo**. With the increase in the number of warehouses, and the decrease in contention, **TicToc** can still provide better throughput until the use of 80 warehouses, where the results become the same.

LSD is compatible with these algorithms and can be used in conjunction with them to further improve performance.

2.3.3 Replication

Salt [45] proposes a type of sub-transaction that needs to be manually created from a top level ACID transaction. These sub-transactions can expose their modifications to each other allowing for more coordination, resulting in more concurrency between them. The downside is, they rely on a developer's knowledge of concurrent programming in order to do it safely. In their results in the **TPC-C** benchmark, they were able to obtain 6.6× higher throughput compared to normal ACID transactions.

LSD allows the automatic extraction of concurrency from existing transactions with minimal intervention from the programmer.

Callas [44] extends **Salt**'s work by providing an automatic method of partitioning the original transactions, without any manual modification to the existing transactions. But it requires that all transactions are known in advance to perform static analysis to decompose the transactions into partition groups. In the **TPC-C** benchmark, they were able to be reach 95% the throughput of **Salt**.

LSD, in comparison, can extract concurrency from the transaction at runtime, adapting to the introduction of new transactions into the system.

Doppel [30] increases performance by replicating contended data, allowing parallel commutative updates, that are then merged into the global store. But, by doing this, it blocks transactions that need to read or write non-commutative updates from executing. According to their tests, they achieve better performance when writes outnumber reads, and are being executed over a small number of objects, obtaining 3× more throughput at the cost of increased latency.

LSD does not limit the types of transactions that can execute in parallel.

ROCOCO [29] introduces distributed transactions that allow for more concurrency than 2PL and OCC. It forces prior knowledge on how partitions are structured on the servers in order for transactions to be organized into portions that access the same partition. It also requires static analysis to be performed on the transactions, and, therefore, for transactions to be known beforehand.

LSD does not require transactions to be known beforehand, and it is not bound to any specific partition policy.

2.3.4 Specialized Hardware

FaRM [11] leverages two new hardware functionalities, RDMA, remote direct memory access, and non-volatile DRAM, eliminating storage and network bottlenecks. It also proposes an improved optimistic concurrency control protocol, that reduces the number of messages in a distributed setting.

LSD could take advantage of the hardware improvements proposed by FaRM, but LSD goes further by reducing the number of conflicts in OCC and 2PL leading to more concurrency in high contention scenarios, something that FaRM deals by deploying more machines.

2.3.5 Redefined Transactions

Sinfonia [1] proposes a reduce form of transactions, microtransactions, that allows the batching of operations, reducing the number of network trips. Because of their reduced scope, these microtransactions can be executed within the commit phase. This comes with a loss of expressiveness, like the lack of read-modify-write operations.

LSD does not have such limitations, and when futures are not resolved before the commit phase, can also execute the transaction just in the commit phase in the 2PC protocol.

2.4 LSD

In this section, we will discuss why we believe that LSD [43, 42] has a better method of concurrency control than other proposed solutions. We will explain the general idea behind LSD, and its inner workings. Ending with a summary of the work already done, and what can still be done to improve the current implementation.

2.4.1 General Idea

LSD proposes a new interface to describe a transaction in a way that it can expose semantic information to the database. To accomplish this, a new interface was proposed and new optimistic and pessimistic concurrency control mechanism algorithms were design and implemented to take advantage of the new semantic information, allowing the use of conditional locking and conditional validation.

The exposure of semantic information to the database allows for a more informed decision on what constitutes a conflict between two concurrent transactions. To accomplish this, LSD adds a new *tx-read* operation that, instead of returning a concrete value, returns a future. These future values allow the transaction to execute over and abstract state of the database, that is resolved as late as possible, if possible, during the *commit* phase, reducing the conflict window between concurrent transactions and decreasing the probability of conflict.

With the addition of futures, we still need to operate over them. For this, LSD adds another operation, *tx-is-true*, allowing transactions to use conditional branching using futures. It also allows the transaction to define updates to the database utilizing functions based on those future values. These functions are lazy-evaluated, and therefore, they

Algorithm 1: Traditional interface

```
tx-begin;
$v \leftarrow$ tx-read($stock$);
if $v \geq qty$ then
    $v \leftarrow v - qty$;
    tx-write($stock, v$);
    tx-commit;
else
    tx-abort;
end
```

Algorithm 2: LSD interface

```
tx-begin;
$\square \leftarrow$ tx-read($stock$);
if tx-is-true({$\square \geq qty$}) then
    $\triangle \leftarrow \{\square - qty\}$;
    tx-write($stock, \triangle$);
    tx-commit;
else
    tx-abort;
end
```

Figure 2.1: Traditional Interface versus LSD Interface

can be used with futures, and provide a method for conditional updates. The operation *tx-write* also needed to be modified, to allow writing future values, including writing values based on futures keys.

To take advantage of LSD, the OCC and 2PL algorithms were modified. In the case of OCC, by using conditional validation of objects. And in the case of 2PL, by utilizing conditional locking. These modifications will be discussed further in Section 2.4.3.

With the addition of futures and lazy-evaluation, LSD allows increased concurrency while still providing strict serializability.

In Figure 2.1, Algorithm 1 we can see an example of a simple transaction, where we try to buy a certain quantity (qty) of stock. It checks if there is enough stock to buy, before decrementing its value and committing the change, or aborting if there is not enough stock. The database sees this transaction as simple read and write operations, with no understanding of the relation between the value read and the value written. This is problematic because, if another concurrent transaction modifies the value of stock, before this transaction finishes, this transaction fails, even if the value of stock stills makes the condition (stock >= qty) true. The reason behind this is, in the case of OCC, that the database resisters the value of the stock when it is read, and then validates that value at the commit phase. If they do not match, the transaction aborts. In the case of 2PL, the stock object is locked when read, and no other transaction can modify the value. This is a conservative measure, to ensure the conditional branching done with the stock value does not get invalidated, but results in a loss of potential parallelism when concurrent transactions execute over the same objects. We can conclude that the problem stems from a lack of semantic information at the level of the database. This is where LSD comes in, by exposing the semantics behind the transaction and allowing the database system to make more informed decisions on what constitutes a conflict. Specifically, in the case of OCC, LSD introduces conditional validating at the commit phase, and for 2PL, it introduces conditional locking of an object. Another feature of LSD is that we do not need to observe a concrete state of the stock object, instead we only need to convey to the database that the value needed to make the condition (stock >= qty) true. This allows LSD to operate over an abstract state, made up of futures, that can be resolved at the commit phase. This means that the conflict window is reduced from possibly starting when the first object is accessed and ending at the end of the COMMIT phase, to being limited to the execution of the COMMIT phase.

The LSD algorithm introduces new methods and modifies the functionality of existing ones. To solve the problem of conditional branching using futures, LSD adds a new operation, **tx-is-true(condition)**. With this new operation, we can ask the database whether a condition that utilizes futures is true at that moment in time, allowing us to make a control flow decision. This operation only expos an abstract state of the database, and not its concrete state, in Algorithm 2 we only know if (stock >= qty) holds, and not the concrete database value for stock. This allows for concurrent transactions to modify the value of the stock object as long as it does not invalidate the condition (stock >= qty).

The **tx-read** operation, was redesigned to return only a future value and can also be called with a future key. The same is true for the **tx-write** operation, that can now write a value that is future to an object with a certain key, which can also be a future. With these modifications, the **tx-commit** and **tx-abort** operations also needed to be updated to deal with the new future values. While LSD adds new operations and modifies existing ones, the traditional database transaction operations continue to be available.

The last problem to address with LSD is how to perform computations using future values. As in the case of decrementing the stock object before updating it in the database. To solve this, LSD provides a method of defining the required computation and then pass that definition to the database, letting it execute the computation when the transaction commits and the futures values are already resolved.

With this in mind, the future values of LSD are functions that, when evaluated, compute a concrete value.

We can now conclude that, with the use of LSD, we can: (1) decrease the time window in which a transaction requires isolation, and (2) reduce the number of concurrent transactions that abort, or wait, when executing concurrently and need to guarantee a certain isolation level.

2.4.2 LSD Model & Design

The LSD model is composed by a Client, that executes the application code, and a Server running the database system. The Client interacts with the server via transactions, using the LSD API. The Client and Server components are logical, meaning they can co-exist in the same machine or be physically separated. The Server can be partitioned or replicated. With the LSD API we can, with minimal changes, implement high performance ACID transactions.

As depicted in Table 2.1, the LSD API modifies the existing **tx-read** and **tx-write** operations to support the passing of the application's semantics through futures. We can

Table 2.1: LSD interface

OPERATION	DESCRIPTION
tx-begin()	Starts a new transaction.
tx-read(key) → □	Returns □, a future for the value of object key.
tx-read(△) → □	Evaluates the future △ and returns □, a future for the value of the object that △ evaluates to.
tx-is-true(□) → $boolean$	Returns whether the condition □ is currently true in the database.
tx-write(key, □)	Updates object key's value to the value that □ will evaluate to.
tx-write(△, □)	Updates the value of the object that △ will evaluate to, to the value that □ will evaluate to.
tx-commit() → $boolean$	Attempts to commit the ongoing transaction.
tx-abort()	Aborts the ongoing transaction.

also see the interface of the **tx-is-true** operation, that is an addiction to the traditional database transactional API. Now follows a discussion on how each operation works.

tx-read Instead of returning the value of the request object and exposing the database state, with LSD this operation returns a future for the value. The transaction then uses this future as if it were the actual value, with the promise that the database will resolve its value, as late as possible.

tx-write Normally, this operation would receive a key and a value, and update the object referenced by that key with the given value. But because the **tx-read** operation returns a future, this operation also as to support future values. Adding to this, the transaction can also modify the future value before issuing the write. This is done by defining a function that expresses how the future value will be modified and is then evaluated by the database to resolve the concrete value. The new **tx-write** operation can also receive future keys and future values, which can be a function, that are also resolved at the commit phase, where the actual key and value are computed and the object updated.

tx-is-true A transaction may need to perform conditional branching based on the state of the database. In Figure 2, we can either continue with the transaction, if (stock >= qty) if true, or abort it if not. Because we cannot validate that condition on the transaction side, as stock is a future value, the database needs to expose a method that validates it for us. The **tx-is-true** operation receives a condition over the database, i.e., a function based on futures that, when evaluated, returns a boolean. When the database evaluates the value of a **tx-is-true** condition, it does so over an abstract and, not a concrete, database state because the concrete values of the futures are not returned. This is essential to preserve the isolation between concurrent transactions. In relation to enforcing the semantics of this operation, we can either validate the condition at the commit phase, in the case of OCC, or ensure that the result of the condition does not change until the commit phase, in the case of 2PL.

In relation to the operations **tx-commit** and **tx-abort**, their only modifications are at the execution level, having the same interface as their traditional homonyms.

The functions that use futures, described earlier, work in two stages. First, we need to resolve the future reads on which the function depends. Second, we execute the functions' logic. The functions are composed by utilizing a library of pre-defined operations. As an example, in Figure 2, the operation that decrements the stock would be represented using *sub(stock, qty)*.

