NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

EDUARDO BEZERRA SUBTIL
Bachelor of Science

# LAZY STATE DETERMINATION FOR SQL DATABASES

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2021

DEPARTMENT OF
COMPUTER SCIENCE

# LAZY STATE DETERMINATION FOR SQL DATABASES

**EDUARDO BEZERRA SUBTIL**
Bachelor of Science

**Adviser:** João Manuel dos Santos Lourenço
*Associate Professor, NOVA University Lisbon*

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2021

**Lazy State Determination  for SQL databases**

*Gaudeamus igitur iuvenes dum sumus.*
*Alea iacta est.*

# Acknowledgements

**TRADUZIR PARA INGLÊS!!!**

Acaba agora uma etapa muito difícil e importante, e estará prestes a iniciar-se um novo capítulo da minha vida. Ao longo deste percurso académico, especialmente neste último ano, existiram muitos obstáculos, muitos desses impossíveis de ultrapassar não fosse pelo suporte de diversas pessoas e entidades. Passo a mencionar todos estes que estiveram presentes neste caminho.

Ao Professor João Lourenço, obrigado por ser um orientador muito exigente e critíco, sempre a puxar o melhor de mim, mas também por estar sempre disponível. A sua porta estava sempre aberta quando mais necessitava, independentemente do tipo de problemas que pudesse ter, mesmo se já estivéssemos nas longas horas da noite. Nunca hesitou em dar-me direção quando me sentia perdido, quer na escrita, quer no processo de investigação.

Para a nossa bela instituição, a Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, obrigado por todo o conhecimento, educação e oportunidades que me foram facultadas nestes últimos anos.

Para a Fundação para a Ciência e Tecnologia, obrigado pela Bolsa de Investigação que me proporcionaram ao longo deste projeto.

Para os meus colegas do projeto *HiPSTr* David Carpinteiro, Tiago Vale e Ricardo Dias, obrigado por todos os momentos que despenderam dos vossos dias preenchidos para me ajudar. Sempre que nos encontrávamos, recebia ideias novas que impulsionavam o progresso deste projeto.

Para os meus colegas de curso de MIEI, obrigado por me terem ajudado com a transição do ambiente Secundário para o ambiente Universitário. Não poderia ter pedido um melhor acolhimento e integração no que era na altura um ambiente completamente novo e exótico, cheio de possibilidades.

Para os meus companheiros de tasca do *Space Poker Imaginário*, de onde destaco os membros André "Mármore" Augusto, Beatriz Rebelo, Bernardo "Boris" Baldaia, Gonçalo Fazenda, Henrique "do Mal" Silva, Joana "Ju" Martins, João "Dini" Sampaio, Marta Carlos, Nuno Morais, Ricardo Leitão, Tiago Gonçalves e Tomás Alagoa, obrigado por todos os momentos de parvoíce e por todas as conversas pseudo-intelectuais que vamos tendo a cada lua cheia. Este período não teria sido o mesmo sem vocês, e espero que venham muitos mais no futuro.

Para a Sara Simões, obrigado pela tua presença permanente, pelo teu suporte incondicional e por seres a minha inspiração. Não poderia ter ninguém melhor ao meu lado.

Finalmente, mas igualmente importante, para os meus pais, obrigado por tudo o que fizeram e ainda fazem por mim, agora e sempre.

This work was partially supported by the Foundation for Science and Technology and European Regional Development Fund, under the Research Project *HiPSTr: High-Performance Software Transactions* (LISBOA-01-0145-FEDER-032456 / PTDC/CCI-COM/-32456/2017).

*“Audaces Fortuna Adiuvat” (Virgilius)*

# Abstract

Transactional systems have seen various efforts to increase their throughput, mainly by making use of parallelism and efficient Concurrency Control techniques. Most approaches optimize the systems' behaviour when under high contention.

In this work, we strive towards reducing the system's overall contention through Lazy State Determination (LSD). LSD is a new transactional API that leverages on futures to delay the accesses to the Database as much as possible, reducing the amount of time that transactions require to operate under isolation and, thus, reducing the contention window.

LSD was shown to be a promising solution for Key-Value Stores. Now, our focus turns to Relational Database Management Systems, as we attempt to implement and evaluate LSD in this new setting. This implementation was done through a custom JDBC driver to minimize required modifications to any external platform.

Results show that the reduction of the contention window effectively improves the success rate of transactional applications. However, our current implementation exhibits some performance issues that must be further investigated and addressed.

