NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

THALES VINÍCIUS ALVES PARREIRA
Bachelor of Computer Science and Engineering

EMPOWERING A RELATIONAL DATABASE WITH LSD: LAZY STATE DETERMINATION

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2022

DEPARTMENT OF
COMPUTER SCIENCE

EMPOWERING A RELATIONAL DATABASE WITH LSD: LAZY STATE DETERMINATION

THALES VINÍCIUS ALVES PARREIRA

Bachelor of Computer Science and Engineering

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

Examination Committee

Chair: João Costa Seco
Associate Professor, FCT | NOVA University Lisbon

Rapporteur: Nuno Antunes
Assistant Professor, FCT | University of Coimbra

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

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2022

Empowering a Relational Database with LSD: Lazy State Determination

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.17) [9].

To my wife, mom and dad.

Acknowledgements

After completing a late bachelor degree, I have decided to continue with a masters in Computer Science. This dissertation marks the end of this very long journey, that had its ups, downs, failures, successes, starts, and stops. There were many times I imaged I would stop studying and focus on my career, yet every time I continued I felt there was a lot more to learn. Although the journey was not easy, I have grown fond of it, and I am very grateful for the opportunity of having gone through it. It is because of this that I express my gratitude to, not only those who have helped me in this dissertation, but also in my bachelor degree.

First, I express my gratitude towards my advisor, João Lourenço. Thank you for bringing me closer to the academic world, for always showing confidence in my work, by inspiring and motivating me to keep on working hard. Above all, thank you for always being available, striving for the best and for doing what I always felt was more than anyone would expect.

To our special institution, NOVA School of Science and Technology, thank you for the knowledge you have provided and all the personal growth you allowed me to have.

To my dear wife, a special thank you, for the endless support, motivation, and care along this journey; with you this would have not been the same.

To my family, specially my mom and dad, who have given me this opportunity, thank you for always supporting me, for all you have done, and for all you continue to do.

Last but not least, thank you, Tomás Pessanha, for accompanying along this journey, for all the car sharing and for all the good and bad moments we have shared.

This work was developed within the framework of the HiPSTr project (PTDC/CCI-COM/32456/2017 & LISBOA-01-0145-FEDER-032456).

> *“Do, or do not. There is not try.”*
>
> **— Yoda**

Abstract

Computer systems are a part of today's most common activities and, more often than not, involve some type of interaction with a database. In this scheme, databases play a big role, where even small operational delays could cost millions to big tech companies. It is then, of utmost importance that such systems are responsive and adapt automatically to different types of workload.

To this date, Relational Database Management System remain the most popular database type, which allows the executing of concurrent transactions with Atomicity, Consistency, Isolation and Durability guarantees. Enforcing such properties requires strict control over the execution of transactions. However, maintaining such properties and controlling the transactions' concurrency may hamper performance of the system, being this specially the case when database contention is high.

Motivated by such behavior, we propose the lazy evaluation of database SQL queries — using Futures/Promises and Java Database Connectivity (JDBC) — by empowering a relational database with Lazy State Determination (LSD). This novel Application Programming Interface (API) allows delaying operations to the *commit* time, which in the end reduces the transaction window where conflicts may occur.

We observed that, by introducing our implementation of a JDBC-LSD driver, in high contention scenarios the throughput increased by 50% and latency reduced by 40%.

Keywords: Concurrency Control, Relational Databases, Lazy State Determination, Java Database Connectivity, Transactions

Resumo

Os sistemas informáticos são parte das atividades mais comuns na atualidade e, na maioria das vezes, envolvem algum tipo de interação com uma base de dados. Neste cenário, as bases de dados têm um grande papel, sendo que pequenos atrasos operacionais podem custar milhões às grandes empresas tecnológicas.

Até os dias de hoje, os Sistemas de Gestão de Bases de Dados Relacionais continuam a ser o tipo de bases de dados mais popular, permitindo a execução concorrente de transações garantindo as propriedades de Atomicidade, Consistência, Isolamento e Durabilidade. A aplicação de tais propriedades requer um controlo rigoroso sobre a execução de transações. No entanto, manter tais propriedades e controlar a concorrência das transacções pode diminuir o desempenho do sistema, sendo especialmente o caso em bases onde a contenção é elevada.

Assim, propomos o atraso na execução de queries SQL na base de dados através da introdução do protocolo de controlo de concorrência Lazy State Determination (LSD), com a utilização de Futuros/Promessas e Java Database Connectivity (JDBC). Esta nova Interface de Programação de Aplicações (API) permite adiar as operações para o momento do *commit*, o que acaba por reduzir a janela da transação onde conflitos podem ocorrer.

Observamos que, ao utilizar LSD em um cliente JDBC, nós conseguimos aumentar a taxa de execução de transações em 50% e reduzir a latência em 40% num ambiente de contenção elevada.

Palavras-chave: Controlo da Concorrência, Bases de Dados Relacionais, *Lazy State Determination*, *Java Database Connectivity*, Transacções

Contents

Appendices

List of Figures

List of Tables

Acronyms

2PL	Two Phase Lock *(pp. 9, 12, 17, 19, 20)*
ACID	Atomicidade, Consistência, Isolamento e Durabilidade *(p. vii)*
ACID	Atomicity, Consistency, Isolation and Durability *(pp. vi, 2, 4, 5, 14–16)*
API	Interface de Programação de Aplicações *(p. vii)*
API	Application Programming Interface *(pp. vi, xi, xii, 2, 3, 16–26, 28, 31, 32, 35, 36, 39, 50, 51)*
ATM	Automated Teller Machine *(p. 55)*
BASE	Basically Available, Soft-state, Eventually-consistent *(pp. 14, 15)*
CC	Concurrency Control *(pp. 1, 2, 7, 9, 11–17, 19, 51)*
DBMS	Database Management System *(pp. 2, 5–7, 9, 58, 60)*
GC	Garbage Collector *(p. 12)*
JDBC	Java Database Connectivity *(pp. vi, vii, xi, 3, 20, 22–26, 28, 31, 32, 34–36, 39–51)*
JVM	Java virtual machine *(p. 32)*
LSD	Lazy State Determination *(pp. vi, vii, xi, xii, 2–4, 16–21, 23, 24, 28, 31, 32, 35, 36, 39–51)*
MCC	Modular Concurrency Control *(p. 15)*
MV2PL	Multi-Version 2 Phase Lock *(p. 12)*
MVCC	Multi-Version Concurrency Control *(pp. 12, 13)*
MVOCC	Multi-Version Optimistic Concurrency Control *(p. 12)*
OCC	Optimistic Concurrency Control *(pp. 10–13, 17, 19)*

ODBC	Open Database Connectivity *(p. 22)*
OLTP	Online Transaction Processing *(p. 55)*
RDBMS	Relational Database Management System *(pp. vi, 2–4, 20, 21, 31, 42, 50, 58)*
SGBDR	Sistemas de Gestão de Bases de Dados Relacionais *(p. vii)*
SQL	Structured Query Language *(pp. xi, 2, 19, 20, 23–25, 28, 31–34)*
TID	Transaction Id *(p. 11)*
TPC	Transaction Processing Performance Council *(p. 55)*
TPC-C	TPC Benchmark C *(pp. 2, 11–13, 16, 42, 43, 45, 55–58, 60)*
YCSB	Yahoo! Cloud Serving Benchmark *(pp. xii, 11–13, 55, 58–60)*

1

Introduction

1.1 Context and Motivation

Long has gone the era where single-threaded was the norm for computer systems. The decades of exponential performance growth for single-threaded applications are gone, and multithreaded architectures are ubiquitous in the current computer systems nowadays. With this in mind, techniques such as parallelism and concurrency have grown in importance in order to fully exploit today's multi-threading hardware capabilities.

When we think about today's world of millions of people interacting on a daily basis with multiple interconnected systems, where delays in milliseconds could cost millions to big tech companies [1], it is of great importance that systems are adapted to be responsive to many types of workload.

One thing that should be common in most of these systems is that, more often than not, they involve some type of interaction with a database. For example, accessing an online bank account, viewing a feed on Instagram [1], reserving an airline ticket, and even the non-trivial cases such as browsing your web browser bookmarks might involve some sort of database. These databases can range from memory fitting size to having to be deployed in multiple data centers scattered around the world.

From the above, it is obvious that efficient processing of these databases is needed and, as a consequence, fast execution of transactions is of great importance to any modern system. Processing one transaction at a time won't cut it, and employing parallelism is necessary to speed up execution.

If a database's single purpose would be for reading data, parallel execution would not be an issue, as there was no way transactions would interfere with one another. However, for most systems this is not the case, as the database serves both read-only and read-write transactions. In this setting, employing parallelism in an uncontrolled manner would cause data consistency to be lost. This is where Concurrency Control (CC) techniques come into play, as they are responsible for ensuring the correct interactions of users on a multi-user database system, that executes multiple read-write transactions concurrently. Different

[1] https://www.instagram.com/

CC techniques have different kinds of impact in concurrent execution of transactions and can generally be divided in two main classes, optimistic and pessimistic solutions.

The context of this dissertation relates with allowing a greater concurrent execution of transactions. The following chapters will detail even further the followed approach for achieving that.

1.2 Problem and Goals

Reasoning with database transactional systems is simplified by the Atomicity, Consistency, Isolation and Durability (ACID) abstraction, where transactions appear to execute atomically without interference, despite being executed concurrently. Maintaining a system with such requirements requires strict control over the transactions execution and ways of handling or preventing conflicts. CC techniques are what enable Database Management System (DBMS) to maintain data consistency and isolation in concurrent environments. Many proposals have been made to improve these techniques in high contention environments, however, this dissertation will focus on reducing the chance of data conflicts, hence reducing data contention.

This work will focus on Lazy State Determination (LSD) [23], which proposes to reduce contention by decreasing the time window in which a transaction might conflict with one another, hence, increasing concurrency between transactions. Vale [23] notes that a lack of semantics in transactions leads to a conservative view of what is a transaction conflict. The example of two transactions updating the number of items in an inventory is given, where both transactions conflict due to both writing to the same tuple. However, provided that aggregate effects are preserved, the semantic of the transactions do not conflict. It was with this in mind that Lazy State Determination (LSD) was proposed.

The LSD Application Programming Interface (API) has already been tested successfully in NoSQL context. However, it remains to be proven successful in a Relational Database Management System (RDBMS) context. An initial prototype for RDBMS has already been done by Subtil [18], but its performance failed to deliver similar results of NoSQL LSD implementation.

1.3 Approach and Contributions

This dissertation plans to implement LSD for Structured Query Language (SQL) databases, following the work done by Vale [23], where in that case LSD was tested in a NoSQL database. To achieve this, we propose an approach to Vale [23] that relies solely on the client side, allowing LSD to be used more easily in different databases. The solution we came up was tested using a standard RDBMS test for databases, the TPC Benchmark C (TPC-C) [19] benchmark, and achieved considerable improvements on high contention environments, with close to no impact in low contention scenarios.

With this, our contributions are as follows:

- A methodology for defining futures in an object-oriented setting and how to use them to delay transaction execution;
- A new Java Database Connectivity (JDBC)-LSD API working as an extension of the JDBC API;
- A JDBC-LSD driver compatible with all RDBMS databases that have a JDBC driver;
- A detailed approach on how to implement LSD in the client;
- A performance evaluation of having delayed operations solely in client.

1.4 Outline

The remainder of the document is organized as follows:

Chapter 2 — This chapter explores the basics of transactions and also visits relevant related work to this dissertation. It also describes the approach followed by this dissertation work, LSD;

Chapter 3 — This chapter gives an overview of the JDBC API;

Chapter 4 — This chapter looks into how to define a future and the properties it should have;

Chapter 5 — This chapter defines the JDBC-LSD API by looking at its interfaces, implementation and giving an example;

Chapter 6 — This chapter focuses on presenting the results of the new JDBC-LSD API when compared to the standard JDBC;

Chapter 7 — This is the final chapter which makes a summary of the work presented in this document and gives guidelines to future work that will assist the solution presented in this document to achieve even better results.

2

Background and Related Work

This Chapter introduces some base concepts for transactions, and presents relevant related work on concurrency in databases.

Section 2.1 starts by defining a transaction and its desired properties. Subsection 2.2.1 studies current literature on concurrency inside databases. Finally, Section 2.3 studies Lazy State Determination (LSD) [23, 22], and Subtil [18] approach on LSD is contrasted with the studied literature.

2.1 Transactions

More often than not, database operations appear as a single unit of work from the point of view of the database user. For instance, a bank transfer from a savings account to a debit account, to a database user appears as a single unit of work, while internally it should involve more than one operation. This is what we call transactions and form the basis for interactions with Relational Database Management System (RDBMS) systems.

2.1.1 Transaction Properties

In order to ensure data integrity, RDBMS require four important properties, know as Atomicity, Consistency, Isolation and Durability (ACID) [17]:

Atomicity — a transaction is a single unit of work, it should be either executed in its entirety or not executed at all;

Consistency — a transaction should preserve consistency, meaning it should take the database from a consistent state to another consistent state after its execution;

Isolation — a transaction must execute independently of other transactions, meaning that concurrent changes occurring in one transaction should not be visible to others until they are committed; and

Consistency — all transactions changes must be permanent in the database and must not be lost in case of failure.

As described previously, isolation is one of the foundations of transaction execution, it also often the most relaxed propriety of Atomicity, Consistency, Isolation and Durability (ACID), as maintaining the highest isolation level typically results in loss of concurrency and lower throughput. In most commercial Database Management System (DBMS) it is possible to fine tune isolation level, providing a balance between performance and consistency.

The ANSI/SQL 92 refers to isolation with to three different phenomena in mind [2]:

Dirty read — a transaction is allowed to read uncommitted data from another executing transaction;

Non-repeatable read — a transaction will read only committed data, but a row retrieved more than once might differ between reads; and

Phantom read — these phenomena occurs when in the same transaction two identical queries may return two sets of rows where one is the subset of another.

The ANSI/SQL-92 (and Berenson et al. [2]) defines four different isolation levels:

Serializable — the highest level which enables the output of concurrent transactions to appear as being executed serially;

Repeatable read — transactions are guaranteed to only read committed data, and see the same result if the same read operation is executed repeatedly;

Read committed — transactions are guaranteed to only see changes that have been committed; and

Read uncommitted — changes made by a concurrent transaction are visible by other transactions (this is the lowest isolation level).

Table 2.1 relates each anomaly with each isolation level. Read uncommitted is the weakest isolation level and consequently allows dirty reads, non-repeatable reads and phantom writes. An improvement to read uncommitted is made with read committed by ensuring that dirty reads do not occur, however, other anomalies are still present. Non-repeatable reads and dirty reads are not present in repeatable read isolation level, however, phantom reads may still occur. Finally, serializable, the strongest of isolation levels, ensures that no anomalies occur.

2.1.2 Serializability

It is generally accepted that serializability is the strongest property and defines a standard notion for correctness in DBMS [11]. Essentially, the execution of a group of transactions is said to be serializable if the concurrent, and possibly interleaved, execution of the

Table 2.1: Anomalies per isolation level, as described in by Berenson et al. [2].

Isolation level	Dirty read	Non-repeatable read	Phantom read
Repeatable read	Possible	Possible	Possible
Read committed	Not Possible	Possible	Possible
Repeatable read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

transactions has the same effect, and produces that same outcome as some serial execution of the same transactions.

It is important to notice that no order for serial execution is specified, but rather that it should be equivalent to some serial order. This simple notion gives DBMS added flexibility when scheduling operations and may result in increased responsiveness of the system.

As an example, let's imagine two concurrent users that want to increment a counter, but need to first check its value. If both users retrieve the counter's value before either one of them updates it, the resulting sequence of execution, or schedule, is not serializable. This is because the serial execution of both transactions would have resulted in smaller than expected total increment. However, if any of the users updates the value before the other reads it, then the resulting schedule would have been serializable.

In the following sections, we go into more detail on what is a serializable schedule.

2.1.2.1 Serializable Schedules

A schedule, or history, is an abstract model that describes the execution ordering of read and write operations, for a set of transactions in a DBMS. More specifically, the literature defines two types of schedules, serial and non-serial schedules [17].