The final problem with the implementation of LSD is when futures are used as keys for the **tx-read** or **tx-write** operations. The solution for the **tx-read** is to resolve the keys immediately so that the object being read is known. The reason behind this is that if the key was not resolved, we would have to deal with the problem of futures of futures, and that would require a possibly lengthy chain of resolves at the commit phase. For the **tx-write**, the future keys are kept unresolved, as the future key of an object being written might depend on the database state.

2.4.3 Concurrency Control

Now we will discuss how to adapt two concurrency control protocols, OCC [26] and 2PL [12], to make use of LSD. The main problems with the adaptation are the addition of futures, and how to handle them at commit time, and the introduction of the new **tx-is-true** operation that because it exposes an abstract state to transactions.

Regarding the two protocols, the general idea is to maintain three extra sets, adding to the traditional *read* and *write* sets, for a total of five sets. Two for the *read* and *write* futures, where they are kept unresolved until the commit phase, and a third *conditional* set to support the conditions in the **tx-is-true** operation.

Here we go over the four operations, that are equal to both protocols, and explain what they do.

tx-begin This operation initializes the five sets previously discussed.

tx-read(key**)**→ □ Creates a new future value for the given key, adds the key to the read set, and returns the future value.

tx-write(key, □**)** Adds the value □ into the write set.

tx-write(△, □**)** Adds the value □ into the future write set. The future value □ will later be assigned to the future key △.

Follows a discussion of the inner workings of the OCC protocol. In this protocol, each item is associated with a version. When the database receives a read, it records the object key and current version in the read set. For the writes, they are buffered in the write set. At the commit phase, the objects in the read set are atomically verified to ensure they remained unchanged, and then the writes are applied to the database, incrementing each object's version. To execute this atomic operation, we need to lock the write set, validate the read set, perform the pending/buffered writes and release the locks.

Next, we go over the operations that are specific to OCC.

tx-read(△)→ □ Resolves the future value △, adds the object's version to the read set, and executes **tx-read(**$value$**)**.

tx-is-true(□)→ $boolean$ Resolves the future values in □, adds the result to the condition set and returns the result.

tx-commit())→ $boolean$ First, it resolves and locks the keys in the future read set, then it locks the keys in the write set and future write set, next it validates the read set and condition set, finally it resolves the buffered future values and updates the required objects, releasing the locks in the end.

In the 2PL protocol, when an object is accessed, a lock is used to prevent conflicting transactions from modifying that object and breaking isolation. To implement the **tx-is-true** operation, conditional locks were used. A conditional lock has the write and read modes of a traditional read-write lock, plus two additional modes. (1) A **read condition** mode where a condition is associated with the lock and transactions can still modify the value if the condition remains the same. (2) A **write value** mode where we register the value intended for an object, if a read condition lock exists for an object, and the condition remains valid for this new value, the write mode can acquire the lock.

Next, we go over the operations that are specific to 2PL.

tx-read(△)→ □ Resolves and locks the future key △, reading its value and then executes a **tx-read(**$value$**)**

tx-is-true(□)→ $boolean$ Observers and locks the value of each key in □, calculates the value of □, downgrades the acquired locks to read condition using the result.

tx-commit()→ $boolean$ First, it locks and performs the delayed reads, resolving the future read set, removes the conditions from the **tx-is-true** operation, resolves the future keys in the future write set, and then it resolves all future values in the future write set and write set.

Transactions can also span multiple nodes in the case of a partitioned database. LSD also adapts the 2PC algorithm to support distributed transactions. In short, LSD requires an extra round of communication during the commit-prepare phase in the case of a transaction that values to resolve in its future write set. In the commit-prepare phase, each node resolves and returns the future read set. The extra round involves using these values to resolve the future write set and send the result to the required nodes. It was also proposed the possibility of removing this extra round if: (1) it's possible to identify the future-key's partition without resolving it, (2) the futures on which the future-key depends are from the same partition it belongs to.

2.4.4 Current Status

In this section we will go over the existing LSD prototypes, their components, limitations, and evaluation results.

1st Prototype

The first prototype consists of a transactional Key-Value Store running in a distributed setting. Each server is running Apache Thrift [16] with the data being stored in disk using the RocksDB [13] database. Each database partition is on a different machine, with each machine only having one thread. The Clients support the traditional transaction interface as well as the LSD API, including the classical OCC and 2PL concurrency control protocols, and the LSD aware variations. Clients are responsible with routing requests to the correct partition. The prototype relies on a Client/Server integration model. Distributed transactions are supported using the two-phase commit protocol (2PC). The resolution of deadlocks, in 2PL or 2PC, is done by the wound-wait strategy.

The evaluation was done by running the TPC-C benchmark in high and low contention scenarios. This prototype was able to reap higher throughput and lower latency under contention due to fewer aborts. In OCC, the abort rate went down from 92% to 8%. It was also observed that LSD-aware 2PL incurs more overhead than the OCC variant, this being cause by the complex conditional locks. It was concluded that the OCC variant was the best alternative, showing the best results, while having less impact on performance when contention is low.

Some effort must be made in analysing both prototypes to reveal possible optimizations, while also looking into performing the evaluation tests with better machines and network infrastructure. Both insights will be discussed further in the next Chapter.

In this chapter, in Section 2.5, we present the current status of LSD and the work done so far, as well as the available prototypes and their limitations. In Section 3 we describe how improved the current prototype, the resulting architecture design and the methods for evaluating the new solution.

2.5 Current Status of LSD

Lazy State Determination was developed by, Tiago Vale, as part of his PhD [43, 42]. Although there are currently two working prototypes, described in Section 2.4.4, they both fall behind in terms of throughput, the focus now being the evaluation and optimization of LSD in a new setting.

With this in mind, in this section we will briefly present the model of LSD, the two available prototypes, discussing their limitations and how we intend to address them in order to produce a higher performance system that demonstrates the benefits of utilizing LSD.

2.5.1 LSD Model

As presented before, in Section 2.4.2, LSD has two key insights. The first is the use of futures to represent an abstract state of the database system, and the second is the passing of semantic information about the application to the database system, to provide finer granularity over objects using conditionals.

These insights complement each other. By using futures and not observing the concrete state of the database, we can delay the observation of the database state, possibly until the commit time, decreasing the time required to provide isolation between transactions. By reducing the time span of the window when isolation is necessary, we also reduce the conflict window, with transactions appearing to execute faster leading to less probability of conflict over highly contention objects. The passage of semantic information to the database exposes more concurrency between transitions by redefining what constitutes a conflict and allowing, previously conflicting transactions, to execute concurrently.

2.5.2 Prototypes

The two existing prototypes of LSD, presented in Section 2.4.4, demonstrate the possibilities of LSD, but fall behind in terms of performance.

The first prototype uses the RocksDB [13] in disk database as durable storage. In this prototype, for concurrency control protocols were implemented: traditional OCC, traditional 2PL, OCC with LSD, and 2PL with LSD. This prototype assumes a Client/Server interaction model, making no use of stored procedures. The server is running Apache Thrift [16] and exposes both the normal database operations and the LSD variants. Each Server is single threaded and runs one partition of the database. The Client routes the requests to the required server, based on the partition. It implements the 2PC protocol required for distributed transactions when a transaction accesses more than one partition.

The second prototype is running on a single server, using Seastar [34], and uses a hash-map as a means for non-durable data storage. In the Server, each thread is responsible for a database partition, supporting concurrent data access if objects are in different partitions. It supports two models of operation, single requests and stored procedures.

We can conclude that the focus of the first prototype is providing an evaluation on a single server, being the base of analysis of vertical scalability, where we scale by adding more resources to a single machine. While the second prototype operates over distributed servers and can be used to test the horizontal scalability of LSD, where we scale by adding more machines to the system.

2.5.3 Limitations

In this section, we are going to discuss the limitations of LSD, in terms of the algorithm, the testing environment used for the prototypes and the prototypes themselves.

In relation to the algorithm, we observe two main limitations. The first is the necessary addition of a new database operation, **tx-is-true**, which, together with the new future values, leads to mandatory modifications on the client side of the application. The second is the lack of support for SQL like queries, where the reply could constitute a range of values.

In relation to the testing environment, we can observe a few conditions that lead to a perceived lower performance from the prototypes, namely the small number of machines used during the tests and the network bandwidth. We can anticipate that the use of a few slow machines connected by slow network links can have a negative effect on the overall performance of the prototype, especially when the main metric is throughput. Adding to the low throughput, the lack of an upper bound for the tested system, means it is unclear how close, or how far, the observed results are from the theoretical maximum.

Finally, in relation to the prototypes, the first one uses a disk for the database, which while providing durable storage also imposes a considerable high latency on data access, which in turn leads to longer executing transactions and slower overall performance. A more general limitation is the overhead imposed by LSD when testing in low contention scenarios, which leads to lower performance.

Regarding the limited scalability of the current prototypes, it lays more in the available resources, rather than in their implementation. Still, we foresee that there is space for improvements on both prototypes.

3

Approach

3.1 Work Plan

Here we go over our work plan and how tackled each step, this work can be divided into four stages.

Code Analysis and Prototype Update §3.2 First, as we need to have a good understating of the current LSD prototype, we studied and analysed its implementation and became acquainted with the code base. This allowed us to get a more in-depth understanding of how the system was built, how to modify it and compile it, as well as setting it up for testing. We use this information to update the prototype to make it run on newer systems.

Performance Analysis §3.3 Second, we review the performance evaluation previously done, to enlighten us to possible areas of work. After, we designed and ran a new set of tests to better evaluate the previous LSD prototype and discover possible areas of work.

Prototype Optimization §3.4 Third, based on the outcome of the new evaluation procedure, we started to improve and optimize the prototype. This included changes to run the database storage in memory using a ram disk. We also implemented a new TPC-C benchmark written in c++, coupled with the use of a c++ client to further optimize the prototype. This also involved setting up the testing environment with the better machines and network infrastructure, taking advantage of the newly available hardware.

Validation and Evaluation §4 Finally, we validated our solution utilizing basic transaction tests as well as the TPC-C benchmark. Then, rerun all tests under this new and improved prototype and collected our results, discussing the improvements and limitations seen and possible future areas of work. We then present the benefits in dealing with high contention in transactional systems using LSD.

In the next sections, we will provide detail on the work done on each stage, explaining our process to achieve our main objective, to increase parallelism between transaction, by reducing contention, leading to increased throughput. We can achieve this by reducing the conflict window between transactions, which will reduce the probability that two or more transactions require the same lock or observe the same value simultaneously. This is the main objective of LSD, and we examine which areas need to be worked on to further improve this system.

3.2 Code Analysis and Prototype Update

First, we introduce the architecture of the previous prototype, by Vale [43, 42], shown in Figure 3.1, that provides a general overview of each component. In the following subsections 3.2.1, 3.2.2 and 3.2.3, we will go into more detail and discuss the implementation details of each of the part. This will include the language used as well as versions of libraries utilized. We will follow this with the changes done in an initial state of our development to make the previous prototype run in a more modern system and be compatible with the cluster provided by the Computer Science Department[1].