A *serial* schedule is an ordering where each transaction is executed after another. There is no interleaving of operations between transactions and consequently the performance of the execution of such schedule is slower.

In *non-serial* schedules, transaction operations are allowed to be executed in an interleaved manner. The overall ordering of operations within a transaction is always maintained, however, other operations (from other transactions) may execute in between a transaction. This gives the DBMS better opportunities for concurrent execution and improved transaction throughput.

Table 2.2 shows an example of three schedules. Schedule a represents a serial schedule, and in this case, transaction $T1$ is executed after transaction $T2$. Executing all operations of $T2$ before $T1$ would still generate a *serial* schedule. Schedule b represents an equivalent *non-serial* serializable execution of schedule a. It is equivalent because the outcome of the operations is the same. This interleaving of operations in *non-serial* schedules however, may result in an inconsistent state of the system, hence, it is important to find equivalent schedules that are serializable. Table 2.2 has an example of a *non-serial* non-serializable execution in schedule c. In this case the interleaving of operations is not equivalent to the *serial* execution present in schedule a and will leave the database in an inconsistent state.

The concept of *equivalence* is introduced in order to provide a set of syntactic rules for transforming a schedule to another that provides the same resulting effects. This is important because it enables us to validate that a given is schedule is equivalent to serial schedules, and consequently, will leave the database in a consistent state.

An important form of equivalence is *conflict equivalence*. A schedule is said to be *conflict equivalent* to a serial execution if the operations contained in each one are the same, and all pair of conflicting operations are ordered in the same way. Verifying that a schedule is *conflict serializable* is done by analyzing two transactions to check if they access the same item and if at least one of the transactions is doing a write operation. The outcome of such analysis may then be modeled to a *precedence graph* and checked for cycles [17].

In practice, *DBMS* do not test for serializability of *schedules* and instead relies on protocols, such as the Concurrency Control (CC), for ensuring serializability.

2.2 Concurrency in Databases

Given that this dissertation focus on improving transactions' execution by leveraging concurrent methods, it is important to revisit traditional methods and also study the state of the art in order to reap the right benefits for this work.

2.2.1 Concurrency Control

Concurrency Control (CC) is the mechanism used to manage simultaneous operations in a DBMS. It is what enables multiple transactions to execute concurrently without one interfering with another, while ensuring that the system goes from a consistent state to another. Along the years, many techniques have been studied and developed. Berenson et al. [3] describes three generic approaches for designing CC algorithms:

Wait — conflicting actions of one transaction must wait until the other transaction is completed;

Timestamp — transaction execution order is selected based on timestamps, where each transaction is assigned a timestamp and conflicting actions are solved by timestamp order; and

Rollback — if two transactions conflict, one is aborted and rolled back.

These three generic approaches form the basis for most algorithms and are employed to today's most common algorithms for CC.

The usage of such three generic approaches tend to lead to a categorization for the types of CC available, namely, pessimistic and optimistic [10], in which the first one preemptively locks database items while the later involve validating concurrent transactions at commit

Table 2.2: Example of a *serial*, a *non-serial* serializable and a *non-serial* non-serializable schedule. Schedule a represents a *serial* schedule. Operations of *T2* may come before T1 and the schedule would still be *serial*. Schedule b interleaves the operations of *READ* and *WRITE* of *A* and *B* for both transactions and forms a *non-serial* serializable execution of Schedule a. Schedule c represents a *non-serial* non-serializable schedule of a. It is non-serializable because its operations would leave the database in an inconsistent state.

T1	*T2*
READ(A)	
A = A - 50	
WRITE(A)	
READ(B)	
B = B + 50	
WRITE(B)	
COMMIT	
	READ(A)
	*temp = A * 0.1*
	A = A - temp
	WRITE(A)
	READ(B)
	B = B + temp
	WRITE(B)
	COMMIT

(a) *Serial* schedule.

T1	*T2*
READ(A)	
A = A - 50	
WRITE(A)	
	READ(A)
	*temp = A * 0.1*
	A = A - temp
	WRITE(A)
READ(B)	
B = B + 50	
WRITE(B)	
COMMIT	
	READ(B)
	B = B + temp
	WRITE(B)
	COMMIT

(b) *Non-serial* serializable schedule.

T1	*T2*
READ(A)	
A = A - 50	
	READ(A)
	*temp = A * 0.1*
	A = A - temp
	WRITE(A)
WRITE(A)	
READ(B)	
B = B + 50	
	READ(B)
	B = B + temp
	WRITE(B)
	COMMIT
WRITE(B)	
COMMIT	

(c) *Non-serial* non-serializable schedule.

time. In the following subsections, we detail some common pessimistic and optimistic approaches, while also presenting recent work that use different techniques.

2.2.1.1 Pessimistic Concurrency Control

A simple yet thoughtful solution for conflicting transactions is to make transactions wait for others to finish working with conflicting database items. To implement this, the DBMS can provide locks on database items. Transactions will get a lock when working with the database and keep it as long as it is used, and then release it after its usage is completed. If the lock was already acquired by the other transaction, the transaction will wait for the lock to become available. An optimization can be done to the locking by having two types of locks:

Read-lock (or shared lock) — must be acquired when reading a database item. The item is locked in shared mode, which allows other transactions that want to read the same item to also acquire the lock; and

Write-lock (or exclusive lock) — most be acquired when writing a database item. The item is locked in exclusive mode and blocks any other transaction trying to read or write to the same item.

This concept is part of the CC algorithm Two Phase Lock (2PL) [5], and is further expanded by ensuring that locks are acquired and released in two specific phases:

Growing Phase — locks are acquired, but no lock can be released; and

Shrinking Phase — locks are released, and no lock can be acquired.

Figure 2.1 demonstrates 2PL transaction phases. The lock point in the figure represents the point from which no more locks can be acquired.

When all transaction follows the 2PL locking approach, the system is serializable [5], and this happens because after a transaction completes the *Growing Phase* all other conflicting transactions will have to wait for it to finish.

Methods that rely heavily on locks may be susceptible to deadlocks, and this is the case of 2PL. This occurs when two transactions wait for locking resources that each other own. Three different techniques might be used for addressing this issue: prevention, detection and avoidance [6].

Deadlock prevention works by having transaction locks tested beforehand. This consists in checking that the lock the transaction is attempting to acquire will or not cause a deadlock in the system. If it leads to a deadlock, the transaction is aborted and all the pending operations are rolled back for the transaction to restart its operation. Otherwise, the transaction is allowed to wait for the lock.

Deadlock detection uses *wait-for-graph* for detecting transaction deadlocks. Transactions that are detected to be deadlocked are aborted and restarted. Transactions are set

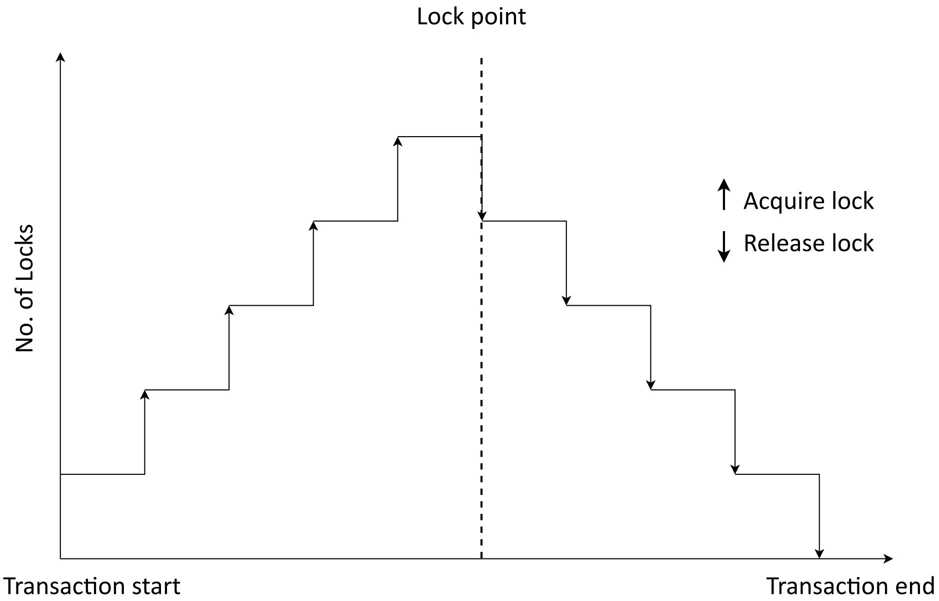


Figure 2.1: Two phase locking phases. Source [3].

as graph vertexes and graph edges are defined based on the resources a transaction is waiting from another transaction. Deadlocks are then detected by looking for cycles in the *wait-for-graph*.

With respects to deadlock avoidance, this is achieved by having transactions declare what resources they are going to acquire. The standard technique for this is to have transactions to linearly order (or any other order) the resources and acquire them in that order only.

2.2.1.2 Optimistic Concurrency Control

Another approach that differs from the locking methods presented in Subsubsection 2.2.1.1 is to assume that transactions don't conflict that often, also known as an optimistic solution. This is the approach followed by [7] in the Optimistic Concurrency Control (OCC) mechanism. In this approach transactions are free to read and write, but the write operations remain pending and, before committing, transactions have to validate that their operations do not conflict with the operations from other concurrent transactions. For this, the algorithm defines three important phases:

Read phase — transactions read database items and store local copies of them in a read set;

Write phase — optional phase where transactions write locally to database items in a private write set; and

Validation phase — atomically verifies that the records in the read set and write set were not modified by other concurrent transactions. If no conflict is found, the pending updates in the write set are applied and the transaction commits. If a conflict is found, the read and write set are discarded, and the transaction is aborted and restarted. A typical approach for deciding on the transaction to be aborted is to compare the transactions timestamps, which are assigned at the start of each transaction.

Tu et al. [21] designed Silo, an in-memory database that uses an adapted version of OCC for high performance workloads. The work centers around efficient memory usage while ensuring great scalability for multicore systems in shared-memory databases, i.e., where the entire database is accessible to all database workers. Silo's CC centers around the usage of a unique Transaction Id (TID) for each transaction. A TID is a numeric word that is used to identify transactions, record versions, serve as locks and detect conflicts. Every record of Silo's database is associated with the TID of the transaction that most recently modified it.

Similar to OCC, Silo's protocol works in three phases and uses local read and write sets for validating transactions and storing local changes. The three phases are described as follows:

Phase 1 — the database worker locks modified records and after that takes a snapshot a global epoch number in order to ensure serializability between transactions;

Phase 2 — read records are examined to check for changes during the transaction execution. This is done with the records TID. If no changes is found, a TID is generated for the running transaction; and

Phase 3 — writes new record values and updates their TID with phase 2 computed TID. Releases acquired locks.

Another important feature of Silo's CC is the usage of epochs, which enables the protocol to enforce serializability. Epochs are also present within the numeric word TID and are extracted from a global epoch number that governs the system and enables the system to ensure serializability. More specifically, serializability is ensured by the fact that:

- All written records are locked before validating TIDs;
- Locked records are treated as dirty and transactions are aborted upon encountering them; and
- Snapshot of global epoch guarantees that the most recent version of requested items is available to the running transaction

The authors of Silo validated their system against TPC Benchmark C (TPC-C) and Yahoo! Cloud Serving Benchmark (YCSB). Results show close to linear scalability when increasing the database worker thread count.

TicToc [27], also based in OCC, builds upon research on poor scalability of current timestamp generation methods and proposes a novel data-driven timestamp management protocol. In this protocol, data items are assigned a read and write timestamp, which are then used by the protocol to lazily compute a valid timestamp for each transaction.

The protocol distributes assignment of timestamps, leading to a better scalability of the generation algorithm. Additionally, this also removes the need of having a global centralized mechanism for generating timestamps for transactions, which leads to improved performance.

Another key feature of TicToc, is the fact that it lazily assigns timestamps. This is relevant because it gives a greater flexibility when compared with statical timestamp assignment which then forces a fixed order of execution of transactions. Schedules that would potentially not be allowed by general OCC methods can potentially execute under TicToc. An example of this would be:

1. A read(x)
2. B write(x)
3. B commits
4. A write(y)

This interleaving of operations would be allowed in TicToc, but not in OCC. This is because Transaction B has modified x and hence A would fail in the validation step.

TicToc was validated against YCSB and TPC-C benchmarks and was reported to achieve great performance, even when compared with state-of-the-art OCC algorithms such as Silo [21].

2.2.1.3 Multi-Version Concurrency Control

Another important CC mechanism is the Multi-Version Concurrency Control (MVCC) and is widely used today in commercial databases. It's implementation varies from vendor to vendor (Oracle, Postgres, MS SQL Server, etc.) [24] but they revolve around the idea of having multiple versions of each database tuple. Wu et al. [24] describes some flavors in detail, but the most widely used commercially involve modifications to the original OCC and 2PL in the form of Multi-Version Optimistic Concurrency Control (MVOCC) and Multi-Version 2 Phase Lock (MV2PL).

Independently of the implementation, some key aspects of the MVCC are the usage of unique monotonically incrementing timestamps for determining the version of tuples each transaction uses, the increase in storage due to added need to store multiple versions of the same tuple, and the need for having Garbage Collector (GC) for pruning versions that are no longer needed.

Cicada [8], by Lim, Kaminsky, and Andersen, is a single node multicore transactional in-memory database that leverages MVCC and OCC. It employs several optimizations to both CC algorithms in order to achieve top of class performance.

In Cicada, transaction timestamps are assigned at the beginning of the transaction and are used to decide which records version shall be used. These are obtained via loosely synchronized software clocks, a multi-clock design thought to alleviate issues with original MVCC timestamp generation. Each thread is responsible for generating its thread transaction timestamp, and special optimizations are made so that clocks between threads are close to synchronized.

Since this is a MVCC based protocol, Cicada stores multiple versions of records using the best effort inlining, a method for reducing allocation of new version records. Additionally, and for each record version, read and write timestamps are stored for later transaction validation.

Optimizations are also done to the validation step of the protocol.

1. Sort of write set based on contention in order to reduce footprint of following steps in case of abort;

2. Early consistency checks validation after sorting write set for contention; and

3. Incremental version search in order to reduce cost when validating versions timestamps against the read set.

Contention is also a target of study by Lim, Kaminsky, and Andersen. A mechanism based of *backoff* is proposed in order to alleviate contention caused by aborts of OCC transactions. This mechanism forces aborted transactions to sleep before restarting its work. Adjustments over the *backoff* value are made overtime based on system contention and throughput.

Cicada evaluates its performance with TPC-C and YCSB benchmarks. Results show that Cicada outperforms or matches the performance of state-of-the-art in-memory databases such as Silo or TicToc in various workloads.

2.2.1.4 Other Approaches

In this section, we study different CC approaches that move away to from the traditional methods of pessimistic and optimistic CC by approaching the problem in different ways in order to increase throughput.

In order to improve performance in high contention environments, Prasaad, Cheung, and Suciu [12] propose STRIFE, a new CC algorithm that leverages insights on the transaction accessed data in order to optimize its scheduling and execution. This is done by dividing transactions into batches of clusters and a set of residuals. Optimization here comes from the fact that inter-cluster wise, transactions are allowed to execute in a conflict-free manner without the usage of any CC algorithm. Transactions that do not fit in said clusters, due to conflicting with other transactions, are grouped into what is called a residual set and executed with traditional CC solutions. Prasaad, Cheung, and Suciu describes three-phase for the algorithm:

Analysis phase — batches incoming transactions and models them into an undirected bipartite graph of data accesses. This allows STRIFE to know which transactions conflict with one another and to decide on the clustering to select. Transactions that interact with data present in multiple clusters are grouped into the residuals set.

Conflict-free phase — schedules clusters execution on multiple execution threads. Each execution thread gets a cluster and executes it. Transactions that belong to the same cluster are executed serially. Isolation is guaranteed since clusters are conflict-free.

Residual phase — transactions that do not fall under the conflict-free clusters are executed on this phase. Its execution is concurrent on all available execution threads, but are managed by traditional CC mechanisms.