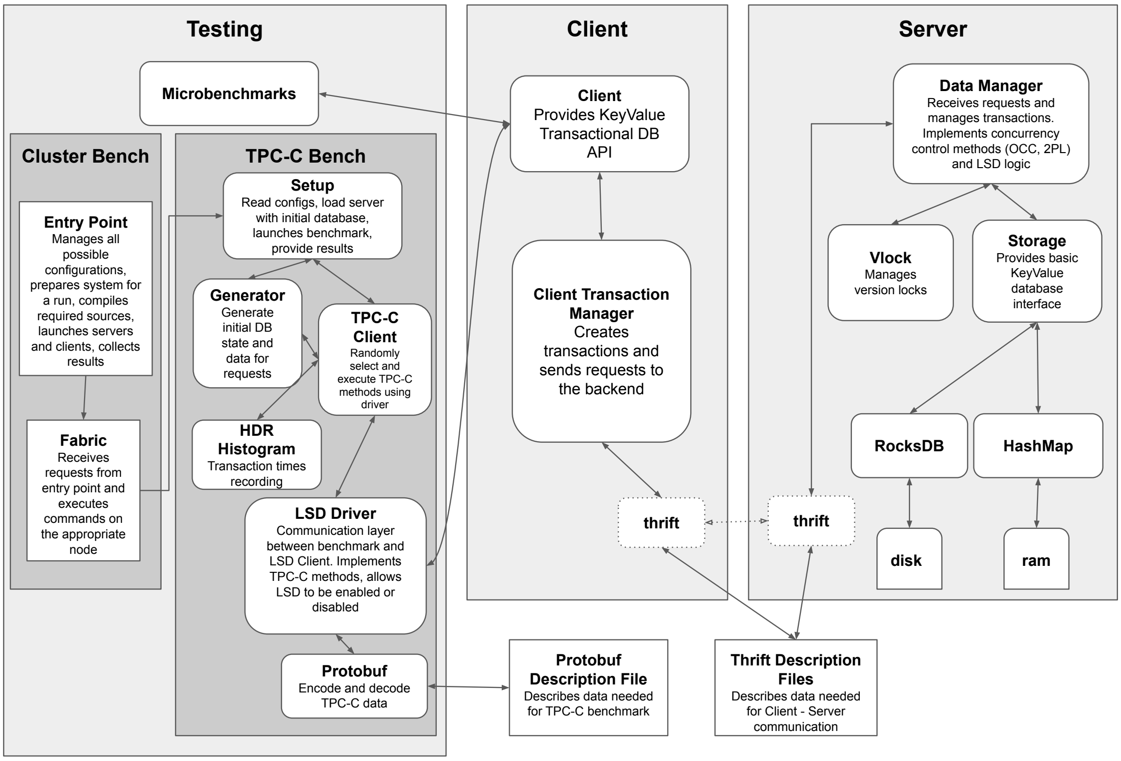

Testing
Microbenchmarks
Cluster Bench
Entry Point
Manages all possible configurations, prepares system for a run, compiles required sources, launches servers and clients, collects results
Fabric
Receives requests from entry point and executes commands on the appropriate node
TPC-C Bench
Setup
Read configs, load server with initial database, launches benchmark, provide results
Generator
Generate initial DB state and data for requests
TPC-C Client
Randomly select and execute TPC-C methods using driver
HDR Histogram
Transaction times recording
LSD Driver
Communication layer between benchmark and LSD Client. Implements TPC-C methods, allows LSD to be enabled or disabled
Protobuf
Encode and decode TPC-C data
Client
Client
Provides KeyValue Transactional DB API
Client Transaction Manager
Creates transactions and sends requests to the backend
thrift
Server
Data Manager
Receives requests and manages transactions. Implements concurrency control methods (OCC, 2PL) and LSD logic
Vlock
Manages version locks
Storage
Provides basic KeyValue database interface
RocksDB
HashMap
disk
ram
thrift
Protobuf Description File
Describes data needed for TPC-C benchmark
Thrift Description Files
Describes data needed for Client - Server communication

Figure 3.1: Architecture diagram of the prototype

[1] http://cluster.di.fct.unl.pt/

To better explain the architecture of the prototype We divided it into three areas, the server, the client, and testing. In a general overview, this solution supports the execution of multiple clients and multiple servers with a partitioned database, where the client and testing benchmark is implemented in python and the server in c++.

3.2.1 Server

Starting with the server side, it implements a transactional key-value store that functions as a non-blocking server capable of serving multiple clients simultaneously. Here we have three main components.

The **Data Manager**, the entry point to the server, receives requests from the client for simple **GET** and **MULTI-GET** operations, and its LSD variant **GET-LSD**, the **IS-TRUE** operation and **COMMITS** transactions, all using the Apache Thrift library [16] as the network communication layer. The **Data Manager** also implements two classical concurrency control methods, OCC (optimistic concurrency control) and 2PL (two phase locking) and their LSD aware versions. For distributed transactions, the database supports 2PC (two phase commit), as a means of coordinating the execution of transactions across multiple servers.

This means that the server can function as a normal transactional key-value database and as an LSD capable database. It is what allows us to test the performance impact of LSD against a normal database.

To solve possible deadlocks, the prototype implements a wound-wait strategy [4] for both 2PL and 2PC. The wound-wait strategy is used as a preemptive technique to prevent deadlocks. It allows a new transaction to wait if it is requesting a lock held by and older transaction, but forces the new transaction to abort if an older transaction requests a lock held by this new transaction. This way, only the most recent transaction, to request a lock on an item already held by another older transaction, can be aborted.

The **Data Manager** utilizes two other main components, the **Vlock** and **Storage**.

The **Vlock** is responsible for managing the locks utilized by the **Data Manager**, it provides a way to create locks, check for locked status and remove locks.

The **Storage** component provides a basic key-value database interface to the **Data Manager**, and it is then implemented either with **RocksDB** that utilizes an in disk database or with **HashMap** that stores its data in memory only. **RocksDB** is a high performance in disk key-value database. The use of this database for storage provides us with persistent storage for data, as well as a low latency data access utilizing memory as cache and then writing that data to disk using background processes. The **HashMap** is using an unordered hashmap for even lower possible bottleneck regarding data access, writes and reads, in the previous prototype this option was left unused.

As for changes made to make these components work in a more modern system, we updated the c++ version used from version 11 to version 17, while doing this we removed the dependency on the Boost library [2] used for things like a **shared_ptr**. The

Apache Thrift library was updated from version 0.9.3 to 0.13.0, and RocksDB from version 4.11.2 to 7.0.3. While updating these two libraries, we had to update the source code to accommodate changes done to the API of these libraries.

As a way of optimizing the server, we can compile it to only use OCC or only use 2PL, by setting different flags during compilation. This was done with the help of Bakefile (v1.2.2) [35], a native makefiles generator, used to create the different makefiles needed to compile each version of the server. As Bakefile proved to be very verbose in the way of defining different compilation targets we decided to use CMake [23] instead, as we would need to extend the number of targets later on, and is a more recent and used tool.

We made changes to allow RocksDB to run in memory only, to observe if there was a significant improvement, as we wanted to be able to test the prototype with the lowest amount of I/O latency possible. With this, the server can have three types of database storage, RocksDB in disk, RocksDB in memory and a hashmap.

As we were updating these libraries and introducing changes to the source code, we used this opportunity to do some refactoring to have a better overall organization of the code base. We also added new configuration options to the server to allow the control of the location of the RocksDB database.

We also complemented the existing transactional API methods with two new execution modes. The first, **NO-OP** (no operation), that simply replies to any client request with a default message. This was used to calculate the theoretical maximum RPC's, remote procedure calls, that the prototype was able to achieve, in its current state. This execution mode made sure that any transaction processing on the server side was skipped.

The other execution method, **NO-TX** (no transaction), was used to also skip the transactional processing, but it stills writes and reads data, transforming the prototype into a simple key-value database, without the use of transactions. This allows us to measure the impact of accessing and writing data utilizing each of the database storage options. Again, we wanted to measure the maximum amount of RPC's possible, while actually writing and reading data.

Finally, we updated several scripts, that were used to prepare a system to be able to compile and run the prototype, by updating the installation process and libraries used. We also updated scripts related to the compilation and execution of the server binaries.

3.2.2 Client

Next, we have the client. It has two main components, the **Client** itself and the **Client Transaction Manager**. The **Client** provides an extended traditional key-value transactional database interface, with **begin-transaction**, **get**, **multi-get**, **write**, **is-true**, **commit**, and **abort** methods.

This component then makes use of the **Client Transaction Manager** to actually create the transactions, and requests, and send them to the server using the Apache Thrift library. It is also used to manage partitions across multiple servers, by providing two ways of distributing data across the available servers. This distribution, or partitioning, was done by hashing the keys and dividing the data by the available servers or using an application aware partitioning method, which in your case involved distributing the TPC-C data in an effective away to minimize communication rounds for each transaction, by trying to group all data used in a TPC-C operation in a single server.

Both the server and client make use of the **Thrift Description Files** to encode the transaction data, including the operations that use future values, before sending it through the network. This library allows the definition of the required communication messages in an implementation independent language, allowing for easily serialization of our messages into a binary format, which is a highly efficient way of transmitting data over the network, and to easily add implementations in different programming languages.

The previous implementation of the client was done in python version 2.7. The reason behind the use of python was so the client could be used with an existing python implementation of the TPC-C benchmark, facilitating the work required to benchmark the prototype. To get everything working, we updated the **Client** and **Client Transaction Manager** to use the newer version of python 3.8. This required the update of the source code to work with the new python API.

We also completed the implementation of an existing c++ client. This implementation was incomplete and required an extensive rewrite of the way messages were being created and structured using the Apache Thrift library.

3.2.3 Testing

Finally, we have the testing group, this consists of every tool used to test the prototype.

We have **Microbenchmarks** that involves every test done outside the TPC-C benchmark [40], we will go into more detail about these tests in Section 3.3 and Chapter 4. These tests made direct use of the **Client** component to create transactions to add, modify and delete data.

Next is the **TPC-C Bench**, this is the implementation of the TPC-C benchmark specification. It has six main components.

First is the **Setup** component. Here is where all the configurations for a run are read and used to configure the system. This involves loading the database with the correct initial state, making use of the **Generator** component, and initiating the run by using the **TPC-C Client** component.

The **Generator** is used every time there is the need to randomly generate data for the benchmark. This can be to create an initial state for the database, as well as generating data for a TPC-C request. The **Generator** makes sure that the random data generator fits within the TPC-C specifications, we will go into more detail later in Chapter 4

The **TPC-C Client** is where the TPC-C method is randomly selected as the next request, it is prepared, using data from the **Generator**. It is also where the transaction times (latency), types (TPC-C methods) and number of successes (throughput) is measured with the help of the **HDR Histogram** [38], a library used to measure values with high dynamic range.

The **TPC-C Client** then uses a driver, in our case the **LSD Driver**, to communicate with our database. This driver implements the TPC-C methods in a way that most makes sense to our particular solution, this includes making use of LSD's futures to optimize when possible the transaction and converting the TPC-C data into a key-value pairs. This is where the use of **Protobuf** comes in. With it all the data that is not needed to be interpreted by the server, i.e., data that is not used for futures, is encoded into a single key-value, where the key is generated by grouping multiple information into a single string, and the value is the resulted string from the encoding using **Protobuf**. For example, for a single customer, we have can have a first name, last name, street address, zip number, phone, etc. As we do not need to have every field in its own key-value, we group all this information into a single value and create a key for it that represents this specific client.

Lastly, we have **Cluster Bench**. Here is where everything that is needed to run our tests in the computing cluster. This was implemented using the Fabric library [15] and is divided into two components. In **Entry Point** is where the configuration settings are read and then utilized to configure the tests. This includes preparing all the cluster nodes for the next run by compiling the correct server version in the correct node or nodes, loading the database with the correct data for the run and lunching the clients with the correct parameters and coordinating their start and end. At the end of a run, it collects all the HDR data from the clients and compiles it into a single result for the run. This result includes all the transaction times, number, and types of transactions, number of aborts, mean, median, minimum, maximum latencies and calculates the throughput (transactions per second).

Everything that needs to be run on a specific node, this can be calling a compile script, moving data, launching executables, uses the **Fabric** component. This component allows the definition of roles for each node, for example, the role of server or client, and then receives instructions from the **Entry Point** and executes them in the correct node, according to their role.

To facilitate the execution of the TPC-C benchmark, as the database loading stage can take hours to complete, the server can store the in memory data to a file and load this data back into memory. This way, consecutive tests can be done much faster. The same can be done for the RocksDB database files. We can create a snapshoot of the state of the database after the loading phase finishes and then use it for subsequent runs.

Both the **Cluster Bench** and **TPC-C Bench** were implemented in python version 2.7 and had to be updated to python version 3.6 making the required changes for the new API.

The **Cluster Bench** component had to undergo some heavy changes. First, the Fabric library was updated from version 1.10.3 to version 2.7.0, again the code had to be adapted to the new API. Second, we had to modify the existing implementation in order to accommodate the new computing cluster configuration. We later removed this implementation entirely and wrote a new one from the ground up with the new computing cluster in mind.

The last library updated was Protobuf, from version 3.1.0 to 3.12.3, this also required some minor changes to the **Protobuf Description File**.

3.3 Performance Analysis

In this phase of our work, we focused on the performance analysis of the existing prototype as a means of detecting possible areas of work for the next phase. To achieve this, we will look at the previously obtained by Tiago Vale and discuss its shortcomings in Subsection 3.3.1, and then present a series of microbenchmarks we developed and ran to test the impact of modifying different areas of the prototype in Subsection 3.3.2.

3.3.1 Previous Prototype Evaluation

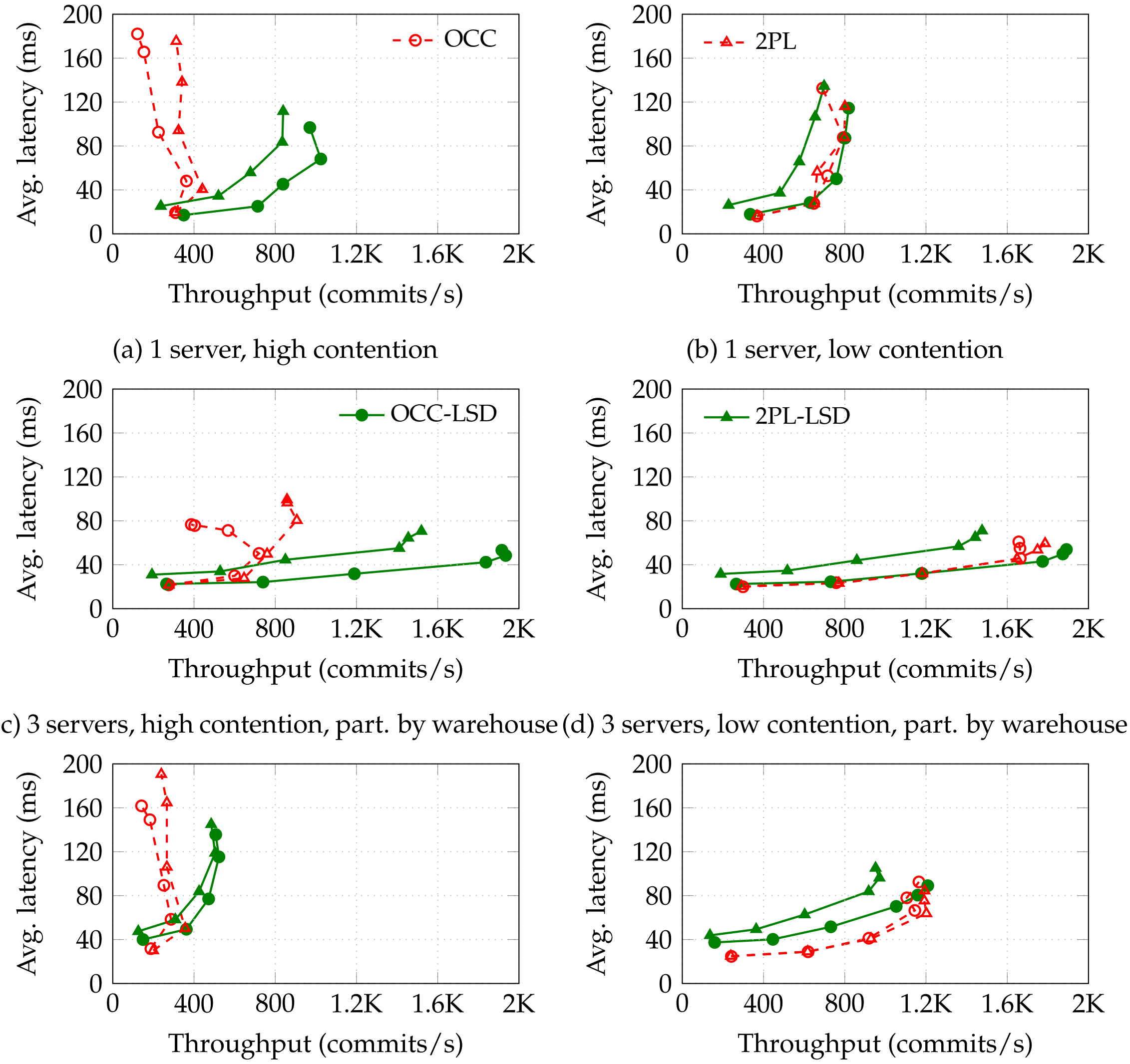
The results in Figure 3.2 are our baseline and will be used to measure and analyse our improvements when we present our new results in Chapter 4. They were obtained with a previous prototype, by Vale [43, 42], running the **RocksDB** database in disk and the client, as well as the TPC-C benchmarking, implemented in python. The previous environment had a 1Gbps network speeds and each server had a 2Ghz Intel Xeon E5-2620 processor, 32 GB of RAM, and a 7200 RPM hard drive. The tests vary between high and low contention scenarios, alternating between the use of OCC and 2PL, with LSD on and off. In these tests, the number of clients varies from 1 to 88. We will go into more detail about the specifics of the TPC-C benchmark in Chapter 4.

Figure 3.2a and Figure 3.2b depict tests running in just one server, and in Figure 3.2c and Figure 3.2d we have three servers partitioned by a TPC-C aware sharding, and finally in Figure 3.2e and Figure 3.2f we also have three servers, but this time partitioned by a simple hash based sharding.

In Figure 3.2a we can see that by using OCC with LSD we were able to achieve a peak throughput of 1k transaction per second, which is a 6.5× improvement over using just OCC. This while having a 40% of the original latency, from 175ms to 70ms under the same conditions, e.g., number of clients.

The 2PL version without LSD performs better under high contention than OCC without LSD, and we see a smaller improvement when using 2PL and LSD, compared to OCC without LSD and with, of 2.5× the throughput and 66% of the latency, achieving a peak of 340 transactions per second and 120ms latency without LSD and 850 transactions per second and 80ms latency with LSD.

These results are to be expected, as LSD functions better under a high contention scenario, where the reducing the conflict windows is most advantageous as we have the most amount of transaction trying to access the same data.



(a) 1 server, high contention

(b) 1 server, low contention

(c) 3 servers, high contention, part. by warehouse

(d) 3 servers, low contention, part. by warehouse

(e) 3 servers, high contention, part. by hash

(f) 3 servers, low contention, part. by hash

Figure 3.2: TPC-C benchmark using: 1 server with high (a) and low (b) contention; 3 servers with TPC-C aware partitioning (b,e); and 3 servers with partitioning by hash (c,f).

Locking at Figure 3.2b, we can see the results for the same test, but under a low contention scenario. Here, using LSD does not show any improvements, with the tests reaching a peak of 800 transaction per second at 120ms of average latency. The exception being 2PL with LSD where the overhead of running LSD is detrimental to the performance and results in a 12.5% loss in throughput and a 15% higher latency.

In a distributed setting, in Figure 3.2c and Figure 3.2d, using a TPC-C aware sharding policy, we can see similar results, where in a high contention scenario LSD can provide a 1.8× improvement when using 2PL and a 5× improvement when using OCC. This time, thanks to 3 servers and better distribution of the workload between them, we can reach a peak of nearly 2k transactions per second with 50ms latency with OCC and LSD. This is a 2× improvement in throughput and a 30% reduction in average latency when compared to a single server setup.

But in the low contention scenario, we see no significant performance difference when using OCC with LSD, while not using LSD shows improvement for both OCC and 2PL. The 2PL and LSD variant still underperforms compared to just using 2PL.

The final test, in Figure 3.2e and Figure 3.2f, show us that simply having a random, equal distribution of data across the 3 servers is detrimental to the performance of all variants. This is because, by not optimizing the data location, the clients will have to execute more communications rounds with the servers, resulting in higher latency and reduced throughput. Still, thanks to LSD we can still see benefits in a high contention scenario, with 2.2× the throughput when comparing OCC and 1.4× with 2PL. Again, with low contention the OCC LSD variant can still perform the same as the non LSD variant, showing that the benefits of LSD can still overcome its overhead.

In conclusion, based on these results we can say that the previous prototype can provide better performance when using LSD in all scenarios, OCC with LSD being the best variant, being able to provide a peak of 1k transactions per second with an average of 70ms latency, a 6.5× improvement in throughput and a reduction of 60% in latency, with a centralized database and 60 clients, in a high contention scenario. With a distributed database, we get a peak of near 2k transactions with an average of 50ms of latency, a 5× improvement, also in a high contention scenario.

With these results in mind, our next goal was to see how much we could improve the prototype's throughput, while still keeping latencies low. We discuss our approached in Section 3.4.