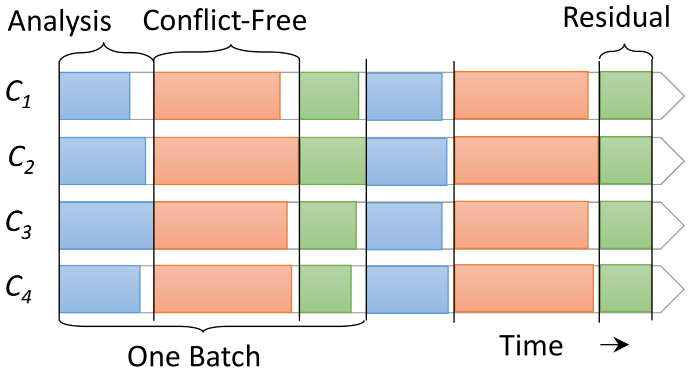


Figure 2.2: Execution scheme of STRIFE in 4 core machine. Source [12].

The above phases are grouped into batches and delivered for execution to each core in the machine. Figure 2.2 demonstrates the execution of such batches in a 4 core machine. First there is an analysis phase in each core, it executes the conflict-free phase as already described and finally executes the residual phase. The process is then repeated.

Differently from what has been done by Prasaad, Cheung, and Suciu [12], Xie et al. [26] approach the problem from a different perspective motivated by the Pareto principle. They note that for many applications, there are few transactions that test the performance limits of ACID. In parallel, they refer to the widely adopted in NoSQL databases Basically Available, Soft-state, Eventually-consistent (BASE) [13], which by loosening requirements in terms of consistency and isolation allows higher a throughput. It is with this in mind that they propose the concept of BASE transactions. An abstraction that loosens the requirements of isolation and consistency in order to achieve greater performance.

The BASE transaction abstraction works by enabling ACID transactions to be decomposed into multiple nested transactions, called alkaline subtractions, that run at BASE level. ACID transactions can still be present, and its guarantees are still present even when interacting with other ACID or BASE transactions. For this to work, the *Salt Isolation* property is established. This property makes use of three different types of locks to achieve isolation:

Saline locks — ensures isolation between ACID and BASE transactions, while at the same time allows for increased concurrency by allowing transactions to view intermediate states;

Alkaline locks — ensures isolation between alkaline subtractions and ACID transactions; and

ACID locks — strongest and traditional ACID write and read locks.

Referring back to the mentioned Pareto's principle, they show that not all transactions need to be converted to BASE transactions. This is specially highlighted by their results, in which with the conversion of only two ACID transactions to BASE transactions, they were able to achieve a great improvement over throughput.

Another interesting approach that shares some authors of Xie et al. [26] is Callas [25]. This work builds upon what was demonstrated with Salt's methodology of partitioning transactions into sub-transactions by automating it through complex static analyzes and an iterative process for finding good transaction decomposition. Additionally, and similarly to what was done in STRIFE [13], Callas automatically groups and assigns each transaction group its own CC mechanism.

Callas achieves its desired properties through the usage of a novel CC algorithm called Modular Concurrency Control (MCC) and by applying chopping techniques (Shasha et al. [15]) for improved concurrency within transaction groups.

As described by Xie et al., MCC aims to "decouple the abstraction from the mechanism", i.e., separate ACID the abstraction from its implementation, and it does so by allowing transactions to be partitioned into different groups while also enabling each group to be assigned its own optimized CC mechanism. MCC also ensures isolation within groups and intra-groupings through the usage of a purposely developed *nexus locks*.

2.2.2 Different Take

A different take from what has been described for CC is the scheduling of transactions execution to the processor cores in in-memory databases. Sheng et al. [16] speculate that it is possible to increase throughput of transaction execution and reducing abort rate of transactions by analyzing the patterns of the transactions abort. In their work, two distinct machines learning based approach are proposed for doing that. One that uses supervised machine learning and another one that uses unsupervised machine learning [16, Section 1.3]. The first computes for transactions pairs probability of abort, while the second groups transactions that are likely to abort into the same group for execution. Transactions are then scheduled into multiple processors cores in order to obtain a greater degree of concurrency. Their work show improvement on transaction throughput in the TPC-C benchmark, providing preliminary evidence that in-memory databases could benefit for such approach.

2.3 Lazy State Determination

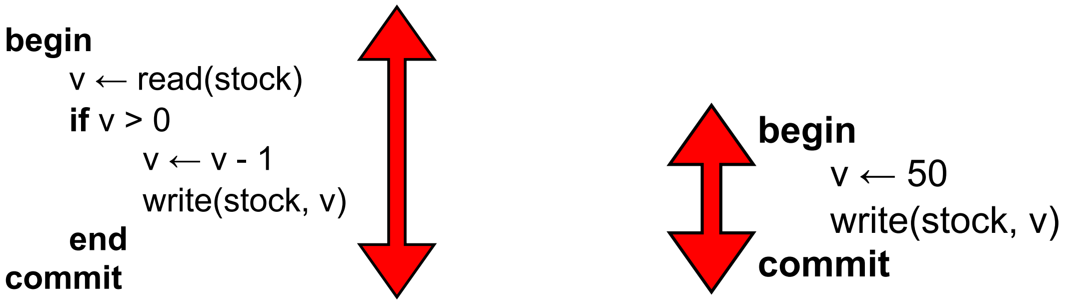
Reasoning with transactional systems is simplified by the ACID abstraction, where transactions appear to execute atomically without interference, despite being executed concurrently. A transaction that involves transferring money from one bank account to another will most likely involve multiple records that do not get updated atomically, even though the ACID abstraction entails that. A CC system would ensure that such transaction could run in parallel with others, while also ensuring that an inconsistent state is not visible to others. It is also the job of the CC to ensure the maximum throughput is achieved when transactions are running in parallel. However, when met with contention, CC systems lose performance. This happens mainly due to the fact that conflicting transactions tend to execute sequentially.

2.3.1 Overview

Typical CC systems can only make conservative assumptions on what a transaction is doing. If two transactions conflict, they will most likely need their execution to be synchronized. For example, two transactions that increase the available balance of the same bank account would conflict because they would be writing to the same database tuple. However, provided that the aggregate effects of both transactions is preserved, we could view them as non-conflicting.

To better understanding this issue, we may think about a transaction as having an execution window, which starts when the transaction makes its first query or update to the database and ends when the transaction commits. Figure 2.3 has a representation of the mentioned analogy. Transaction 1 starts by reading the stock value for a determined item, and then proceeds to updated it if above a certain value. Transaction 2 does a direct update to the stock value of the same item. The conflict window between two transactions is then defined by the intersection of their execution windows. There is a possible conflict when such conflict window is not empty.

From this observation, results that, if we decrease the size of a transaction windows we also decrease the conflicting window, which will increase the probability of successful commits and improve database throughput. It is based on this insight that Vale proposes Lazy State Determination (LSD) [23], a transactional Application Programming Interface (API) for conveying semantics to the database while improving concurrency and ensuring serializability of operations. For doing this, LSD converts traditional transactional API to be lazily evaluated and changes its operations to return *futures* instead of concrete values.



```
begin
    v ← read(stock)
    if v > 0
        v ← v - 1
        write(stock, v)
    end
commit
```

(a) Transaction that updates the stock by decrementing it a certain amount.

```
begin
    v ← 50
    write(stock, v)
commit
```

(b) Transaction that updates the stock value to 50.

Figure 2.3: Two transactions and their execution windows, represented by the red arrows.

2.3.2 Anatomy of a Future

A *future*, also known as *Promise*, is an abstraction used to refer to a proxy of a value that, as the name implies, will only be available at a future time. In its original form, *futures* encapsulate asynchronous tasks and their result. In doing so, it provides means for having deferred computations, allowing programs to create a *future* and only retrieving its result when needed. It is based on this abstraction that LSD transactions execute, with a change that the *futures* are not asynchronous.

A key insight from LSD is that, in general, transactions do not need to know the concrete state in which the database is for executing transactions, they only need to provide semantics on how to execute the transaction. Hence, LSD operations can be defined as something to be eventually executed, i.e., a *future*. In particular, LSD defers execution for commit time in order to allow for more concurrency.

In the end, what happens is that the amount of time transactions need to be in isolation is reduced. Isolation is still ensured at commit time.

2.3.3 Operations

LSD transactions operate in the typical client-server transactional model. In this model, clients issue traditional transaction commands such as *BEGIN* and *COMMIT* to, respectively, start and commit a transaction. The Traditional transactional API is evaluated immediately and so, returns concrete results, which is not compatible with LSD requirements of returning lazy objects, i.e., futures. This forces LSD to have its own API for transactions. Hence, clients that wish to reap the benefits of lazy evaluation must use LSD specific API. On the other hand, the database must also be compatible with said API. To this end, LSD changes the behavior of both 2PL and OCC CC protocols. It is to note that the client may still need to use the traditional API, however, the benefits of delayed execution and short isolation periods will not be present in such operations.

LSD operations are implemented as follows:

Table 2.3: Comparison between LSD and Traditional API. □ and Δ denote futures.

Operation	LSD API	Traditional API
BEGIN	Starts a new transaction	Same
READ(key) → □	Returns a future □ held by key *key*	Returns a value held by key *key*
READ(Δ) → □	Returns a future □ held by future key Δ	Not compatible
WRITE(key, X)	Not present	Writes the value X held by *key*
WRITE(key, □)	Writes the future value of □ into value held by *key*	Not compatible
WRITE(Δ,□)	Writes the future value of □ into value held by future key Δ	Not compatible
IS-TRUE(□)→boolean	Conditional checks for future □	Not compatible
COMMIT→boolean	Commits transaction	Same
ABORT	Aborts transaction	Same

***READ(key)* → □** — Returns to the client using LSD a future □, an opaque representation of the actual value held by *key*. From the database perspective, the future is known and resolvable. From the client perspective, it only knows the opaque representation of the future. Hence, other interactions with the database that need □ value need to use it as is, and the database will decide if and how to resolve it. The database on the other hand promises to lazily resolve it when needed.

***READ(Δ)* → □** — Returns a future □ held by future key Δ. This operation is similar to *READ(key)* → □, however, in this case, Δ future value needs to be resolved before returning future □. This is in order to avoid having "futures of futures".

WRITE(key, □) — Writes the future value □ to value held by *key*. From the application perspective, it is as if the operation has executed in the database and the value has been updated. On the other hand, from the database perspective this is an indication of an operation to be executed, not necessarily an immediate execution. The value of □ will only be resolved in the commit phase and, hence, the write operation will only take place at commit time as well.

WRITE(Δ,□) — Writes the future value □ to value held by future key Δ. This operation is similar *WRITE(key, □)*, however in this case the key is also a future, and consequently Δ needs to also be resolved in the commit phase (before □).

IS-TRUE(□)→boolean — Conditional check for future □. This operation allows transactions to operate based on concrete database states. As was the case for other LSD transactions, it does not expose the database state to the transaction. The idea here is to expose an abstract state instead of the concrete state. This allows for more concurrency and safeguards isolation because this operation is only executed in the commit phase, after □ is resolved.

COMMIT*→*boolean — Resolves all futures generated along the transaction execution and, if the transaction validation succeeds, commits to the database.

Table 2.3 compares LSD API with traditional Structured Query Language (SQL) API. Apart from the *BEGIN*, *ABORT* and *COMMIT* which remain the same, all other operations are excluded from LSD API.

For better understanding the LSD API, please refer to Figure 2.4, where we compare the general structure of a real transaction under LSD to the traditional API. In particular, this example models a simplified view of a transaction that decreases the stock number of an item in a warehouse. Note that the general structure remains unchanged when converting from a traditional to the LSD API.

```
begin
v ← read(stock)
if v ≥ qty
  v ← v - qty
  write(stock, v)
  commit
else
  abort
end if
```

(a) A transaction under traditional transactional API

```
begin
  □← read(stock)
  if is-true({□ ≥ qty})
    Δ← {□ - qty}
    write(stock, Δ)
    commit
  else
    abort
  end if
```

(b) A transaction under LSD API.

Figure 2.4: A comparison of traditional SQL API and LSD API.

As already mentioned, LSD also forces changes in the CC mechanism of the database. These are required in order to allow the database to create futures and resolve them at a later time. LSD changes OCC and 2PL similarly, but each has its own nuances in terms of implementation.

In general, both CC protocols require the addition of *future reads*, *future writes* and *future conditions* sets. These are for holding unresolved reads, unresolved writes and unresolved conditions respectively. The original *read* and *write* set are maintained, as the LSD API still allows for traditional API usage.

In case of OCC, where each record has a version associated with it, the protocol for commit works as follows:

1. Lock tuples in the *write* set;

2. Resolve and lock tuples in the *future read* and *future write* set;

3. Verification that tuples in the *read* set haven't changed in the meantime. If an item has changed, the transaction is aborted and moves to step 6.;

4. Resolves and validates conditions in the *future condition* set. If condition returns false, the transaction is aborted and moves to step 6;

5. Update tuples with final value;

6. Unlocks locked tuples.

2PL on the other hand, requires locks to be acquired when records are accessed. Read-locks and write-locks are still acquired by LSD implementation, however, most may be lazily evaluated in the commit phase. Although most of the locks may be deferred to the commit phase, there is still some cases where this might not be feasible. This is the particular case of the *READ(Δ)* → □ and *IS-TRUE(□)→boolean* operations. For the first one, LSD avoids having "futures of futures", so Δ is resolved immediately and locked. In the case of *IS-TRUE* operation, a new concept of locks is established, a *conditional lock*. In a sense, *conditional locks* introduce the required semantics for allowing readers and writers more flexibility over the aggregate effects two or more transactions would have. This is done by having two more types of locks:

Read condition lock — Installs a condition □ to the item at hand. This happens when an *IS-TRUE* operation is issued.

Write value lock — Acquired by transactions that wish to update a locked item with a new value. This value must respect all *read condition locks* imposed on it, otherwise it blocks and waits for other transactions to finish.

2.3.4 The First Prototype

The work of Vale [23, 22] focused on the implementation and validation of LSD in a NoSQL database called RocksDB [14]. Subtil [18] continued on his work by proposing a prototype implementation leveraging the Java Database Connectivity (JDBC) API to add futures.

This was the first attempt at using futures in a RDBMS context, and it did so by focusing on the client that is interacting with the database where the proposal was to implement the futures by extending the SQL language with new SQL directives, namely, `SELECT_LSD`, `INSERT_LSD`, `UPDATE_LSD` and `DELETE_LSD` opposing the well known `SELECT`, `INSERT`, `UPDATE` and `DELETE`.

This solution, however, proved unsuccessful as it led to additional steps when parsing a SQL statement, which in the end added time to the total execution time of the SQL statements.

Overall, the idea was that LSD insights can still be leveraged when we look only at the client side. Lazily evaluating queries can still reduce database latency and reduce contention in the database. This happens because the client code may continue execution and only get the results when needed. Additionally, this approach takes the problem from another angle and, in doing so, it leaves open the possibility of combining reviewed methods such as Silo [21], TicToc [27] Cicada [8] with the aim at optimizing the database functioning. Finally, this can also be seen as the first step to introducing the LSD to a RDBMS, since we still require some sort of API to interact with a database that uses LSD.

3

An Introduction to JDBC

This chapter provides an overview and highlights the main interfaces present in Java Database Connectivity (JDBC) Application Programming Interface (API).

3.1 Overview

The Java Database Connectivity (JDBC) is an API provided to Java applications that allow them to interact with database systems. The API defines a set of Java interfaces that encapsulate database functionality such as executing database queries, updating or inserting data, or even managing configuration information. To applications that need to interact with the database, only this high level interface is provided, while internally, each database vendor must implement the specifics for communicating and/or translations required to communicate with the database. Although the driver implementation is database specific and may depend on the variety of systems and hardware where Java is present, all JDBC drivers fall into one of the following categories:

Type 1 — *JDBC-Open Database Connectivity (ODBC) bridge*, is a type of driver it internally converts JDBC calls to ODBC calls. This driver is platform dependent since it uses ODBC which in turn depends on native libraries. It is not supported from Java 8, but it is still widely used to this date;

Type 2 — *Native API*, is a type of driver that is partially written in Java that relies on the native database API libraries to make database calls;

Type 3 — *Network Protocol*, is a type of driver that is written in Java and that relies on a network middleware to make database calls; This means that the driver will make a call to the middleware first, which will then make the database specific translations and make the call to the actual database;

Type 4 — *Database Protocol*, is a type of driver that is written in Java and uses the database protocol for communicating. It is the most widely used type of driver.

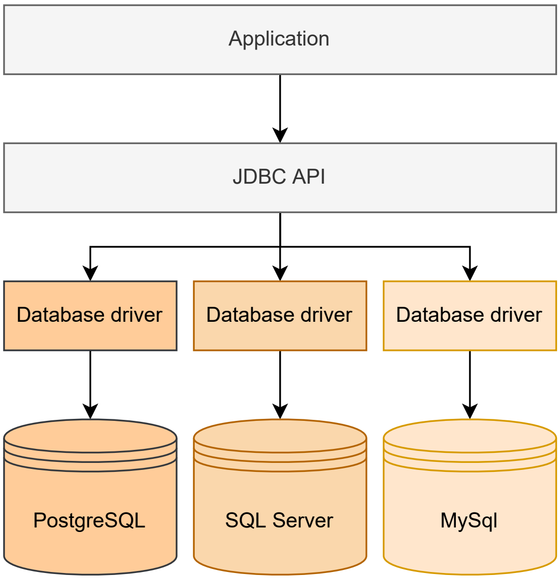


Figure 3.1: Generalized architecture of an application interacting with databases using JDBC API.

Figure 3.1 provides an overview of an application interaction with a database using the JDBC API. The application interacts with the API which then interfaces that interaction into the specific JDBC driver.

3.2 Interfaces

Standardizing database access requires a strict definition of the JDBC API, meaning that a clear separation between the database driver and the API must exist. This allows to have a vendor neutral API that allows multiple databases to be accessible via a single common interface. The API defines a big set of interfaces that may be used by a client when accessing a database or vendor when implementing a database driver. For brevity, we focus on the ones that were most relevant to the Lazy State Determination (LSD) work. These are listed below:

`Driver` — The interface used by the JDBC `DriverManager` to create connections to a database;

`DriverManager` — The interface used by the application to find the desired JDBC driver;

`Connection` — The representation of a real connection to a database;

`Statement` — This interface represents a static Structured Query Language (SQL) query which is executable in the database;

`PreparedStatement` — An extension to the `Statement` interface that adds parameters support, batching of SQL operations and statement caching in the database;

`ResultSet` — The representation of the results that are returned by the database on a `SQL` query execution.

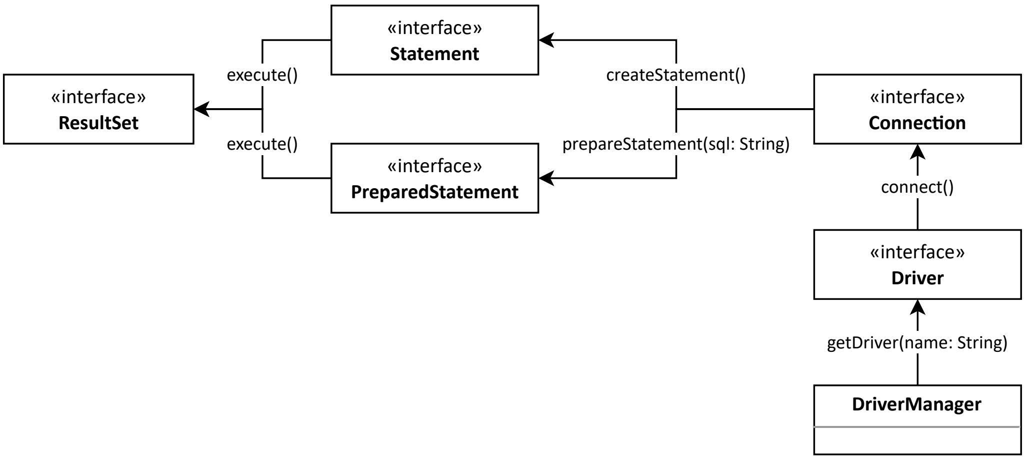


Figure 3.2: Diagram of the most relevant JDBC interfaces and their interactions.

Figure 3.2 shows a diagram with the mentioned interfaces and their interact with each other.

In the following subsections we go into more detail on the `Connection`, the `PreparedStatment` and the `ResultSet` interfaces. These are the ones that incurred the most changes after introducing the LSD API, hence, are the ones that required most of our attention.

3.2.1 Connection

As the name implies, the `Connection` interface holds the actual connection to the database. It allows interaction with the database by providing methods for creating database statements. The most relevant methods are listed below:

`createStatement(): Statement` — Creates a SQL query object represented by the interface of `Statement`;

`prepareStatement(sql: String): PreparedStatement` — Creates a SQL query object represented by the interface of `PreparedStatement`. Takes as parameter the SQL query;

`setAutoCommit(autoCommit: Boolean)` — Configures the intended commit behavior for the `Connection`. Takes as value a boolean which, when set to `true` will force the `Connection` to automatically commit to the database on every statement execution, and when set to false will force the `Connection` to only commit on an explicit `commit()` call;

`commit()` — Commits all executed statements to the database;

`rollback()` — Rolls back all executed transactions if auto commit is turned off.

3.2.2 Statement and PreparedStatement

The `Statement` and `PreparedStatement` are the interfaces that hold the actual SQL statement that is to be executed. Both interfaces support any type of SQL statement, with the difference being the fact that the `PreparedStatement` supports parameters while `Statement` does not. This makes the first one a better fit for dynamic statements (i.e. parametrized) to the database.

The two interfaces share a common set of features, with the most relevant being the support for batching of operations. This feature allows the client to execute multiple statements with a single hop happening between the client and database.

The most relevant methods that support the features of `Statement` and `PreparedStatement` are as follows:

`addBatch()` — Adds the current set of parameters to an internal list of parameters;

`clearBatch()` — Clears the list of parameters that the statement has;

`executeBatch(): IntArray` — Loads all the sets of parameters internal to the statement, forming multiple SQL queries, and then executes all of them in one hop to the database. It returns the number of affected tuples in the database;

`executeQuery(): ResultSet` — Executes the statement in the database and returns a `ResultSet` with the statement results;

`executeUpdate(): Int` — Executes an update or insert in the database and returns an integer representing the result of the operation. A positive value is returned if the statement execution affected any tuple, or 0 if no tuple was affected.

The support for parameters is done by setters' method in the `PreparedStatement` interface. Setters depend on the type we want to set, and for brevity, below we only describe the `setInt` method, as many setters exist depending on the data type.

`setInt(parameterIndex: Int, v: Int)` — Sets the parameter with index `parameterIndex` to value `v`. The parameter is set in the current list of parameters and is then affected to the query when the statement is executed.

3.2.3 ResultSet

The `ResultSet` is created after the execution of `executeQuery()` of a statement, and is a wrapper of the results returned by SQL select query.

This interface is intended to work as an iterator of the results, with each item in the iterator being a row. Each row is then divided by columns, which are accessible via getter methods. Since we may want different data types, the interface defines a method for every data type supported by the JDBC API.

The most relevant methods that support this interface are the following:

`next(): Boolean` — Advances to the next row in the results and returns a boolean indicating if there are more items in the `ResultSet` or not;

`getInt(columnLabel: String): Int` — This is an example of a getter method present in the `ResultSet`. This method returns the currently positioned row in the ResultSet from the column with label `columnLabel`. An equivalent getter using column indexes is also available with `getInt(columnIndex: Int): Int`. Other getters exist for different types of data;

3.2.4 Others

There are many other interfaces that support the JDBC API, such as the `DriverManager` and the `Driver`. The `DriverManager` is responsible for first the correct `Driver` based on the connection URI and then to return a connection to the database (the API `Connection`), while the `Driver` is responsible for creating the actual database connection based on the connection URI it gets from the `DriverManager`. The connection URI used by both interfaces is in the form of $jdbc :< vendor >: // < database >$, where $< vendor >$ is the database vendor and $< database >$ is the URL of the database.

Other interfaces also exist, but they are not relevant for the work present in this dissertation.

3.3 An Example

Recalling all the interfaces presented in the previous sections, we provide an example (Listing 3.1) of a transaction that uses JDBC and *Kotlin* [1] to decrement the stock number of an item in the database. The first step (line 1) is to create a connection using the `DriverManager`, which internally uses the database specific `Driver` to create the connection. The connection is then used to create a `PreparedStatement` (lines 3–4) that is used to retrieve the stock of item with `id` of 1. The `PreparedStatement` is then executed (line 6–7) to get the stock value via a `ResultSet`. The `ResultSet` is then used as a parameter to a final update query that decrements the stock value of item with `id` of 1 (lines 9–15).

Listing 3.1: JDBC transaction for updating the stock number of item 1 in the a database.

```
 1 val connection = DriverManager.getConnection("jdbc:postgresql://localhost/database")
 2 
 3 val stockStatement = connection.prepareStatement("SELECT stock FROM items
 4     WHERE id = 1 FOR UPDATE")
 5 
 6 val resultSet = stockStatement.executeQuery()
 7 resultSet.next()
 8 
 9 val updateStock = connection.prepareStatement("UPDATE items
10     SET stock = ? - 1
11     WHERE id = 1")
```

[1] https://kotlinlang.org/

```

updateStock.setInt(1, resultSet.getInt(1))

updateStock.executeUpdate()

connection.commit()
```

4
LSD Strategy

Extending a database with Lazy State Determination (LSD) requires adapting the client to handle futures it may receive as response from any Structured Query Language (SQL) statement. In this chapter we focus on the strategy used to represent a future as well as the desired properties we need futures to respect. We also go into detail on a solution implemented using the Java Database Connectivity (JDBC) Application Programming Interface (API).

4.1 Properties

A typical operation in a database has immediate effects when executed. Reasoning with this is easy, as the state is immediately affected. This type of workflow being straightforward to understand programmers. However, a future is not an immediate operation, nor an asynchronous or parallel execution, it is the delay of the operation to commit time. Extending any client with non-immediate operations, such as, asynchronous and parallel execution, is a non-trivial task that takes a tool on the readability of computer programs. This is particular relevant when we think about futures as being independent of the programming language and database. So, one of our goals is to represent futures in such a way that, little to no readability impact is felt when using Lazy State Determination (LSD). To achieve such goal, we define some properties that must followed when representing futures. The properties are listed below:

Resolvability — A future may be computed, or resolved, at any time by anyone;

Immutability — Resolving a future yields the same result;

Encapsulation — A future has all the information it needs.

In the next subsections, we go into more detail over each property here defined, and its implications.

4.1.1 Resolvability

As the name implies, the resolvability property means that the future must be resolvable. Moreover, the future must be resolvable by anyone who needs it at any time. This property allows a future A that depends on future B to resolve future B. This gives flexibility by allowing the creation of dependencies between futures. These dependencies can then be chained between different futures, creating a dependency graph.

4.1.2 Immutability

The immutability property states that the value returned by a future, when executed, should always be the same. This property is here in order to ensure that the same state is seen along the program execution.

4.1.3 Encapsulation

The encapsulation property ensures that a future knows to resolve itself. This includes knowing the dependencies it has and how to resolve them, and also know to execute the operation it was defined to execute. Meeting this property ensures transparency to the programmer, as it hides complexity and dependencies the future may have.

4.2 Future

Defining the desired future properties gives us a clear representation of what a future should look like. Listing 4.1 shows a representation of a future. We define that a future has always a function (*fun* in the listing) that captures the operation to execute, an optional list of dependencies (*dependencies* in the listing) and a result (*result* in the listing) for storing the outcome of the operation when executed. It should also have a single function to resolve the value of the future.

Listing 4.1: The representation of a future.

```
class Future
  member:
    dependencies <- [Future]
    fun <- function()
    result <- empty

  function:
    resolve(): V
```

The algorithm for resolving a future is simple and is captured in algorithm 1 by the *resolve* function. First, the algorithm starts by checking if the future was already computed by looking at the value of the *result* member. When the value is not *empty*, it returns the value of the *result*. When the value of the result is *empty*, it iterates over each dependency it has and then resolves them. After resolving all dependencies it executes the future operation fun, stores the result in the $result$ member and then returns the $result$ value.

Algorithm 1: Algorithm for the computation of the value of a future.

```
members : dependencies: [$Future$], fun: $function$, result: $V$
output  : The value of the future

function resolve(): $V$ is
    if $result = empty$ then
        foreach $d \leftarrow dependencies$ do
            $d.resolve()$
        end
        $result \leftarrow fun()$
    end
    return $result$
end
```

Going back to the properties presented in Section 4.1 and cross-checking them with the future representation, we can see that:

Resolvability — Ensured by the function `resolve` present in the future and `dependencies` member;

Immutability — Ensured by the function `resolve` and member `result`;

Encapsulation — Ensured by a single function `resolve`, which is all we need to execute.

5

JDBC WITH LSD

This chapter focuses on the adaptations that were done to Java Database Connectivity (JDBC) to add support for the Lazy State Determination (LSD) Application Programming Interface (API).

5.1 Overview

The most obvious way of adding LSD to the client API would be to modify the database driver to support lazy operations. This would be the ideal, however, would mean that a lot of effort would be required for normalizing usage of LSD on multiple Relational Database Management System (RDBMS). To avoid such scenario, we started with the goal of creating a LSD interface that would abstract all LSD operations and would serve as a middleware for a real database driver. This enables LSD to be paired with any RDBMS database driver without changes to their specific code. To achieve this, we define an extended JDBC API with LSD support that meets the properties defined in Section 4.1. The extended API defines a new set of interfaces and methods that allows the `Connection` to return new a types of `Statement` and `PreparedStatement` that supports futures; the `PreparedStatements` to support future parameters and, on its execution, to return a new type of `ResultSet` that supports futures; the `ResultSet` to, on a getter call, to return future values. Defining and implementing such interfaces, means creating a *Type 3* (Section 3.1, page 22) JDBC driver where the middleware is local to the client.

We aimed at supporting all Structured Query Language (SQL) operations available in the JDBC API while maintaining interoperability with the original interfaces. Over the course of the next sections we will review each of the interfaces and go into it detail over its implementation.

5.2 Interfaces

In this section we give an overview of the interfaces used to implement LSD API into JDBC. The details of the interfaces here presented use `Kotlin` syntax for better comparison with the JDBC. However, this needs not to be the case as the JDBC-LSD API could have been implemented in any Java virtual machine (JVM) based language. Moreover, other non-JVM languages that do not have JDBC, have a similar database interaction model, which makes the details here present, work as a guideline for other languages ports.

Before diving into how those interfaces were extended, we first focus on how to represent a future. The Future interface is presented in Listing 5.1. This interface is simple, yet powerful, as it serves as basis for other interfaces of the extended JDBC API. The resolve method computes the value of the future. Its return value is dependent on the class that implements it. The dispose method is responsible for doing any clean-up the future deems necessary.

Listing 5.1: The representation of a Future in *Kotlin* syntax.

```
interface Future<T> {
  fun resolve(): T
  fun dispose()
}
```

Besides defining the interface that allows future execution, we also defined an interface called ResultChain with the goal of allowing the programmer to process the results of operations. The idea here is that, after the future execution we should be able to do something based on the future results, this gives the ability to programmer to log results, emit messages or event abort the entire transaction. The interface is presented in Listing 5.2. There we define a single method, that takes a Consumer [1] as parameter, where the parameter is a function that will execute something on the value that it takes.

Listing 5.2: The representation of a ResultChain in *Kotlin* syntax.

```
  interface ResultChain<T> {
    fun then(function: Consumer<T>)
  }
```

5.2.1 Connection

The Connection interface is the starting point for creating SQL queries statement objects. This is done via the createStatement and prepareStatement (Chapter 3) methods. As these are regular statements that will not be lazily executed, we need to extend the Connection to have new methods that return statements capable of being lazily evaluated. To achieve this, we defined a new interface with name FutureConnection that extends the Connection interface, so that standard JDBC operations are still available.