3.3.2 Microbenchmarks

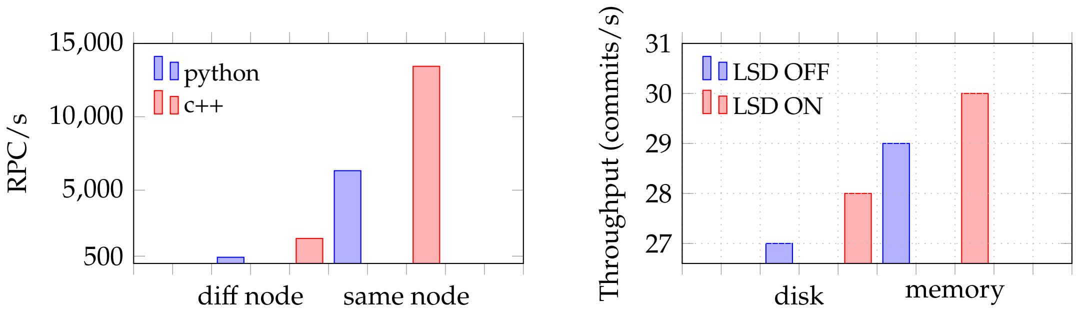
Here, we will discuss the microbenchmarks we did to help identify the possible sources of problems related to our goal of increasing throughput.

The first test we executed, whose results are in Figure 3.3a, was done so we could observe two things. The first was the impact of the network bandwidth, or lack of, had on performance, and second was the difference between using a client written in python vs. a client written in c++.

Here we are trying to see if we are network bound or client-performance bound, or both. For this test, we ran a simple loop that executed a basic transaction, it read a value, updated it and committed. The server was running with the **NO-OP** mode, so we could remove any possible bottlenecks.

This test was run in two **charmander** nodes of the computing cluster, described in detail in Table 4.1 in page 43, and we measured the RPC/s resulted from averaging three runs.

When running both the client and the server in the same node, we observed a 15× increase in the throughput for the python client, and a 32× increase for the c++ client compared to running them on different nodes. This means that network latency plays a big part in the resulting throughput.



(a) Running with a python vs a c++ client

(b) Database in disk vs in memory

Figure 3.3: Difference between same vs different node execution, python vs c++ and disk vs memory.

The other observation is that the performance increase between having a python client and a c++ client varies depending if we are running the client and server on the same node or not. While running on separate nodes, we see an increase of 4×, and only an increase of 2× when running on the same node. This means that the increase in throughput provided by having a c++ client is not as extreme as the increase from reducing the network latency, but we still see that it does provide and improvement.

This tells us two ways to improve performance. One is to utilize the fastest possible network infrastructure for our TPC-C benchmark tests, something that we will expand more next, and second is to use a c++ client for the TPC-C benchmark.

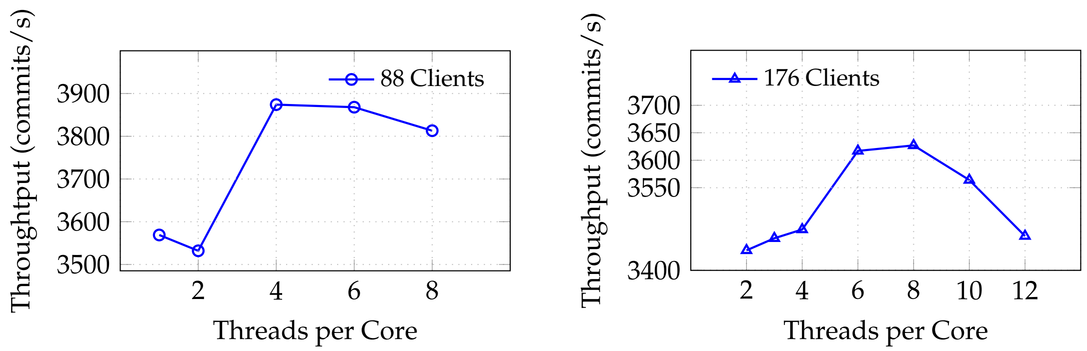
In the next test, in Figure 3.3b, we test the impact of running a server using a disk-based database and a memory-based database. Here we wanted to see if we are server-performance bound, more specifically I/O bound. We ran the TPC-C benchmark on the **charmander** node, with one client and one server on the same node, and alternated running the RocksDB database in disk and memory using a ramdisk. We also ran the test with LSD on and off.

This resulted in just a 7% improvement, for both LSD on and off. Because of this small improvement, we decided to run the TPC-C benchmark, with RocksDB in disk and memory, but also utilize a hashmap to further reduce possible bottlenecks, and compare again the performance.

Our thought process being that a hashmap, while providing no persistence for the database, has a much lower performance impact on I/O operations, and our objective is measuring the maximum performance possible. These results will be showed in Chapter 4.

The final micro-benchmark involved discovering the final requirement for the execution of the TPC-C benchmark.

We wanted to know the impact of running the test in a mix of 1 Gbps and 10 Gbps network speeds and just a 10Gbps network. We also needed to know how many clients threads we can run in each node. More specifically, we want to know how many clients threads per CPU core can we run before we start seeing a performance decrease. We



(a) Throughput of 88 Clients and 1 Server

(b) Throughput of 176 Clients and 1 Server

Figure 3.4: Testing how many threads of clients per CPU core can be used.

will call this value **threads per core**. These results are shown in figures Figure 3.4a and Figure 3.4b.

For this test, we ran the TPC-C benchmark with a hashmap and with OCC as the concurrency control method and with LSD on. We wanted to, again, provide the maximum performant configuration to create the maximum load for the system.

In the first test, we utilized 88 clients and spread them across 5 nodes, varying the **threads per core** value. The node used for the server was **charmander-3** and for the clients we were using nodes **charmander-4**, **charmander-5**, **squirtle-2**, **squirtle-3** and **shelder-1:32**, in this order, more information on the nodes is available in Table 4.1. We measured the throughput of the 3 runs and display the average.

In the first graph, Figure 3.4a, we can see that when we start using 4 threads per core, the throughput increases 10%, which corresponds to running all clients on only the **charmander** nodes which are all connected by a 10Gbps network. The same is true on the second graph, Figure 3.4b, where we ran the same test, but with 176 clients, and at 6 threads per core, when all clients are on the same 10Gbps network, we see an improvement of 6%.

These results also tell us that using more than 8 threads per core starts to show a decrease in performance, this because the nodes executing the clients cannot handle more than 126 ($8 \times 16 cores$) clients.

These micro-benchmarks gave us important insight as to where we needed to focus our next phase of work and the best way to set up our testing environment to achieve the best possible results. For our next phase of work, in Section 3.4, we decided it was important to focus on making sure we use the best network possible between nodes, use the c++ client for all benchmarks, requiring the addition of a TPC-C benchmark in c++, and finally, the use of in memory database storage.

3.4 Prototype Optimization

With the knowledge obtained from the previous phase of work, we were now ready to implement the changes necessary to improve the throughput of the prototype.

The first optimization was in the testing environment. To ensure we used the best possible network connection between nodes, we decided to only use the **charmander** nodes during the TPC-C benchmark. Using the new computing cluster also involved redesigning and implementing the **Cluster Bench** component shown in Figure 3.1. We keep the same overall architecture but changed the implementation, while still using the python Fabric library, as the performance of this component did not interfere with our testing. The main changes were in regards to the way the source code was compiled and executed in each node, the way we loaded the database snapshoots into the server or servers, and the way clients were executed and coordinated to run and stop at the same time, as well as collecting and compiling all results for the run.

Then, to remove bottlenecks on the client side, we decided to run our tests using just the c++ client. For this, we had to finalize the implementation of the c++ client by adding two sharding methods as a means of partitioning the data by the available servers. The first method is a simple application-agnostic partitioning method where we just use the hash of the key to ensures equal distribution of data by the available servers. The other was an application-specific partitioning method, in this case, specific to the TPC-C benchmark. It shards all the data by warehouse ID, except item and history, which are sharded by item ID and client ID. This ensures a better partitioning of the data. It tries to group the data needed for each transaction on a single server, this way each client only needs to access a smaller number of servers, leading to fewer communications between client and server, and leaving other servers free for other clients.

This meant we also had to develop a TPC-C benchmark in c++, requiring the design and implementation of a new **TPC-C Bench** component, following the architecture shown in Figure 3.1. For this, we found a basic implementation of the TPC-C benchmark in c++[2]. This implementation did not have a client-server interaction model, did not have the ability to run multiple clients, or multiple servers, did not use transactions for the operations, did not make any measurements in a per-request manner, e.g., for each type of operation it did not measure, commits, aborts, latency. We took this base TPC-C implementation and implemented a client server model, implemented a new driver, **LSD Driver**, for our prototype that allowed the TPC-C operations to be executed with LSD on or off and make use of our c++ client to create transactions and send requests to the server. At the same time, making it possible to run multiple clients and multiple servers.

When executing with multiple clients, we added a type of barrier to make sure all clients are online and ready to start before initiating the benchmark run. We added the ability to set the necessary parameters to modify the benchmark to our needs, this includes number of clients, warehouses, partition type, execution time, and warm-up time. We

[2] https://github.com/evanj/tpccbench

also used the HDRHistogram library to keep track of transactions times and types, and whether they succeeded or not.

We wanted to remove as much bottleneck as possible from the I/O operations, and as we already saw some minor improvements by just running the RocksDB database in memory, we decided to add the option to run the database without using RocksDB and in turn use a hashmap. This involved using the already implemented Unordered Map in the previous prototype and simply add a configuration option to the server to allow its usage as the database, as it was previously unused. We also had to correct some of its implementation, more specifically in the way a batch of writes as being handled, as the previous implementation generated errors.

To summarize, our contributions are comprised of updating the code base to use newer versions of, and also different, libraries.

- The design, implementation, and testing of different parts of the prototype to identify possible problems.
- The redesign and implementation of key components to utilize a more performance oriented language.
- And the design and implementation of new testing and benchmarking methods to make use of the new computing cluster.

With this, we were now ready to move to the final stage of work, where we validate and evaluate our updated and improved prototype, we will discuss our finding in Chapter 4.

4

Validation and Evaluation

In this chapter, we will present the testing environment used to validate and evaluate the finalized prototype. Then we will go into detail on how we validated our prototype and present all the results of our evaluation.

4.1 Testing Environment

Here we will go over the testing environment, how it was designed, setup and executed.

The testing environment need to provide us with a few metrics, discussed previously in Subsection 3.3.2. First, we needed to test the maximum RPC's capable by our use of Apache Thrift for the communication layer, and the effect of the network on this metric. Second, we needed to know the impact of the I/O latency in accessing and writing data. Third, the weight of running a transactional system with and without the use of the LSD API. Fourth, and finally, the performance improvements of having a partitioned database.

To measure these metrics, and validate our implementation, we implement several micro-benchmarks that allowed us to:

- Test the safety of our system in the presence of concurrent transactions.
- Test the communication layer, by just sending simple get requests and measuring the RPC's.
- Test the weight of the transactions processing by running the system with and without isolation.

We also needed to develop a new testing deployment and execution system. We made use of simple scripts and the Fabric [15] library to be able to run the tests in the computing cluster provided by our Computer Science Department. The composition of this cluster can be seen in the Table 4.1.