Listing 5.3: The representation of a FutureConnection in *Kotlin* syntax.

```
interface FutureConnection: Connection {
  fun createFutureStatement(): FutureStatement

  fun prepareFutureStatement(sql: String): PreparedFutureStatement
```

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Consumer.html

```
}
```

The FutureConnection interface is presented in Listing 5.3. This interface defines two new methods, the prepareFutureStatement and the createFutureStatement. The first one creates an object of type FutureStatement, an extension of the Statement interface, that enable lazy execution of the statement. The second one creates an object of type PreparedFutureStatement, an extension of the PreparedStatement interface, that supports lazy executions and future parameters. Going back to the example presented in Listing 5.9 the method, prepareFutureStatement (line 4 and 9) is used to create a future statement.

5.2.2 FutureStatement and PreparedFutureStatement

The Statement and PreparedStatement interfaces are what hold the actual SQL query and what allow their execution. These statements execute, upon request, the queries immediately, so we define a decorator interface over these statements in order to add lazy execution. With this in mind, we derive two new interfaces, FutureStatement and PreparedFutureStatement.

Listing 5.4: The representation of a FutureStatement in *Kotlin* syntax.

```
1  interface FutureStatement : Statement, Future<Any> {
2    fun executeFutureQuery(): FutureResultSet
3
4    fun executeFutureUpdate(): ResultChain<Int>
5
6    fun addFutureBatch(sql: String)
7
8    fun executeFutureBatch(): ResultChain<IntArray>
9
10   override fun resolve(): Any
11 }
```

Listing 5.4 demonstrates the FutureStatement interface. This extends the Statement by adding four new methods, the executeFutureQuery, the executeFutureUpdate, the addFutureBatch and the executeFutureBatch, and it also implements the Future interface previously explained in Section 5.1. The executeFutureQuery (line 2) has the responsibility of defining that the operation to execute is a query and for creating a FutureResultSet. The addFutureBatch (line 6) adds a SQL statement to the list of batching operations. The executeFutureUpdate (line 8) defines that the operation to execute is an update to the database and returns a ResultChain. The final method (line 10) represents the resolution, or execution, of the actual Statement, be it a regular or batching operation.

Listing 5.5: The representation of a PreparedFutureStatement in *Kotlin* syntax. For brevity only a few setters are show here, but the PreparedFutureStatement supports the same setters as the ones provided by PreparedStatement.

```
interface PreparedFutureStatement : PreparedStatement, FutureStatement {
  fun setFutureInt(parameterIndex: Int, x: Future<Int>)

  fun setFutureDouble(parameterIndex: Int, x: Future<Double>)

  fun setFutureFloat(parameterIndex: Int, x: Future<Float>)

  fun setFutureString(parameterIndex: Int, x: Future<String>)

  fun setFutureObject(parameterIndex: Int, x: Future<Any>)

  fun setFutureTimestamp(parameterIndex: Int, x: Future<Timestamp>)

  fun addFutureBatch(): ResultChain<IntArray>
}
```

Listing 5.5 shows the interface we used for defining the `PreparedFutureStatement`. This interface extends both the `PreparedStatement` and the `FutureStatement` by adding several new parameter setters to support future parameters. The `FutureStatement` method `addFutureBatch` in this interface is ignored in favor of a new one that takes no parameters. This happens because this interface represents a template of a SQL statement.

5.2.3 FutureResultSet

The `JDBC ResultSet` gets the results immediately after a `Statement executeQuery` execution. This is not the case for the `FutureStatements` defined in Subsection 5.2.2. Hence, we defined a `FutureResultSet` with added methods that are aware that the future it was originated from may not yet have been resolved. For this we present the interface shown in Listing 5.6. The interface extends three other interfaces, a `Future`, a `ResultChain` and a `ResultSet`. The first one defines the interface as a `Future` so that it can be resolved (line 14) by other futures, the second one allows for defining a `then` (line 16) chain after the future is resolved, and, the final one gives us interoperability with JDBC `ResultSet`. With respect to the methods it has, it defines alternatives to the original JDBC getters, with its various *getFutures*.

Listing 5.6: The representation of a `FutureResultSet` in *Kotlin* syntax. For brevity only a few getter are show here, but the `FutureResultSet` supports the same getters as the ones provided by `ResultSet`.

```
 1 interface FutureResultSet: Future<ResultSet>, ResultChain<ResultSet>, ResultSet {
 2     fun getFutureInt(columnIndex: Int): Future<Int>
 3
 4     fun getFutureDouble(columnIndex: Int): Future<Double>
 5
 6     fun getFutureFloat(columnIndex: Int): Future<Float>
 7
 8     fun getFutureString(columnIndex: Int): Future<String>
 9
10     fun getFutureObject(columnIndex: Int): Future<Any>
11
12     fun getFutureTimestamp(columnIndex: Int): Future<Timestamp>
```

```

    override fun resolve(): ResultSet

    override fun then(function: Consumer<ResultSet>)
}
```

5.2.4 FutureCondition

Maintaining expressiveness of operations was also taken in consideration by the LSD API. One may need to branch the program execution based on a condition. This aspect is captured by the $IS-TRUE(\square) \rightarrow boolean$ operation and is translated into our JDBC-LSD API using the `FutureCondition` interface. The responsibility of creating `FutureConditions` is given to the `FutureConnection` interface via the `isTrue` method. Listing 5.7 details the signatures used to define the behavior of $IS-TRUE$. The first `isTrue` method takes a function that returns a boolean value as parameter. The second `isTrue` method extends the first one by allowing a combination of multiple function parameters in the form of an `Invariant`. An `Invariant` here acts as a logical evaluation of multiple functions using the logical operators *and* and *or*. The two methods are only evaluated at commit time.

Listing 5.7: Implementation of the $IS-TRUE$ API into the `FutureConnection` in *Kotlin* syntax.

```
interface FutureConnection: Connection {
  fun isTrue(condition: () -> Boolean): FutureCondition

  fun isTrue(condition: Invariant): FutureCondition
}
```

Listing 5.8: The representation of a `FutureCondition` in *Kotlin* syntax.

```
interface FutureCondition: Future<Boolean> {
  fun whenTrue(future: () -> Unit): FutureCondition

  fun whenFalse(future: () -> Unit): FutureCondition

  fun resolve(): Boolean
}
```

Listing 5.8 shows the `FutureCondition` interface definition. This interface represents the future value of the condition and defines the behavior it should take after knowing the condition value. The `resolve` method takes care of evaluating the condition and executing a follow-up function. This action is defined by the function parameters that `whenTrue` and `whenFalse` take.

5.3 Implementation

Previous sections gave a high level overview on how the several interfaces interact to form a transaction execution under the LSD API, however, there are some details we believe are important in order to fully understand this work. The details of the interfaces presented before used `Kotlin` syntax for better comparison with the JDBC API, but for this section, this needs not be the case, as we want to concentrate on the algorithms we used to support those interfaces.

All the interfaces were implemented using the decorator programming pattern. This allowed, not only to extend the API to support futures, but also to keep existing API functionality. Using this strategy leads us to have, every JDBC-LSD interface with internal backing JDBC counterpart. This means that, the `FutureConnection` has an internal `Connection`, the `PreparedFutureStatement` has an internal `PreparedStatement`, and, the `FutureResultSet` has an internal `ResultSet`.

5.3.1 FutureConnection

The whole idea around the LSD revolves around having operations being delayed to commit time. This is achieved by having the `FutureConnection` creating and storing the futures to execute. With that, the `FutureConnection` is then able to resolve all futures. Algorithm 2 highlights the steps for executing the `commit` operation. First and before executing this operation, `futures` and `connection` must be available. The first one represents the `FutureConnection` stored futures, while the second one represents a real connection to the database. The execution is then to, first resolves all `futures` and then to execute the actual commit to the database.

Algorithm 2: The `FutureConnection` algorithm for executing a commit to the database.

```
members: futures: [$Future$], connection: $Connection$
function commit() is
    foreach $f \leftarrow futures$ do
        $f.resolve()$
    end
    connection.commit()
end
```

5.3.2 FutureStatement and PreparedFutureStatement

In order to support futures in the `FutureStatement` and `PreparedFutureStatement`, we have to enable these interfaces to store the operation that they will eventually execute, and more specifically for `PreparedFutureStatement`, to store a list of future parameters it will eventually set. Executing a future query or setting a future parameter in a `PreparedFutureStatement`, from the programmers' perspective, is similar to executing a regular query or setting a regular parameter. Internally, however, there is a big difference. For both cases, there are some requirements that a `FutureStatement` and

PreparedFutureStatement must fulfil. We expect that the implementation of these interfaces to have available a backing Statement (statement), as described in Section 3.2, an executable function, in this case another future (future), and a store with all the dependencies it has (futureParameters), as defined for a future in Subsection 2.3.2.

For the scenario where we want to set a future parameter, the algorithm is described in Algorithm 3. As the first step of the algorithm, we start by creating a future for setting the value. This future is implemented by a function that will set and resolve the value of the future parameter in the backing statement (statement). After creating the future, we add it to the list of future parameters.

Algorithm 3: The PreparedFutureStatement algorithm for setting a future parameter.

```
members : statement: Statement, future: Future, futureParameters: [Future]
function setValue(parameter, futureValue) is
    future = createFuture(() → {
        statement.setValue(parameter, futureValue.resolve())
    } )
    futureParameters.add(future)
end
```

For the scenario where we want to execute a future query, the algorithm is described in Algorithm 4. The algorithm starts by creating a function that will, execute the intended operation, which in this case is executeQuery. It finishes by returning a FutureResultSet that will have a reference to the function that will generate the actual result.

Algorithm 4: The PreparedFutureStatement algorithm for executing a future query.

```
members : statement: Statement, future: Future, futureParameters: [Future]
function executeFutureQuery(): FutureResultSet is
    future = createFuture(() → {
        statement.executeQuery()
    })
    return futureResultSet(future)
end
```

Finally, and after defining the parameters and operations to execute, we look at the resolve algorithm for the PreparedFutureStatement in Algorithm 5.

5.3.3 FutureResultSet

The ResultSet, as was the case with the other interfaces, needs to store additional information to support futures. This is achieved by having the FutureResultSet to store a reference to the FutureStatement that generated it. This allows the FutureResultSet to, when needed, execute the resolve of the FutureStatement. This is precisely what happens when we try to get a future value and algorithm 6 describes that. First, we create

Algorithm 5: The `PreparedFutureStatement` algorithm for resolving its dependencies and executing the actual operation to the database.

```
members: statement: $Statement$, future: $Future$, futureParameters: [$Future$]

function resolve() is
    foreach $f \leftarrow futureParameters$ do
        $f.resolve()$
    end
    return future.resolve()
end
```

a new future that is implemented by a function that will resolve the `FutureResultSet`'s `futureStatement` and then get the actual value from the real `ResultSet`. After that, we return the future for the value we were looking for.

Algorithm 6: The `FutureResultSet` algorithm for getting a future value from a `ResultSet`.

```
members: futureStatement: $Future$, result: $ResultSet$

function getValue(column): Future is
    future = createFuture(() → {
        resolve()
        return result.getValue(column)
    })
    return future
end
```

As for the resolve of the `FutureResultSet`, this is demonstrated in algorithm 7. The implementation is very simple, first checks if it has resolved the `futureStatement` before by checking if the `result` is empty. If it is empty, it will call the `futureStatement` `resolve`. After that, it returns the real `ResultSet`.

Algorithm 7: The `FutureResultSet` algorithm for resolving to a `ResultSet`.

```
members: future: $Future$, result: $ResultSet$

function resolve() is
    if $result = empty$ then
        result = futureStatement.resolve()
    end
    return result
end
```

5.3.4 FutureCondition

Branching code execution is essential when programming, and this is supported with the `FutureCondition`. To achieve this, the `FutureConnection` interface provides the `isTrue` method that takes a function that returns a boolean as parameter. This function, is the basis for the `FutureCondition` as it is inside it that the programmer may resolve any future it needs and then return the boolean result. The methods of `whenTrue` and `whenFalse` define the branches to follow by taking both as parameters a function. These functions are then members of `FutureCondition` implementation and allow the branching execution. The `resolve` method is responsible for doing this with its algorithm present in Algorithm 8.

Algorithm 8: Algorithm for resolving the results of a `FutureResultSet`.

```
members: conditionFunction: $function$, trueFunction: $function$, falseFunction:
         $function$
function resolve() is
    result = conditionFunction()
    if $result = true$ then
        trueFunction()
    else
        falseFunction()
    end
end
```

5.3.5 Others

One interface that required specific adaptation for the JDBC-LSD was the `Driver`. In order to wrap the `Connection` with a `FutureConnection` we defined that we should wrap the database connection URI. Doing, allows us to not only to decorate the `Connection` with future support, but also takes us to a state where we support any other database that supports JDBC. The connection URI we chose is in the form of $jdbc : lsd :< vendor >: //< database >$. This specific connection URI will only match the JDBC-LSD `Driver`, which after matched will remove the $lsd :$ from the URI and then search for the actual database driver.

5.4 An Example

Serving as a demonstration of the JDBC-LSD API, Listing 5.9 presents an adaptation of the transaction present in Listing 3.1.

Listing 5.9: JDBC-LSD transaction for updating the stock number of item 1 in a database.

```
val connection = DriverManager
    .getConnection("jdbc:lsd:postgresql://localhost/database")

val stockStatement = connection.prepareFutureStatement("SELECT stock FROM items
    WHERE id = 1 FOR UPDATE")

val futureResultSet = stockStatement.executeFutureQuery()

val updateStock = connection.prepareFutureStatement("UPDATE items SET stock = ? - 1
    WHERE id = 1")
```

```
11
12 val futureStockValue = futureResultSet.getFutureInt(1)
13
14 updateStock.setFutureInt(1, futureStockValue)
15
16 updateStock.executeFutureUpdate()
17
18 connection.commit()
```

The example starts by retrieving a `FutureConnection` (line 1) from the `DriverManager`. This `FutureConnection` is then used to create a `PreparedFutureStatement` (line 4–5) for finding the stock of item 1. The `PreparedFutureStatement` is then used for creating a `FutureResultSet` that will create a future for the stock of the item (line 7). After creating the `FutureResultSet`, the `FutureConnection` creates another `PreparedFutureStatement` (line 9–10) for decrementing the stock value (line 16) of the item. Before setting up the future to update the stock of the item, we first create a future (line 12), from the `FutureResultSet`, with the future value of the stock. This is followed by setting (line 14) the future parameter of the future stock update with the value of the future item stock. Finally, and after setting up all the futures to execute, the `FutureConnection` resolves all the futures it holds (line 18) and then commits (line 18) the outcome to the database.

The execution of this transaction creates a graph of future dependencies that is highlighted in Figure 5.1. As can be seen in the figure, each node represents a JDBC-LSD and JDBC interface. The edges are numbered and are created by the interactions each interface has with one another. So, starting from the `FutureConnection` in the graph, and referring to the transaction in Listing 5.9, we know that we must resolve two `PreparedFutureStatements`, the `FutureStockQuery` and the `FutureStockUpdate`. The first one to resolve (edge 1) is the `FutureStockQuery` since the `FutureStockUpdate` depends on its value (line 14 in Listing 5.9). Since this is a future, we need to execute its resolve operation (edge 2), which will execute the actual query to the database. After resolving the `FutureStockQuery` stock value, the commit will resolve the `FutureStockUpdate` (edge 5). For that, the `FutureStockUpdate` will resolve its dependencies, namely setting the stock value in the statement (edge 6). This, however, triggers a new resolve call (edge 7) to the `StockQuery` to get the actual stock value. After setting the stock value parameter of the update statement, the `FutureStockUpdate` will execute the update statement. Finally, the `FutureConnection` will commit to the database.