We used this testing system to be able to execute the TPC-C benchmark in our computing cluster. For a single run of the TPC-C benchmark the following had to happen:

1. Define how many clients, servers, nodes to be used. And what variations of the behaviour of our system will be used. This includes whether OCC, 2PL, LSD is being used, what type of database, what type of isolation, the sharding policy. As well as the benchmark configuration, the level of contention, defined by the number of warehouses, the execution time, warm-up time.

2. As all the nodes in the cluster share the same home folder data, no files have to be copied. The next step is compiling the server and client in the required nodes using the appropriate library versions and flags.

3. Next, we need to launch the servers and clients, in this order, with the correct parameters.

4. Following by loading the correct database file into the correct server/servers.

5. After this, we can start the benchmark and wait for every client to finish their execution and export of their results.

6. We collect and compile all results to generate the final results of the run.

7. Now we kill all processes and can prepare the environment for the next round.

By focussing on creating a complete testing system, we can easily run tests with various configurations, allowing us to select the best settings for the benchmarks. We can also easily run new tests every time we update or implement new features and see the effects they have.

This testing system was used to evaluate our prototype and the results can be seen in the next chapter, where we go into more detail on the types of tests executed, how the environment was configured and what machines were used.

Table 4.1: Technical description of the computing cluster.

Cluster	Nodes	CPU	Cores	Memory	Network
charmander	5	AMD EPYC 7281	16/32	128 GiB DDR4	2 x 10 Gbps
squirtle	4	2 x Intel Xeon E5-2620 v2	12/24	64 GiB DDR3	2 x 1 Gbps
psyduck	3	Intel Xeon X3450	4/8	8 GiB DDR3	2 x 1 Gbps
shelder	1	4 x AMD Opteron 6272	32/64	64 GiB DDR3	2 x 1 Gbps
magikarp	1	8 x AMD Opteron 8220	16/16	27 GiB DDR3	2 x 1 Gbps
oddish	1	Intel Xeon E5-2603 v2	4/4	16 GiB DDR3	2 x 1 Gbps
bulbasaur	3	2 x Intel Xeon E5-2609 v4	16/16	32 GiB DDR4	2 x 1 Gbps
gengar	5	2 x AMD Opteron 2376	8/8	16 GiB DDR2	2 x 1 Gbps

4.2 Validation

In this section, we will go over the validation of the current solution, how we ensure the system is correct and produces the expected results.

4.2.1 Basic Tests

We started with some simple tests. Their purpose was to validate that simple, single transactions updated the database in a correct way. These tests include transactions with a single read operation, with reads and writes (updates), and also delete operations. We developed a client for our system that executed these simple transactions. We know that the prototype is producing correct results because we have developed and implemented these tests, and know the expected results that result from them. Next we repeated the tests, but with concurrent transactions. In this case, we could have two, or more, transactions updating the same key simultaneously. The expected and observed behaviour differed when we used OCC or 2PL. As these tests all proved to be successful, we can, from a practical standpoint, say the database system functions correctly and still provides the ACID properties. Again, because we developed and implemented the tests we know the possible results that can be obtained from running them, and we always obtain the correct ones.

4.2.2 TPCC Tests

The next step in testing involved using the popular database testing benchmark TPC-C [40]. This benchmark is utilized to test the performance of online transaction processing systems, i.e., systems that need to respond immediately to user requests and do so in the presence of multiple concurrent transactions. The benchmark simulates a multi-warehouse database system containing nine tables. The mains ones being the warehouse, items, stock, district, orders and customers tables, which are used when executing a series of transactions. The five operations provided by TPC-C are New-Order, Order-Status, Stock-Level, Payment and Delivery. Each has a different probability of execution, defined by the TPC-C standard, and each operate over one or more tables. Using this benchmark allows getting results that can be compared with other solutions while testing our system in a high contention setting where we can best see the benefits of LSD, and also in a low contention setting, where we can see that performance impact of running when LSD is almost null. This also allows testing of the database integrity. As each client of the benchmark knows what operations were executed successfully on the database, we can also test if the values added, updated or deleted show up correctly on the server side. Again, all tests executed showed the correct results were produced with this simulation of a real-world application, because the clients used in the tests never received incorrect or missing data.

4.3 Evaluation

In this section, we will present how we evaluated our solution and discuss the results. The TPC-C benchmark gives an in-depth insight on the performance of our solution under different circumstances. We can measure throughput (successful transactions), latency, including min, max, mean, and medium (total time between the start of a transaction to the response from the server), total number of transactions and number of aborts. We can control the testing environment in different ways.

- Define the level of contention, by changing the number of warehouses.
- Define the level of concurrency, by changing the number of clients.
- Define the level of workload for each server, by changing the number of servers and partition policies.
- Define the database I/O performance, by changing the type of low-level database used.

All test results discussed below were obtained using the computing cluster provided by the Computer Science Department[1]. This cluster is composed by several machines, each with different architectures and different network connection speeds between themselves, as showed in Table 4.1.

4.3.1 TPCC Benchmarks

Here we will present our results using our improved prototype. In Subsection 4.3.2 we compare the difference between running the **RocksDB** database in disk vs memory, and the use of a **hashmap** using only one server. In Subsection 4.3.3 we show the results of running a partitioned database across multiple servers using only a hashmap as the database. We will now discuss each in more detail.

These results were obtained using the TPC-C benchmark running on the previously mentioned computing cluster, see Table 4.1 in page 43, using only the **charmander** nodes.

We present results for several tests, displaying the average latency on the y-axis using a logarithmic scale and the throughput on the x-axis. Each dot represents the average between 3 runs with a varying number of clients, starting at 1, 4, 8, 16, 32, 64, 128, 256 and 512. The test runs not using LSD are represented by red, while the ones using LSD are green. 2PL is represented by triangles and OCC is represented by circles.

We test the performance of our solution using the TPC-C benchmark using the default values, only changing the number of warehouses: 1 for a high contention scenario and 16 for low contention.

In Figure 4.1, Figure 4.2, and Figure 4.3 we have test results for different database locations and types, as well as the comparison between OCC and 2PL and the LSD variants,

[1] http://cluster.di.fct.unl.pt/

in high and low contention settings, all of these results for a single centralized server. Then, in Figure 4.4, Figure 4.5, and Figure 4.6 we have the results using only OCC with LSD, with a hashmap as the database, in a high and low contention setting, with multiple servers, varying from 3 to 9. Comparing the difference between a TPC-C aware shading policy and a simple hashing.

4.3.2 Single Server

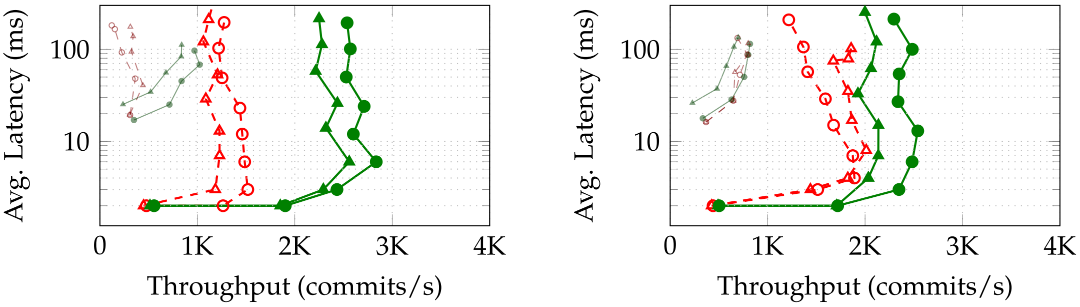
Now we will go over in detail about the results of our tests using 1 server and comparing the different database locations and types. First, we will discuss running RocksDB in disk, and compare these results with the results from the previous prototype, then we show the same tests with RocksDB running in memory and finally using a hashmap as the database.

4.3.2.1 RocksDB in Disk

In Figure 4.1a, we are testing using the RocksDB database running in disk. Represented in the smaller circles and triangles, we have the results from the previous prototype, already discussed in Subsection 3.3.1.

We can see LSD provides a 1.9× increase in throughput for both 2PL and OCC non LSD variants, with a peak of 2.7k transactions per second for OCC with LSD and 2.5k for 2PL with LSD at 25ms latency. While in Figure 4.1b the improvement is less, LSD can still provide better throughput, specially with OCC, where we see a 1.3× increase in throughput, with 2PL this is only a 1.1× increase.

We see, again, that LSD cannot provide the same benefits in a low contention setting as in a high contention setting, but it still provides better performance, meaning that with the use of a more performant client, written in c++, we were able to reduce the overhead of running LSD, even in the case of 2PL with LSD, which previous showed worst results compared with just 2PL without LSD in Figure 3.2b.



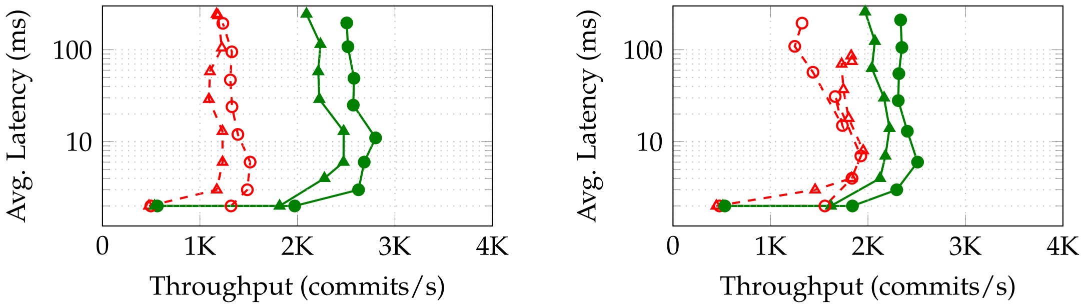
(a) In Disk, 1 server, high contention

(b) In Disk, 1 server, low contention

Figure 4.1: TPC-C benchmark using 1 server with RocksDB in disk.

4.3.2.2 RocksDB in Memory

Next, we have the same tests, but with RocksDB running in memory. In both Figure 4.2a and Figure 4.2b we can see that the results are very similar to the previous ones, meaning that just running the RocksDB database in memory does not result in better performance. This can be attributed to RocksDB caching the most requested items in memory, whether it is saving data to disk or memory, and later on writes the changes to disk in using background processes. Since the database size is at most 1Gb, RocksDB can cache the whole TPC-C database in memory, negating the I/O latency improvements of running RocksDB in memory. To be able to see improvements of having a higher performant low-level database that provides faster I/O operations, we need to utilize a different type of data storage. We did this by utilizing a simple hashmap.



(a) In Memory, 1 server, high contention

(b) In Memory, 1 server, low contention

Figure 4.2: TPC-C benchmark using 1 server with RocksDB in memory.