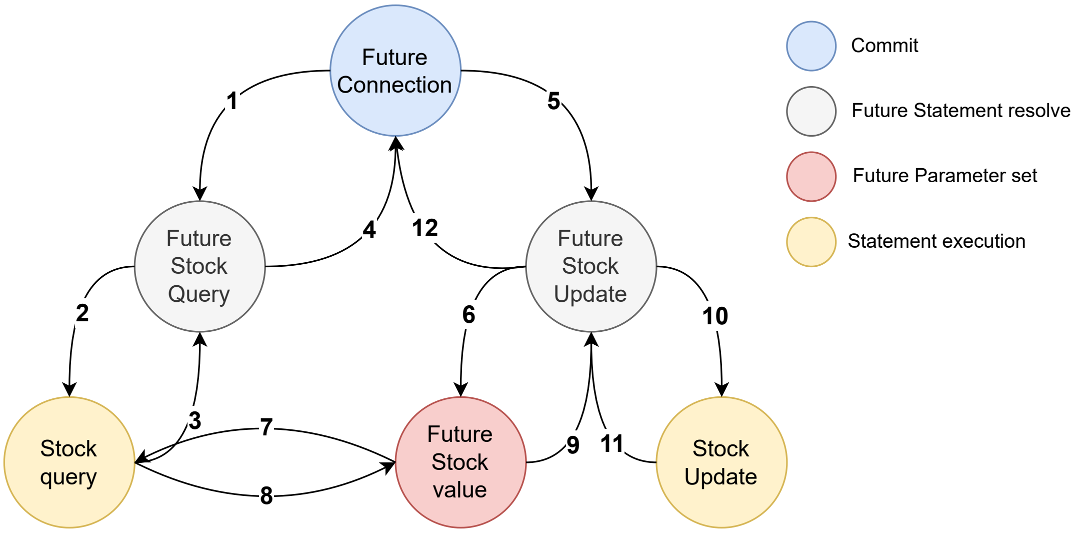


Figure 5.1: An example visualization of the dependency graph that the commit needs to resolve when executing a transaction using JDBC-LSD, more specifically, Listing 5.9. The edges in the graph have a numbered label that indicates execution order. The nodes represent a single JDBC-LSD and JDBC interface generated from the transaction execution. Finally, each node is color coded with the operation that triggers the edge creation.

6

Evaluation

In this chapter, we elaborate on the testing environment we used for validating our work, the tests that we performed and the results we obtained.

6.1 Test setup

The chosen Relational Database Management System (RDBMS) for validating our work was the *PostgreSQL* [1]. This database was chosen because of its popularity and it being an Open Source database [2]. Other databases are also supported, since the Java Database Connectivity (JDBC)-Lazy State Determination (LSD) driver is database agnostic, allowing it to function with any database that has a JDBC implementation.

The tests were conducted in the *NOVA LINCS Cluster* in a distributed fashion across six different interconnected nodes, all running Debian 11. One of the nodes took the responsibility of executing the *PostgreSQL* database while the others acted as test workers executing queries in the database. The node that handled the database had as specification an AMD EPYC 7281@2.7GHz 16 Cores, 128 GB of RAM. As for the worker nodes, they all had the same specifications, each with 2× AMD Opteron 2376 CPU@2.30GHz and 16 GB of RAM. The database and worker nodes were connected via 1 Gbps network connections.

In regard to the test candidates, these were the standard JDBC driver for *PostgreSQL* with version *42.3.3* and the newly implemented JDBC-LSD driver. Both candidates were evaluated using a standard test for RDBMS, the *TPC-C* (Appendix A) benchmark.

The TPC Benchmark C (TPC-C) benchmark is particularly helpful because on the one hand, it captures the entire set of interfaces that were presented in the Chapter 5, and on the other hand, it enables us to control the contention level we want for a test. For the first one, the two core transactions of the benchmark, *New-Order* and *Payment*, are able to capture samples of reads, writes and execution branching. For the latter, the ability to control the desired contention is done via the number of warehouses (see Appendix A for more details) the benchmark uses. More specifically, by reducing the number of

[1] https://www.postgresql.org/

[2] https://opensource.org/licenses/PostgreSQL

warehouses, we are increasing the likelihood of transactions to conflict, on the other hand, if we increase the number of warehouses, then the chance of transactions conflicting is reduced. The warehouse parameter enabled us to test the newly implemented JDBC-LSD driver in two different scenarios, a high contention scenario and low contention scenario.

6.2 Results

In this section, we look at the results obtained for a high and low contention scenarios. For each of these scenarios, we evaluate the performance of JDBC and JDBC-LSD over the full TPC-C benchmark, the *New-Order* transaction and the *Payment* transaction. The decision behind testing different transactions comes from the fact that only in the *Payment* transaction we would be able to test a *FutureCondition*. In case of the full benchmark, some transactions were not converted to LSD because of their read only nature, which would not get any benefit of delayed operation execution.

In total, we execute six different tests and, for each one, we evaluate the following results:

Throughput — The number of successful transactions executed per minute (*tpmC*);

Latency — The execution time, measured in milliseconds, from the moment the client starts execution code of a transaction;

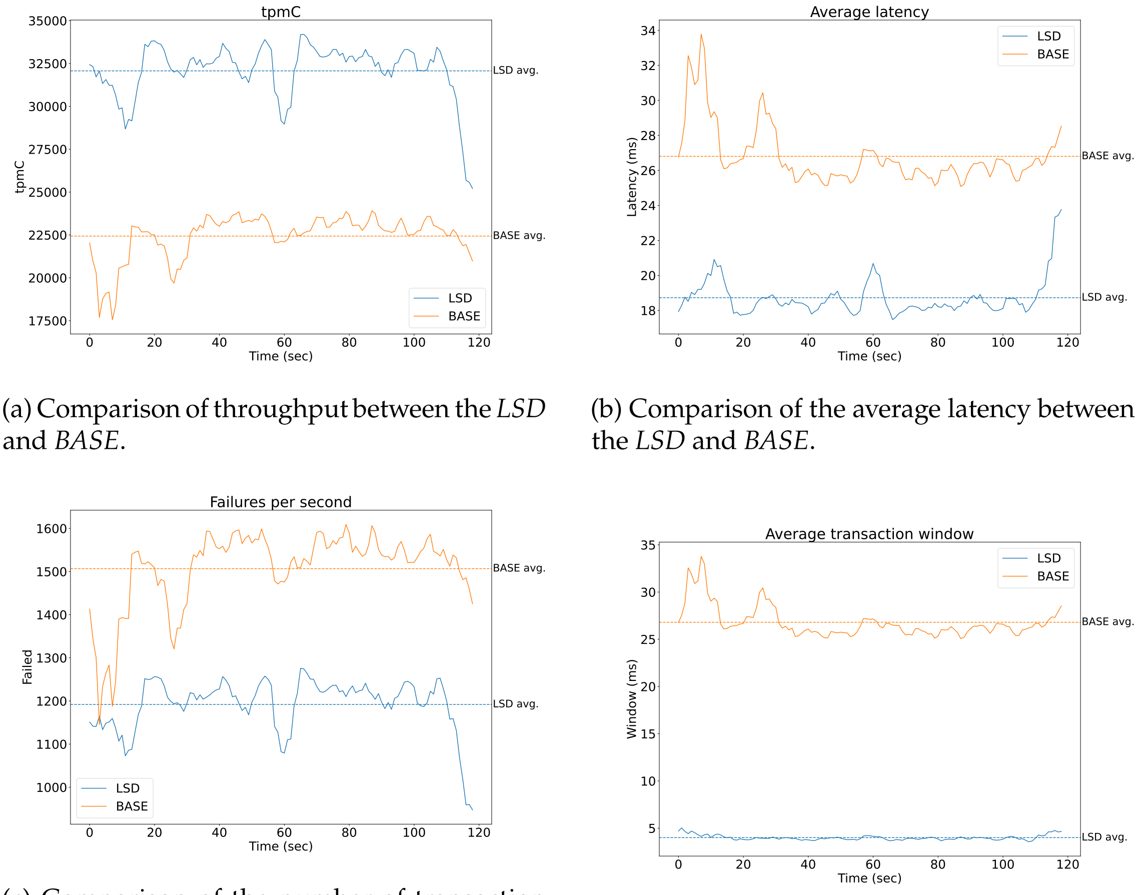
Failure — The number of transactions that failed execution due to conflicts;

Window — The execution time, measured in milliseconds, between the first query the client executes in the database until commit.

6.2.1 High contention

As mentioned, the warehouses number control the contention in the database. Hence, for testing the high contention scenario, we have reduced the number of warehouses to one, yielding maximum contention.

***New-Order* transaction** Starting with an analysis over the Figure 6.1, which shows the results of the *New-Order* transaction results, we can see that we achieve a much higher throughput (Figure 6.1a) with *LSD*. More specifically, the throughput gain over the *BASE* is about 50%. In regard to the total latency in Figure 6.3b, we can see that the latency we get for LSD is a much smaller (40%) than the *BASE* case, which aids in improving the throughput of the transaction. The results of LSD are superior because, as contention increases, so do the chances of conflicts, so reducing the latency greatly reduces the chances of conflicts. Figure 6.1d corroborates this affirmation by showing that the *LSD* transaction window is much lower than the transaction window of the *BASE* case, which helps in reducing the transaction latency. When we look at the failed transactions in



(a) Comparison of throughput between the *LSD* and *BASE*.

(b) Comparison of the average latency between the *LSD* and *BASE*.

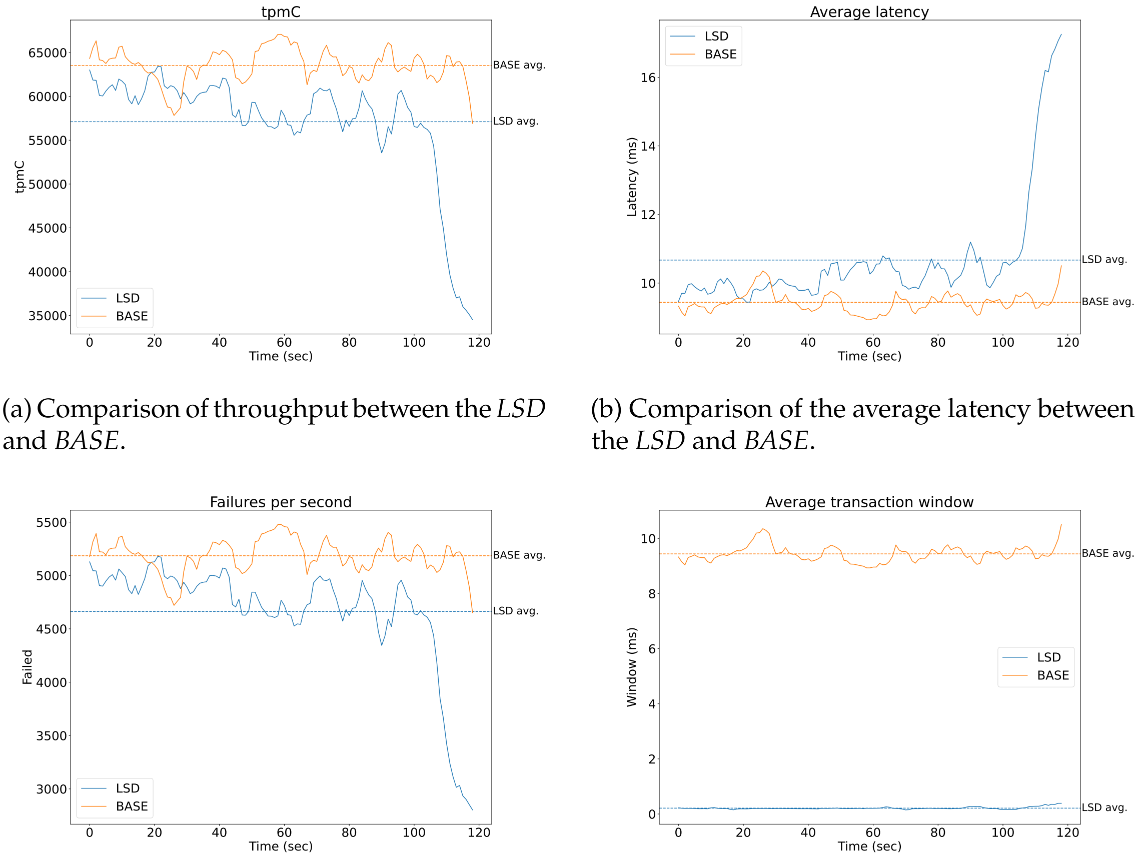
(c) Comparison of the number of transaction failures between the *LSD* and *BASE* in a high contention scenario.

(d) Comparison of the average transaction window between the *LSD* and *BASE*.

Figure 6.1: Throughput, latency, transaction failures and transaction window after executing only the *New-Order* transaction with high contention. JDBC-LSD in represented by *LSD* and the standard JDBC is represented by *BASE* line.

Figure 6.1c, we see, again, better results being achieved by the *LSD API*, with a reduction of about 27% in failed transactions.

***Payment* transaction** The results for this transaction are expressed in Figure 6.2. There, we see that, in Figure 6.2a, *LSD* achieves a lower throughput than the *BASE* case. The latency (Figure 6.2b) is also higher for the *LSD*, but the amount of aborted transactions (Figure 6.2c) does not increase for *LSD*. Inspecting the transaction window (Figure 6.2d) shows that its value is quite small, however, the transaction latency (Figure 6.2b) is higher. These results made us question our implementation, and upon inspection, we found the cause for this difference to be the *FutureCondition* that the *Payment* transaction relies on. More specifically, the issue is that when defining the branch of a `FutureCondition`, we are actually defining another future. This future, however, may have any type of work, be it a database interaction or a local code execution. In the case of the *Payment* transaction, the future branch that we define for the `FutureCondition`, is the creation of another future



(a) Comparison of throughput between the *LSD* and *BASE*.

(b) Comparison of the average latency between the *LSD* and *BASE*.

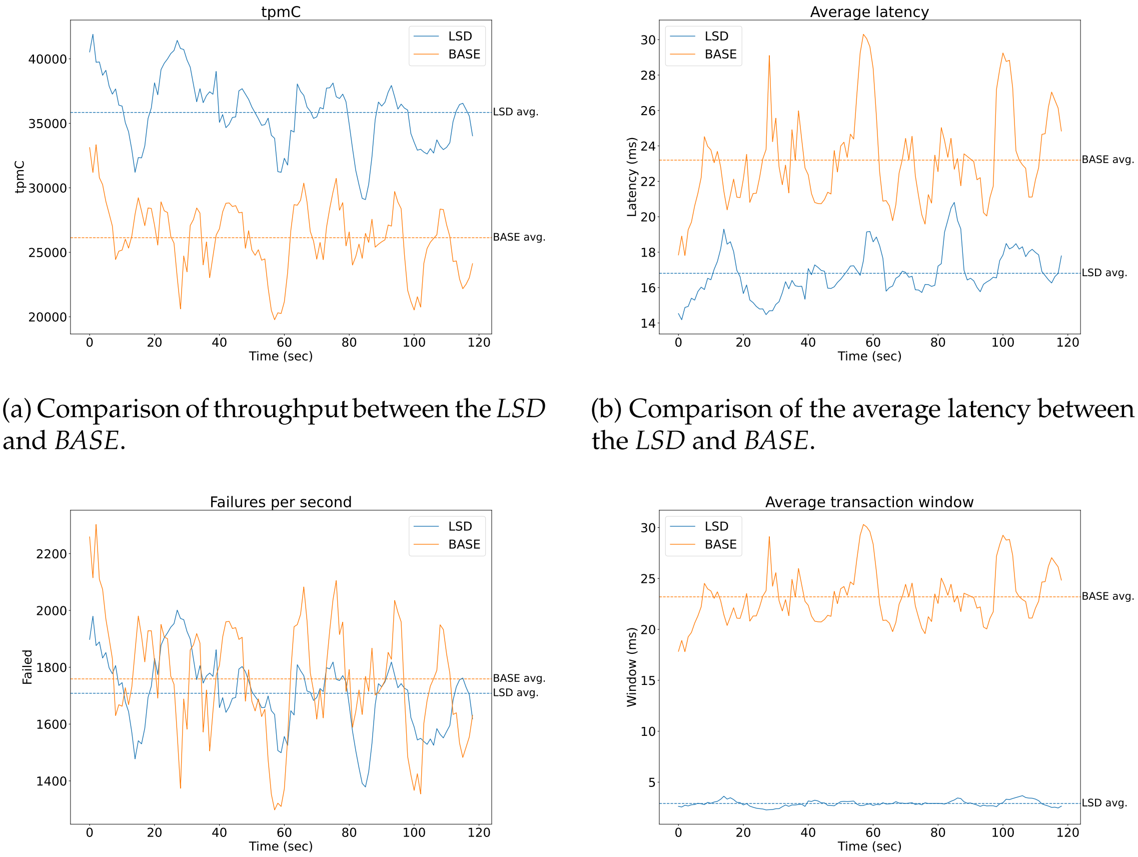
(c) Comparison of the number of transaction failures between the *LSD* and *BASE*.