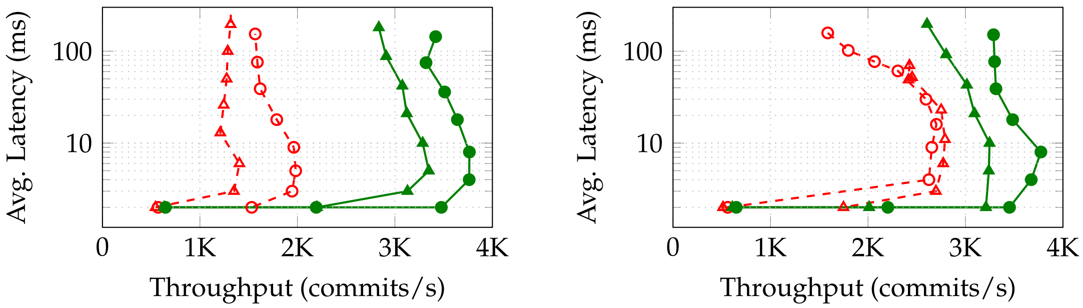
4.3.2.3 Hashmap

Now, in Figure 4.3a and Figure 4.3b, we can see the big improvement of having faster I/O operations to access and write data on the server. Comparing the increase in throughput between basic OCC and 2PL, and using LSD is similar to previous the previous test, but now we can achieve a peak throughput of 3.8k transaction per second with OCC and LSD in both a high and low contention setting, at just 8ms average latency. This is a 1.4× increase compared to using RocksDB and a 1.9× increase compared with the previous version.

4.3.2.4 Discussion

In conclusion, for every type of database used, LSD provides significant improvements in throughput and latency, and by using the new c++ implementation in conjunction with a better computing cluster, we obtain a much higher throughput, showing good vertical scalability.

We also see that in low contention settings the improvement is much lower as it is to be expected having in consideration that there are lower changes of different transactions using the same data, and being that LSD is not meant to deal with this type of senary, low



(a) HashMap, 1 server, high contention

(b) HashMap, 1 server, low contention

Figure 4.3: TPC-C benchmark using 1 server with a hashmap.

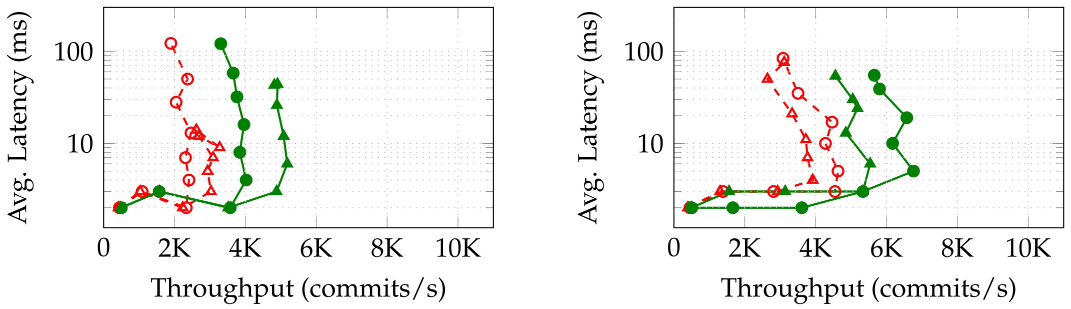
contention, we also conclude that even when not needed running LSD does not introduce a bottleneck in the system.

4.3.3 Partitioned Server

Now, in Figure 4.4, we will look at the benefits of running a distributed database. We selected the best performing configuration based on the previous test, using OCC with LSD, with the low-level database in a hashmap, to test the impact of having a partitioned database. We tested with 3, 6 and 9 servers, using a TPC-C aware sharding policy and simple hash sharding.

4.3.3.1 3 Servers

When using 3 servers in a high contention setting, LSD still provides a performance increase compared to not using LSD. Here, in Figure 4.4a, we can see that we that using LSD gives us a 1.7× increase in throughput when using hash based sharding or a TPC-C aware sharding. But now we can achieve a peak throughput of 5.2k transactions per second and 6ms average latency. In a low contention setting, in Figure 4.4b, the peak throughput rises to 6.7k for the TPC-C aware version, and 5.5k for the hash-based version. This is an improvement of 1.8× compared to the centralized single server configuration.



(a) 3 servers, high contention

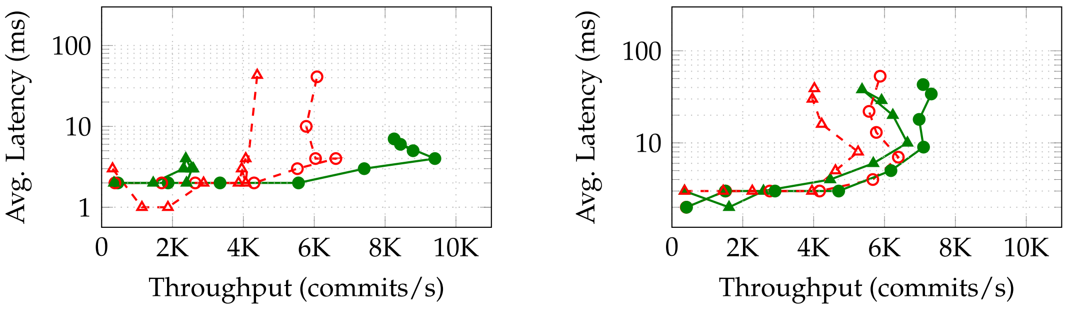
(b) 3 servers, low contention

Figure 4.4: TPC-C benchmark using 3 servers with a hashmap.

4.3.3.2 6 Servers

When using 6 servers we see the throughput increase for all versions, in Figure 4.5a, we see that the TPC-C aware version can achieve 9.4k transactions per second at 5ms average latency, but we start to see worst results for LSD with hash sharding. This could be attributed to the clients having to communicate with more servers, than the TPC-C aware sharding, increasing the number of communication rounds. This also increases the number of aborts and is the reason we see the terrible results of just 2.5k transactions per second, and is a case where using LSD leads to worst performance, as running just OCC gives us a peak of 4k transactions per second. We will discuss possible reasons later.

When we have a low contention setting, in Figure 4.5b, we can see results more inline with what we expected, here we see slightly better results for LSD comparing with the same test with 3 servers, with a peak of 8.9k transactions per second.



(a) 6 servers, high contention

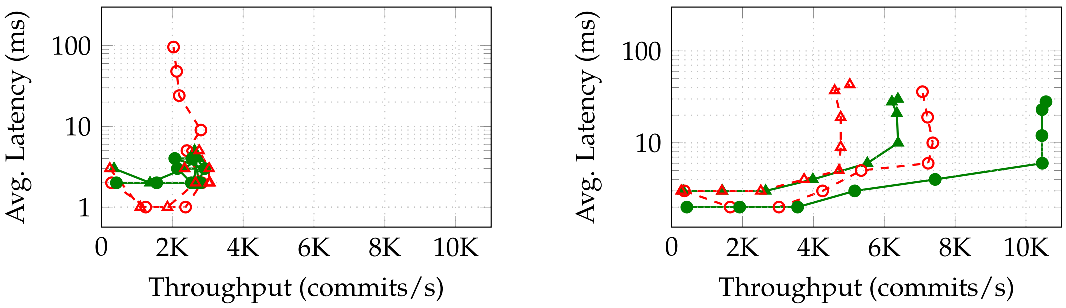
(b) 6 servers, low contention

Figure 4.5: TPC-C benchmark using 6 servers with a hashmap.

4.3.3.3 9 Servers

Finally, in Figure 4.6a, we are using 9 servers. Here we can see that the results go against our expectations and neither configuration was able to surpass 3k transactions per second, a 3.2× worst performance compared with using 6 servers.

Our reasoning for this, and also the previous bad result when using hash based sharding with 3 servers in a high contention setting, is that the load of running this number of servers proves to be too much for the node to handle and leads to the results we see. The same is not true when we are in a low contention setting, Figure 4.6b, here the load on the servers is less, and we can see much better results, when the system is operating correctly. We can achieve 10.5k transactions per second, at 5ms, using the TPC-C aware sharding and 7k transactions per second with the hash based sharding. With LSD still providing an increase in throughput compared to not using LSD, 1.5× with TPC-C aware sharding and 1.4× with the hash based sharding.



(a) 9 servers, high contention

(b) 9 servers, low contention

Figure 4.6: TPC-C benchmark using 9 servers with a hashmap.

4.3.3.4 Discussion

With these tests, we can see that LSD continues to scale horizontally, while still providing better throughput than not using LSD and reaching nearly 10k transactions per second. We also see that further testing of distributed servers, beyond 9, is required using a more powerful system as our tests seem to have reach a limit in the server's node performance.

4.3.4 Conclusion

In conclusion, in our testing LSD can provide better throughput in every situation, being able to increase it by 1.1× to 1.5×. It is most effective when we have a high contention setting, but can still pull ahead in a low contention setting. The combination of LSD with OCC proves to be the best choice for every scenario, as it was the same conclusion reached with the previous prototype.

In comparison with the previous prototype with a centralized database, using the best results obtained: 1k transactions per second, under high contention, with OCC and LSD and RocksDB in disk, our prototype provides:

- 2.8× increase (2.8k txn/s) in throughput when using persistent storage (RocksDB in disk).
- 3.7× increase (3.7k txn/s) in throughput when using non-persistent storage (hashmap).

And comparing the previous prototype with a distributed database, 1.9k transactions per second (3 servers), TPC-C aware sharding, under high contention, with OCC and LSD, and RocksDB in disk, our prototype provides:

- 2.3× increase (4k txn/s) in throughput when using a distributed system (3 servers) (high contention).
- 4.9k× increase (9.4k txn/s) in throughput when using a distributed system (6 servers) (high contention).

It is also important to mention that the peak values of throughput were reached with half to 10 times less latency, a result of faster network speed and faster I/O operations. With this in mind, we can say that our work was successful in our main objective of improving the performance (throughout) of the previous prototype. We were still unable to provide competitive results that can be used to compare LSD to other solutions that can provide throughput in the hundreds of transactions per second, but we are also aware that these solutions are highly optimized to run the TPC-C benchmark, and do so in a not so real-world environment, like forgoing the concept of a client-server database interaction model [41], which is not the case of LSD.

5

Conclusions

In this chapter, we give some final remarks about the work done, its results, limitations and where what future work can be done.

Our work involved the study of a new way of providing better throughput for key-value transactional systems that have to handle multiple concurrent transactions in a high contention setting.

We focussed our work in LSD, as its unique way of dealing with this problem, by providing a new interface for database transactions, making use of futures and delaying the need to observe the concrete state of data, allowing operations to be done on an abstract state, leading to a reduced conflict window between transactions, that have to access the same concrete data, resulting in more parallelism and a higher throughput database system.

We found, through our testing, that LSD can provide a significant improvement to an existing system in a simulated real word application. It is most effective in the present of high contention, where a traditional transactional system has difficulties maintaining its throughput, LSD can provide better throughput at lower latency.

We also found that with our improvements to the prototype, faster I/O operations, faster client and benchmark implementations, code optimizations and library updates and use of a faster testing environment, both in CPU power and network speed, that while the improvements provided by using LSD decreased compared to the previous prototype, where before we could see an improvement of 6.5× the throughput, and now we only see 1.9× improvement, we were able to increase the overall throughput of the system.