(d) Comparison of the average transaction window between the *LSD* and *BASE*.

Figure 6.2: Throughput, latency, transaction failures and transaction window after executing only the *Payment* transaction with high contention. JDBC-LSD in represented by *LSD* and the standard JDBC is represented by *BASE* line.

to update the database, which will only be added as futures of the transaction when the branch is actually executing. This defeats the purpose of the LSD because in reality by having a future create another future we are actually adding time to the transaction window that did not need to be included.

All transactions When we execute all transactions defined in the TPC-C benchmark, we achieve a higher throughput (Figure 6.3) with *LSD* when compared to *BASE*. The number of transaction aborts (Figure 6.3c) that occur in this is case is very similar between *LSD* and *JDBC*, which may be attributed to the *Payment* transaction having some influence on the results. However, the latency (Figure 6.3b) and the transaction window (Figure 6.3d) are much lower for *LSD*, which explains why we achieve higher throughput with *LSD*.



(a) Comparison of throughput between the *LSD* and *BASE*.

(b) Comparison of the average latency between the *LSD* and *BASE*.

(c) Comparison of the number of transaction failures between the *LSD* and *BASE*.

(d) Comparison of the average transaction window between the *LSD* and *BASE*.

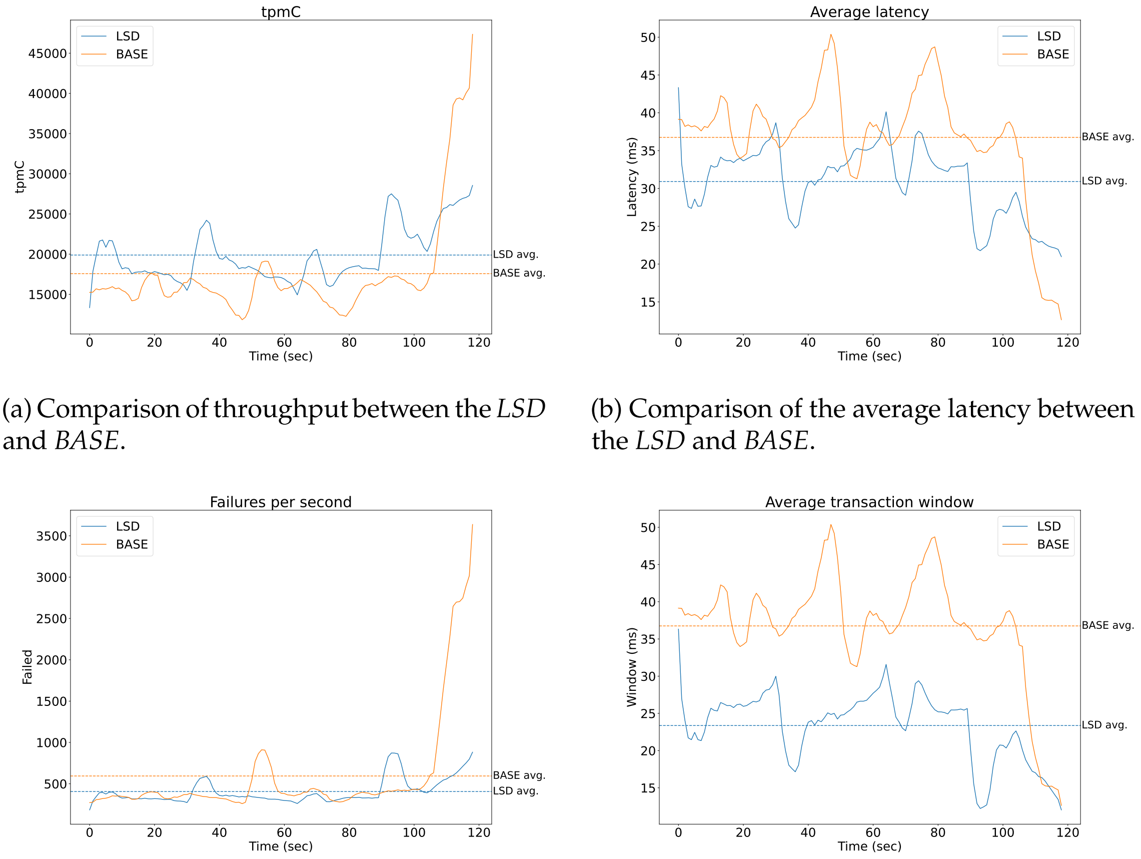
Figure 6.3: Throughput, latency, transaction failures and transaction window after executing all *TPC-C* transactions with high contention. JDBC-LSD in represented by *LSD* and the standard JDBC is represented by *BASE* line

6.2.2 Low contention

The low contention scenario is where we expected to not have differences between the LSD and JDBC. This is because, when contention is low, the chances of transactions conflict occurring are much lower.

***New-Order* transaction** Looking at results obtained for this transaction (Figure 6.4), we can see that we still achieve some improvements with LSD in terms of throughput (Figure 6.4a). The improvement comes from the fact that we are reducing the transaction window (Figure 6.4d), and consequently the latency (Figure 6.4b), which liberates time for more transaction to execute. As for the number of failed transactions (Figure 6.4c), the result is similar between *LSD* and *JDBC*.

***Payment* transaction** The execution of this transaction gives us worse results for *LSD* when compared to *BASE*. The result is similar to the high contention case, lower throughput



(a) Comparison of throughput between the *LSD* and *BASE*.

(b) Comparison of the average latency between the *LSD* and *BASE*.

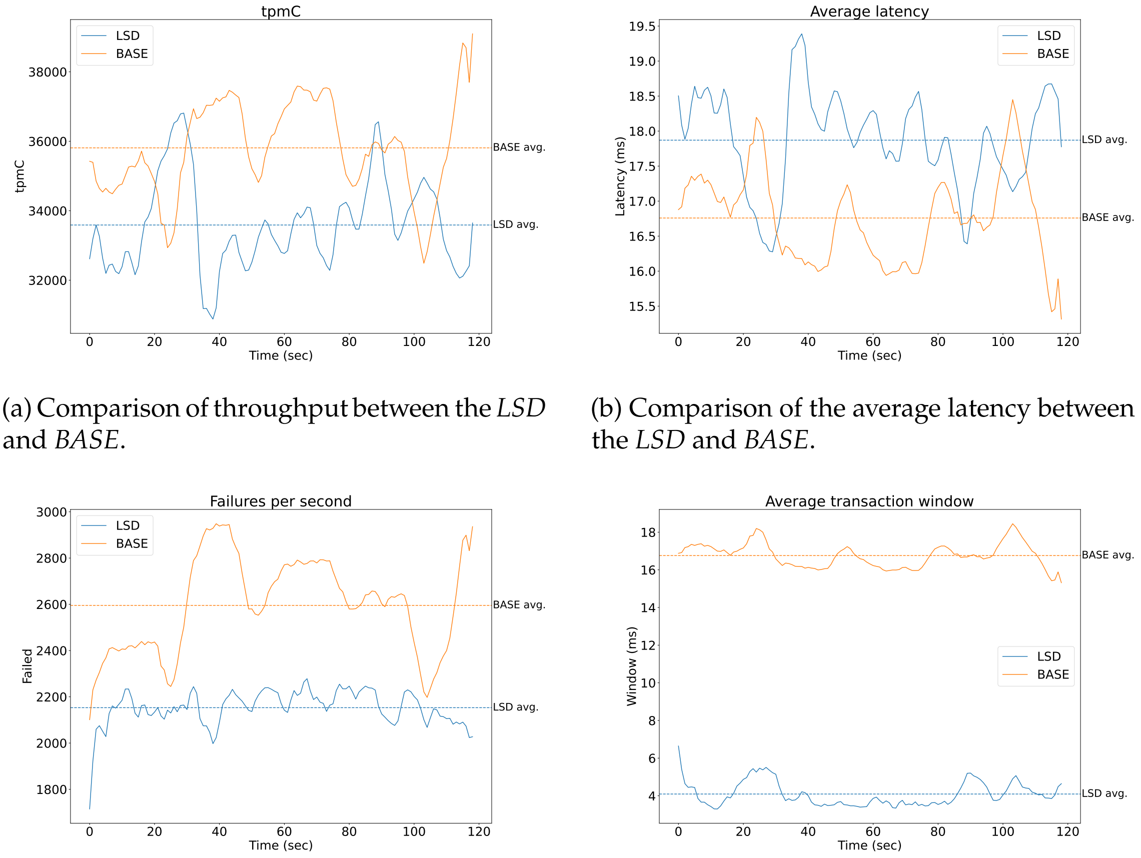
(c) Comparison of the number of transaction failures between the *LSD* and *BASE*.

(d) Comparison of the average transaction window between the *LSD* and *BASE*.

Figure 6.4: Throughput, latency, transaction failures and transaction window after executing only the *New-Order* transaction with low contention. JDBC-LSD in represented by *LSD* and the standard JDBC is represented by *BASE* line.

(Figure 6.5a) and higher latency (Figure 6.5b). There is also instability in the throughput and latency, which happens in both *LSD* and *BASE*, but with greater variance for *LSD*. This, paired with the observation made about the `FutureCondition` in the equivalent high contention scenario, likely indicates an issue with `FutureCondition` which needs further investigation. As for the failures and transaction window, the results we got are better for *LSD* than the *BASE*.

All transactions The execution of all transactions (Figure 6.6) yields, overall, better results for *LSD*, although not so prominent as was the case of high contention. Still, the throughput for *LSD* increased by roughly 20%, the latency for *LSD* decreased by roughly 10%, and the transaction window of *LSD* was cut down to roughly 30% of the original transaction window of *BASE*. As for the failed transactions (Figure 6.6c), the results between *LSD* and *BASE* is very similar, although *BASE* has some moments where a higher amount of transactions failed.



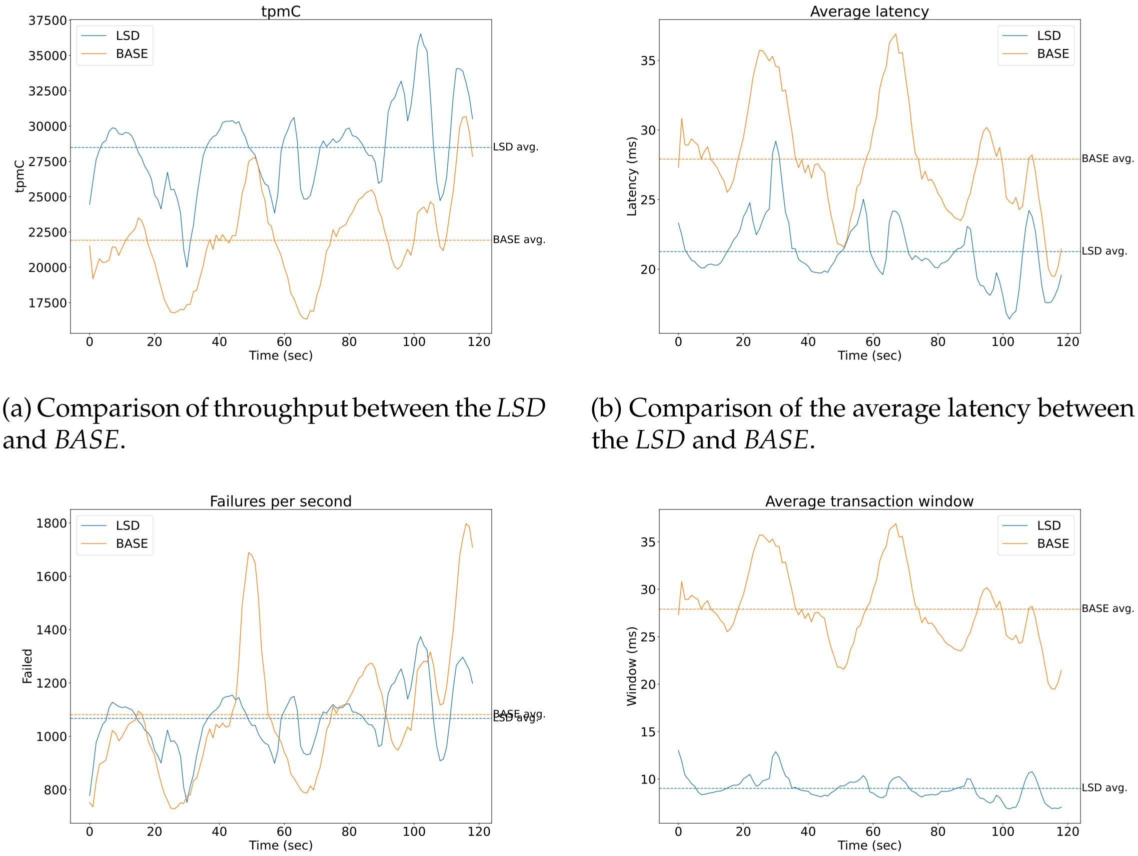
(a) Comparison of throughput between the *LSD* and *BASE*.

(b) Comparison of the average latency between the *LSD* and *BASE*.

(c) Comparison of the number of transaction failures between the *LSD* and *BASE*.

(d) Comparison of the average transaction window between the *LSD* and *BASE*.

Figure 6.5: Throughput, latency, transaction failures and transaction window after executing only the *Payment* transaction with low contention. JDBC-LSD in represented by *LSD* and the standard JDBC is represented by *BASE* line.



(a) Comparison of throughput between the *LSD* and *BASE*.

(b) Comparison of the average latency between the *LSD* and *BASE*.

(c) Comparison of the number of transaction failures between the *LSD* and *BASE*.

(d) Comparison of the average transaction window between the *LSD* and *BASE*.

Figure 6.6: Throughput, latency, transaction failures and transaction window after executing all *TPC-C* transactions with low contention. JDBC-LSD in represented by *LSD* and the standard JDBC is represented by *BASE* line.

7

Conclusions

In this chapter, we conclude this dissertation with final remarks and a summary of the contributions highlighted in the previous chapters. We also give hints upon what to build in the future for improving the results here achieved.

7.1 Summary and Conclusions

In the previous chapters, we have shown an implementation of the Lazy State Determination (LSD)-Application Programming Interface (API) that focuses on the client side. This implementation was shown to provide improvements on the throughput of a system by delaying operations. We have achieved this by defining a methodology, that got implemented as an extension of the Java Database Connectivity (JDBC) and, although being implemented as a JDBC extension, the methodology we defined could have been applied to other different database driver APIs. For the JDBC specific case, we have implemented a database driver that is compatible with all databases with a JDBC driver, providing great flexibility for integrating with many databases. As for the JDBC extension, this new API ended up being very similar to the original JDBC API, and as demonstrated in Section 5.1 by Listing 5.9, enables code conversion of a JDBC transaction to a JDBC-LSD transaction to be a simple task without much impact on the overall implementation.

Overall, we have shown that the proposed solution allows for a greater throughput in high contention scenarios and better, or similar, results in the low contention scenarios. This proves the derived solution to be an alternative when executing database transactions in a Relational Database Management System (RDBMS).

7.2 Future Work

The JDBC-LSD API leaves open some points that may be improved. Overall, the solution for branching code execution is not perfect, since it will only be evaluated at commit time. This is not ideal, since we may have any arbitrary amount of code inside a future condition branch and also forces any future statement that needs to be executed in a future condition branch to be created and executed inside the commit method. To improve this, we propose the evaluation of the condition, and its dependent futures, in a separate database connection and, the validation of the same condition again at the end of the transaction. The idea here is that we give greater flexibility to the programmer for branching code execution, while also enabling multiple statements to be executed inside that same branch.

Another factor that could leverage some optimization, comes from the fact that, when we execute the commit to the database, we know all the statements we need to execute. Knowing this, gives us the opportunity to optimize multiple statements to be batched into a single statement to the database, which would end up reducing the latency that is added between database communications. This could be done by grouping them into single statements that would share a common purpose, for instance, multiple inserts and update statements into a single batch statement. Another thing that comes from the same commit fact, would be to optimize the order of query execution as we know dependencies we have at commit time.