Where previously the peak throughput was 1k transactions per second in a centralized setting, we can now achieve 3.7k transactions per second at half the latency. And in a distributed setting, where we previously saw a peak of 1.9k transactions per second, we can now achieve 4.9k in a similar test and close to 10k transactions per second in the best-case scenario at 10 times lower latency.

After running our tests, we reached the conclusion that our work was limited by many factors.

First, we had to use an existing, old, implementation of LSD, limiting the number of changes and improvements possible. If starting from scratch, we could study and analyse different ways of dealing with client server communications, selecting one that can deliver lower latency. We could do the same for the low-level database, and use a more efficient persistent database. We could also introduce changes to how the algorithm is implemented and try to further optimize it.

Second, we were limited on the testing environment by the number and types of machines, as we needed to utilize the machines connected by the higher possible network speed, 10Gbps, we were limited to 5 nodes. Having more nodes means we could push our testing further, using more clients, and more importantly, having more powerful nodes to run the servers, allowing us to get a better idea of the scalability of having a bigger distributed system.

Third, we could look at other transactional database benchmarks, giving us more results to analyse and evaluate our work and to compare with other solutions.

Future research must be done to continue to push the bounds of what is possible with LSD. We see that further refinements to the algorithm can be an important approach to its optimization, as well as simplifying the use of LSD to make the adaptation of an existing application to use it easier.

An interesting approach would be to analyse and select a popular open-sourced Key-Value Store that supports transactions and adapt it to support the LSD algorithm and API. We could then select an existing application that already made use of this Key-Value Store, and modify it to use the newly added LSD API. This would then be followed by extensive testing and evaluation to derive the improvements of LSD in an actual real-world application.

Bibliography

[1] M. K. Aguilera et al. “Sinfonia: A New Paradigm for Building Scalable Distributed Systems”. In: 27.3 (2009-11). ISSN: 0734-2071. DOI: 10.1145/1629087.1629088 (cit. on pp. 2, 20).

[2] D. A. Beman Dawes. *Boost*. URL: https://www.boost.org/ (cit. on p. 31).

[3] H. Berenson et al. “A Critique of ANSI SQL Isolation Levels”. In: *SIGMOD Rec.* 24.2 (1995-05), pp. 1–10. ISSN: 0163-5808. DOI: 10.1145/568271.223785 (cit. on p. 11).

[4] P. A. Bernstein and N. Goodman. “Concurrency Control in Distributed Database Systems”. In: *ACM Comput. Surv.* 13.2 (1981-06), pp. 185–221. ISSN: 0360-0300. DOI: 10.1145/356842.356846 (cit. on pp. 5, 13, 14, 31).

[5] P. A. Bernstein and N. Goodman. “Multiversion Concurrency Control—Theory and Algorithms”. In: *ACM Transactions on Database Systems* 8.4 (1983-12), pp. 465–483. ISSN: 0362-5915. DOI: 10.1145/319996.319998 (cit. on p. 13).

[6] J. Cachopo and A. Rito-Silva. “Versioned Boxes as the Basis for Memory Transactions”. In: *Sci. Comput. Program.* 63.2 (2006-12), pp. 172–185. ISSN: 0167-6423. DOI: 10.1016/j.scico.2006.05.009 (cit. on pp. 1, 17).

[7] A. Cheung, S. Madden, and A. Solar-Lezama. “Sloth: Being Lazy Is a Virtue (When Issuing Database Queries)”. In: *ACM Trans. Database Syst.* 41.2 (2016-06). ISSN: 0362-5915. DOI: 10.1145/2894749 (cit. on pp. 1, 17).

[8] E. G. Coffman, M. Elphick, and A. Shoshani. “System Deadlocks”. In: *ACM Comput. Surv.* 3.2 (1971-06), pp. 67–78. ISSN: 0360-0300. DOI: 10.1145/356586.356588 (cit. on p. 9).

[9] S. Dasgupta. *It Began with Babbage: The Genesis of Computer Science*. USA: Oxford University Press, Inc., 2014. ISBN: 0199309418. DOI: 10.5555/2582031 (cit. on p. 15).

[10] N. L. Diegues and P. Romano. “Bumper: Sheltering Transactions from Conflicts”. In: SRDS ’13. USA: IEEE Computer Society, 2013, pp. 185–194. ISBN: 9780769551159. DOI: 10.1109/SRDS.2013.27 (cit. on pp. 1, 18).

[11] A. Dragojević et al. "No Compromises: Distributed Transactions with Consistency, Availability, and Performance". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: Association for Computing Machinery, 2015, pp. 54–70. ISBN: 9781450338349. DOI: 10.1145/2815400.2815425 (cit. on pp. 2, 19).

[12] K. P. Eswaran et al. "The Notions of Consistency and Predicate Locks in a Database System". In: *Commun. ACM* 19.11 (1976-11), pp. 624–633. ISSN: 0001-0782. DOI: 10.1145/360363.360369 (cit. on pp. 10, 12, 24).

[13] Facebook. *RocksDB*. URL: https://rocksdb.org/ (cit. on pp. 3, 26, 27).

[14] J. M. Faleiro, A. Thomson, and D. J. Abadi. "Lazy Evaluation of Transactions in Database Systems". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 15–26. ISBN: 9781450323765. DOI: 10.1145/2588555.2610529 (cit. on pp. 1, 17).

[15] J. Forcier. *Fabric*. URL: https://www.fabfile.org/ (cit. on pp. 34, 42).

[16] A. S. Foundation. *Apache Thrift*. URL: https://thrift.apache.org/ (cit. on pp. 26, 27, 31).

[17] D. P. Friedman and D. S. Wise. "The Impact of Applicative Programming On Multiprocessing". en. In: *International Conference on Parallel Processing*. 1976, pp. 263–272 (cit. on p. 16).

[18] J. Gray. "The Transaction Concept: Virtues and Limitations". In: VLDB '81. Cannes, France: VLDB Endowment, 1981, pp. 144–154. DOI: 10.5555/1286831.1286846 (cit. on p. 10).

[19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1558601902. DOI: 10.5555/573304 (cit. on p. 10).

[20] P. Henderson and J. H. Morris. "A Lazy Evaluator". In: *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*. POPL '76. Atlanta, Georgia: Association for Computing Machinery, 1976, pp. 95–103. ISBN: 9781450374774. DOI: 10.1145/800168.811543 (cit. on p. 16).

[21] M. Herlihy and J. E. B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures". In: *SIGARCH Comput. Archit. News* 21.2 (1993-05), pp. 289–300. ISSN: 0163-5964. DOI: 10.1145/173682.165164 (cit. on p. 10).

[22] M. P. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990-07), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972 (cit. on p. 6).

[23] Kitware. *CMake*. URL: https://cmake.org/ (cit. on p. 32).

[24] D. E. Knuth. "Computer Programming as an Art". In: *Commun. ACM* 17.12 (1974-12), pp. 667–673. ISSN: 0001-0782. DOI: 10.1145/361604.361612 (cit. on p. 15).

[25] V. Kumar. *Introduction to Parallel Computing*. 2nd. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201648652. DOI: 10.5555/600009 (cit. on p. 5).

[26] H. T. Kung and J. T. Robinson. "On Optimistic Methods for Concurrency Control". In: *ACM Trans. Database Syst.* 6.2 (1981-06), pp. 213–226. ISSN: 0362-5915. DOI: 10.1145/319566.319567 (cit. on pp. 13, 24).

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

[28] D. A. Menascé and T. Nakanishi. "Optimistic versus pessimistic concurrency control mechanisms in database management systems". In: *Information Systems* 7.1 (1982), pp. 13–27. ISSN: 0306-4379. DOI: 10.1016/0306-4379(82)90003-5 (cit. on p. 12).

[29] S. Mu et al. "Extracting More Concurrency from Distributed Transactions". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 479–494. ISBN: 9781931971164. DOI: 10.5555/2685048.2685086 (cit. on pp. 1, 19).

[30] N. Narula et al. "Phase Reconciliation for Contended In-Memory Transactions". In: OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 511–524. ISBN: 9781931971164. DOI: 10.5555/2685048.2685088 (cit. on pp. 1, 19).

[31] F. Raab. "TPC-C - The Standard Benchmark for Online transaction Processing (OLTP)". In: *The Benchmark Handbook*. 1993 (cit. on p. 2).

[32] M. Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Publishing Company, Incorporated, 2012. ISBN: 3642320260. DOI: 10.5555/2431405 (cit. on pp. 6, 7).

[33] R. Schaller. "Moore's law: past, present and future". In: *IEEE Spectrum* 34.6 (1997), pp. 52–59. DOI: 10.1109/6.591665 (cit. on p. 1).

[34] ScyllaDB. *Seastar*. URL: http://seastar.io/ (cit. on p. 27).

[35] V. Slavík. *Bakefile*. URL: https://bakefile.org/ (cit. on p. 32).

[36] H. Sutter et al. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210 (cit. on p. 1).

[37] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. 2nd. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2016. ISBN: 153028175X. DOI: 10.5555/3019390 (cit. on p. 6).

[38] G. Tene. *HDR Histogram*. URL: http://www.hdrhistogram.org/ (cit. on p. 34).

[39] A. Thomasian. "Concurrency Control: Methods, Performance, and Analysis". In: *ACM Comput. Surv.* 30.1 (1998-03), pp. 70–119. ISSN: 0360-0300. DOI: 10.1145/274440.274443 (cit. on p. 12).

[40] TPCC. *TPC-C*. URL: https://www.tpc.org/tpcc/ (cit. on pp. 3, 33, 44).

[41] S. Tu et al. "Speedy Transactions in Multicore In-Memory Databases". In: SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 18–32. ISBN: 9781450323888. DOI: 10.1145/2517349.2522713 (cit. on pp. 2, 3, 18, 51).

[42] T. M. Vale et al. "Lazy State Determination: More concurrency for contending linearizable transactions". In: *arXiv preprint arXiv:2007.09733* (2020) (cit. on pp. iv, v, 1–3, 20, 26, 30, 35).

[43] T. M. Vale. "Executing requests concurrently in state machine replication". PhD thesis. NOVA School of Science and Technology | FCT, 2019. URL: http://hdl.handle.net/10362/71218 (cit. on pp. iv, v, 1–3, 20, 26, 30, 35).

[44] 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 pp. 1, 19).

[45] C. Xie et al. "Salt: Combining ACID and BASE in a Distributed Database". In: OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 495–509. ISBN: 9781931971164. DOI: 10.5555/2685048.2685087 (cit. on pp. 1, 19).

[46] X. Yu et al. "Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores". In: *Proc. VLDB Endow.* 8.3 (2014-11), pp. 209–220. ISSN: 2150-8097. DOI: 10.14778/2735508.2735511 (cit. on p. 1).

[47] 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 pp. 2, 18).

2022 Improving Key-Value Database Scalability with Lazy State Determination David Carpinteiro

NOVA SCHOOL OF SCIENCE & TECHNOLOGY