Finally, and leveraging the JDBC-LSD API here presented, the next path to follow would be to change the creation of futures to the database instead of the client. The JDBC-LSD API would then send the future operations to the database. These future operations would be stored in the database and when the JDBC-LSD API triggers a commit, the Concurrency Control (CC) of the database would then resolve all the futures. This strategy should be the one that more greatly reduces the window in which a transaction exposes it state to other transactions. This happens because we would be removing the client to database communication of the total transaction latency.

Bibliography

[1] *Akamai Online Retail Performance Report: Akamai.* URL: https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report (visited on 01/23/2022) (cit. on p. 1).

[2] H. Berenson et al. "A Critique of ANSI SQL Isolation Levels". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD '95. San Jose, California, USA: Association for Computing Machinery, 1995, pp. 1–10. ISBN: 0897917316. DOI: 10.1145/223784.223785 (cit. on pp. 5–7).

[3] B. Bhargava. "Concurrency control in database systems". In: *IEEE Transactions on Knowledge and Data Engineering* 11 (1 Jan. 1999), pp. 3–16. ISSN: 10414347. DOI: 10.1109/69.755610 (cit. on pp. 7, 10).

[4] B. F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152 (cit. on p. 58).

[5] K. P. Eswaran et al. "The Notions of Consistency and Predicate Locks in a Database System". In: *Commun. ACM* 19.11 (Nov. 1976), pp. 624–633. ISSN: 0001-0782. DOI: 10.1145/360363.360369 (cit. on p. 9).

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1558601902 (cit. on p. 9).

[7] H. T. Kung and J. T. Robinson. "On Optimistic Methods for Concurrency Control". In: *ACM Trans. Database Syst.* 6.2 (June 1981), pp. 213–226. ISSN: 0362-5915. DOI: 10.1145/319566.319567 (cit. on p. 10).

[8] H. Lim, M. Kaminsky, and D. G. Andersen. "Cicada: Dependably Fast Multi-Core In-Memory Transactions". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 21–35. ISBN: 9781450341974. DOI: 10.1145/3035918.3064015 (cit. on pp. 13, 21, 60).

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

[10] 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. 7).

[11] C. H. Papadimitriou. "The Serializability of Concurrent Database Updates". In: *J. ACM* 26.4 (Oct. 1979), pp. 631–653. ISSN: 0004-5411. DOI: 10.1145/322154.322158 (cit. on p. 5).

[12] G. Prasaad, A. Cheung, and D. Suciu. "Improving High Contention OLTP Performance via Transaction Scheduling". In: *CoRR* abs/1810.01997 (2018). arXiv: 1810.01997 (cit. on pp. 13, 14).

[13] D. Pritchett. "Base an acid alternative". In: *Queue* 6 (3 May 2008), pp. 48–55. ISSN: 15427730. DOI: 10.1145/1394127.1394128 (cit. on pp. 14, 15).

[14] *RocksDB | A persistent key-value store | RocksDB*. URL: https://rocksdb.org/ (visited on 01/23/2022) (cit. on p. 20).

[15] D. Shasha et al. "Transaction Chopping: Algorithms and Performance Studies". In: *ACM Trans. Database Syst.* 20.3 (Sept. 1995), pp. 325–363. ISSN: 0362-5915. DOI: 10.1145/211414.211427 (cit. on p. 15).

[16] Y. Sheng et al. "Scheduling OLTP Transactions via Learned Abort Prediction". In: *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. aiDM '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019. ISBN: 9781450368025. DOI: 10.1145/3329859.3329871 (cit. on p. 15).

[17] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database system concepts*. 6th ed. New York: McGraw-Hill, 1986. ISBN: 0070447527. URL: http://www.db-book.com/ (cit. on pp. 4, 6, 7).

[18] E. Subtil. "Lazy State Determination for SQL Databases". MA thesis. NOVA School of Science and Technology - NOVA University Lisbon, 2021 (cit. on pp. 2, 4, 20).

[19] *TPC History*. URL: http://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp (visited on 01/23/2022) (cit. on pp. 2, 55–57).

[20] *TPC History*. URL: http://tpc.org/information/about/history5.asp (visited on 01/23/2022) (cit. on p. 55).

[21] S. Tu et al. "Speedy Transactions in Multicore In-Memory Databases". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 18–32. ISBN: 9781450323888. DOI: 10.1145/2517349.2522713 (cit. on pp. 11, 12, 21, 60).

[22] T. M. Vale et al. "Lazy State Determination: More concurrency for contending linearizable transactions". In: *arXiv preprint arXiv:2007.09733* (2020) (cit. on pp. 4, 20).

[23] T. M. do Vale. "Executing requests concurrently in state machine replication". PhD thesis. NOVA School of Science and Technology - NOVA University Lisbon, 2019. URL: https://run.unl.pt/handle/10362/71218 (cit. on pp. 2, 4, 16, 20).

[24] Y. Wu et al. "An Empirical Evaluation of In-Memory Multi-Version Concurrency Control". In: *Proc. VLDB Endow.* 10.7 (Mar. 2017), pp. 781–792. ISSN: 2150-8097. DOI: 10.14778/3067421.3067427 (cit. on p. 12).

[25] C. Xie et al. "High-performance ACID via modular concurrency control". In: *SOSP 2015 - Proceedings of the 25th ACM Symposium on Operating Systems Principles* (Oct. 2015), pp. 279–294. DOI: 10.1145/2815400.2815430 (cit. on p. 15).

[26] C. Xie et al. "Salt: Combining ACID and BASE in a Distributed Database". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 495–509. ISBN: 9781931971164 (cit. on pp. 14, 15).

[27] 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. 12, 21, 60).

A

The TPC-C Benchmark

Transaction Processing Performance Council (TPC) is an industry leading organization with regard to Online Transaction Processing (OLTP) benchmarks. In the following sections, we describe the TPC Benchmark C (TPC-C) and how its parameters may influence test execution and generate different scenarios. We also describe a more modern approach on benchmarking databases with Yahoo! Cloud Serving Benchmark (YCSB).

A.1 Brief History

It was in the 1980s that the industry began racing towards accelerating automation in our daily lives, leading them to rely on even more in databases performance. The first application in focus was the Automated Teller Machine (ATM) [20], but automation ended up reaching many industry segments. These industries started to rely on OLTP, stimulating database vendors competition. To prove performance of their systems, database vendors began making claims over, the now defunct, TP1 benchmark. This benchmark, however, had flaws over the assumptions it made and could inflate performance results [20]. It is over this context that the benchmarks from Transaction Processing Performance Council (TPC) have come to live. Over the years many benchmarks have been proposed, but to this date, the most widely used is TPC Benchmark C (TPC-C).

A.2 The Company

The TPC-C benchmark models a multi-warehouse operation, also known as the Company [19]. The company portrayed by this benchmark is a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. Each warehouse is responsible for covering 10 distinct districts with each district serving 3000 customers. All the warehouses are required to maintain stocks for the 100,000 items that are sold by the Company. The Company structure is present in Figure A.1.

For representing such environment, the database is divided into the following entities (Figure A.2):

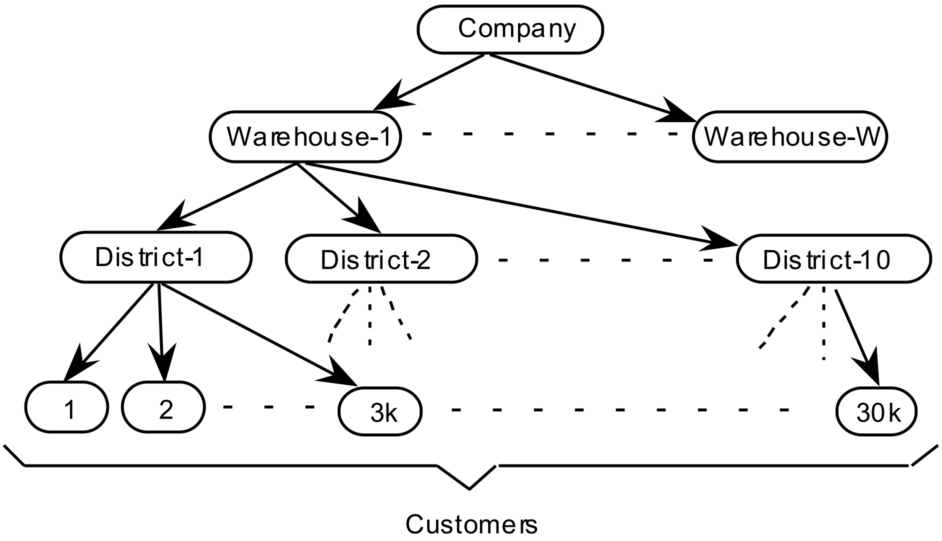


Figure A.1: The Company business structure. Source TPC-C specification [19].

Item — Holds information over the 100,000 products sold by the Company.

Warehouse — Holds information over the warehouses the Company own. Represented by W in the Figure A.2.

Stock — Holds information over the stock of the products sold by the Company. It is of size $W * 100,000$.

District — Holds information over the different districts where the Company is present. It is of size $W * 10$.

Customer — Holds information over the customers the Company has. It is of size $W * 30,000$.

History — Holds information over the purchases of each customer. It is of size $W * 30,000$ but is subject to variations depending on the parameters of execution.

Order — Holds information over the orders sent to the Company. It is of size $W * 30,000$ but is subject to variations depending on the parameters of execution.

New-Order — Holds information over new orders sent to the Company. It starts with size $W * 9,000$.

Order-Line — Holds information over each line of the orders sent to the Company. It is of size $W * 300,000$ but is subject to variations.

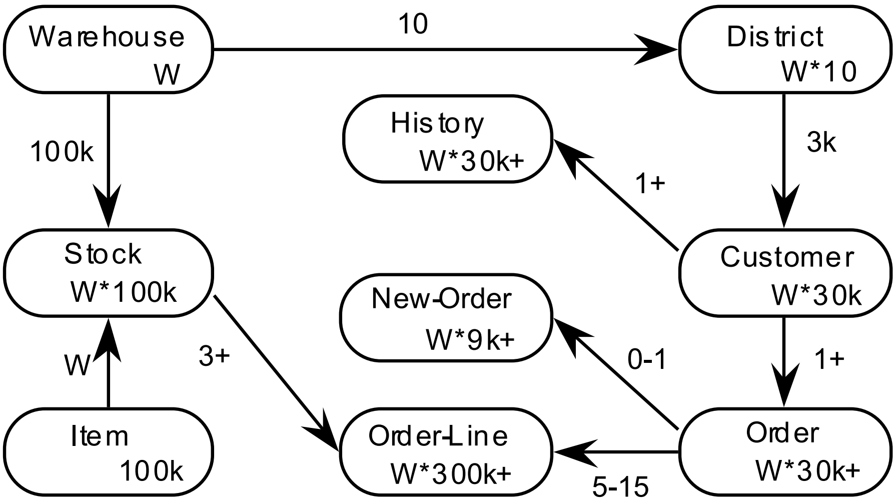


Figure A.2: The Company database entities. Source TPC-C specification [19].

A.3 The Benchmark

Leveraging the structure presented in the previous section, TPC-C [19] defines five different types of transactions:

New-Order — Simulates the entering of a new order in the database through a single transaction. It forms the backbone of the benchmark and has the most strict requirements in terms of latency. It is a read-write transaction that has a high frequency of execution and where about 1% of the transactions will fail in order to test transaction rollbacks.

Payment — It updates the customer balance and reflects the payment value to the district and warehouse statistics. It is a read-write transaction with high frequency of execution and strict latency requirements.

Order-Status — Queries the status of a customer later order. It is a read-only transaction that has a low frequency of execution and low latency requirements.

Delivery — This transaction processes batches of 10 yet to be delivered orders. It is a read-write transaction that has a low frequency of execution and relaxed response time requirements.

Stock-level — This transaction calculates the number of items recently sold that have a stock level bellow a certain threshold. It is a read-only transaction that has a low frequency of execution, has a relaxed response time and consistency requirements.

The number of warehouses (*W*) is a key configurable parameter that determines the scale of the benchmark. Increasing it, increases the number of warehouses, the number of concurrent clients and consequently the data set size.

Results of the benchmark are measured in *tpmC*, which represents transactions per minute.

A.4 YCSB, a Modern Alternative

Cloud computing has gained a lot of traction in the last few years, with many cloud providers appearing to face the demand. The ubiquity of cloud systems bring, not only new of applications, but also new challenges for traditional Database Management System (DBMS). Due to this, today's systems began to rely on database models that differ from the traditional relational data structure present in Relational Database Management System (RDBMS). Models such as *Key-Value stores*, *Document stores* and *Column oriented stores* are quite common today. These models were born from specific needs, and help today's applications perform better on specific workloads. Examples of such database models are Cassandra [1], CouchDB [2], MongoDB [3] and Voldemort [4], and fall under the group of NoSQL databases.

Although the TPC-C benchmark has a lot of similarities with workloads that run in today's systems, it manages to only capture a sub-set, and it may not represent the ways that DBMS are used to this date. It is with this in mind that the Yahoo! Cloud Serving Benchmark (YCSB) [4] is proposed by Cooper et al. Their goal was to create a standard benchmark and benchmarking framework to assist in the evaluation process of different database systems. The framework enables fine tune of the standard workloads in order to match specific system requirements and is also extensible by allowing new DBMS and workloads to be added.

A.4.1 Workloads

The standard benchmarks present in Yahoo! Cloud Serving Benchmark (YCSB) aims to model a *serving* system. Serving systems are systems that provide online read and write access to data. This is modeled from users interaction with websites, that is, while a user waits for a web page to load, a server carries reads and writes to a database in order to construct and deliver that web page. Even though a particular model is defined, the goal of YCSB is to, given an analysis made to several systems, provide standard workloads that most closely capture real world systems, and not a sub-set as is the case of TPC-C.

The workloads present in the YCSB all derive from a basic application. In this application there is a table of records, each with a parametrized number of fields. The value of each field is a random ASCII string with its size being parametrized, and each record is identified by a primary key. Operations executed against the database are chosen

[1] https://cassandra.apache.org/_/index.html

[2] https://couchdb.apache.org/

[3] https://www.mongodb.com/

[4] https://www.project-voldemort.com/

Table A.1: Standard workloads present in the YCSB.

Workload	Operations	Record selection
A - Update heavy	Read: 50% Update: 50%	Zipfian
B - Read heavy	Read: 95% Update: 5%	Zipfian
C - Read only	Read: 100%	Zipfian
D - Read latest	Read: 95% Insert: 5%	Latest
E - Short ranges	Scan: 95% Insert: 5%	Zipfian/Uniform

randomly, and are inserts, updates, reads and scans. Random choices over records to read and write are made using the following distributions:

Uniform — all records have equal chance of being chosen;

Zipfian — some records will be extremely popular while some others are not;

Latest — most recent records are more likely to be chosen; and

Multinomial — specific probabilities of read and write operations on an item.

Table A.1 gives an overview of the standard workloads present in the YCSB. An example for each workload is listed below:

Workload A — A update heavy test in which reads and updates are divided equally, An example of an application would be a session storage recording actions in a user session;

Workload B — A read focused test with very few updates. An example of such workload would be photo tagging, in which the addition of a tag would be an update but most operations are for reading the tags;

Workload C — A read only test and, as an example, models some application that caches user profiles;

Workload D — A test that focuses on reading the latest items with very few inserts. An example of such workload would be an application in which users post status updates, however, the bulk of operations is on reading the latest status;

Workload E — A test that focuses on reading the latest items with very few inserts. An example of such workload would be an application in which most users are reading the latest status and very few are posting status updates; and

Workload F — A test that focuses on scans/ranges of data with very few inserts. This could be the case of threaded conversations, where each scan is for a specific thread.

Unlike TPC-C, which models a specific workload of a subset of applications, YCSB is more flexible and allows for a more generalized approach when looking at real world performance. In a sense, these two benchmarks add to each other and using TPC-C and YCSB may be beneficial when gauging DBMS performance. This is certainly the case of the literature [8, 21, 27], which in some cases uses both TPC-C and YCSB in their benchmarks.

2022 Empowering a Relational Database with LSD: Lazy State Determination Thales Parreira

NOVA SCHOOL OF SCIENCE & TECHNOLOGY