**Keywords:** Concurrency Control, On-Line Transaction Processing, Relational Systems, Lazy State Determination, High Contention Environments, Java Database Connectivity

# Resumo

Os sistemas transacionais têm sido alvo de esforços variados para aumentar a sua velocidade de processamento, principalmente através de paralelismo e de técnicas de controlo de concorrência mais eficazes. A maior parte das soluções propostas visam a otimização do comportamento destes sistemas em ambientes de elevada contenção.

Neste trabalho, nós iremos reduzir a contenção no sistema recorrendo ao Lazy State Determination (LSD). O LSD é uma nova API transacional que promove a utilização de futuros para adiar o máximo os acessos à Base de Dados, reduzindo assim o tempo que cada transação requer para executar em isolamento e, por consequência, reduzindo também a janela de contenção.

O LSD tem-se mostrado uma solução promissora para bases de dados Chave-Valor. O nosso foco foi agora redirecionado para Sistemas de Gestão de Bases de Dados Relacionais, com uma tentativa de implementação e avaliação do LSD neste novo contexto. Este objetivo foi concretizado através da implementação de um controlador JDBC para minimizar quaisquer alterações a plataformas externas.

Os resultados mostram que a redução da janela de contenção efetivamente melhora a taxa de sucesso de aplicações transacionais. No entanto, a nossa implementação atual tem alguns problemas de desempenho que necessitam de ser investigados e endereçados.

**Palavras-chave:** Controlo de Concorrência, Processamento de Transações Em-Linha, Sistemas Relacionais, LSD, Ambientes de Alta Contenção, JDBC

# Contents

# List of Figures

# List of Tables

# List of Listings

# Glossary

This document is incomplete. The external file associated with the glossary ‘main’ (which should be called `template.gls`) hasn’t been created.

Check the contents of the file `template.glo`. If it’s empty, that means you haven’t indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can’t be generated. If the file isn’t empty, the document build process hasn’t been completed.

If you don’t want this glossary, add `nomain` to your package option list when you load `glossaries-extra.sty`. For example:

```
\usepackage[nomain]{glossaries-extra}
```

Try one of the following:

- Add `automake` to your package option list when you load `glossaries-extra.sty`. For example:

  ```
  \usepackage[automake]{glossaries-extra}
  ```

- Run the external (Lua) application:

  ```
  makeglossaries-lite.lua "template"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "template"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.

# Acronyms

This document is incomplete. The external file associated with the glossary 'acronym' (which should be called `template.acr`) hasn't been created.

Check the contents of the file `template.acn`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

Try one of the following:

- Add `automake` to your package option list when you load `glossaries-extra.sty`. For example:

  ```
  \usepackage[automake]{glossaries-extra}
  ```

- Run the external (Lua) application:

  ```
  makeglossaries-lite.lua "template"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "template"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.

# Symbols

This document is incomplete. The external file associated with the glossary 'symbols' (which should be called `template.sls`) hasn't been created.

This has probably happened because there are no entries defined in this glossary. Did you forget to use `type=symbols` when you defined your entries? If you tried to load entries into this glossary with `\loadglsentries` did you remember to use `[symbols]` as the optional argument? If you did, check that the definitions in the file you loaded all had the type set to `\glsdefaulttype`.

This message will be removed once the problem has been fixed.

# 1 Introduction

In this chapter we will present our motivations and the approaches that need to be undertaken to address the problems we are trying to solve, as well as further insight upon the content that will be addressed later in this report.

## 1.1 Context and Motivation

The evolution of Computing, as we see it today, is moving towards *Hardware Resource Pooling* [47], as we have witnessed with the growth of *Distributed Systems*, *Cloud Computing* [2], and the *Internet of Things* [4], due to the performance of individual components becoming harder and harder to improve. Ever since, we have seen the rise of *multicore* and *multiprocessor* systems and, with it, the growing importance of concurrency and parallelism to exploit efficient use of these systems.

In Database Management Systems (DBMSs), if every operation requested were to be run serially, users would quickly find themselves waiting for a long time before receiving any response from the system, many of those times waiting for too long, due to some operations being time-sensitive.

To improve their performance, these databases employ parallelism, allowing the database to run as many transactions as possible at the same time. However, uncontrolled concurrency can quickly destroy data consistency, rendering these systems useless. Thus, Concurrency Control (CC) techniques were developed, providing different degrees of data consistency and allowing these systems to better serve their users.

With the growing number of users and the increasing ubiquity of services that require data storage, databases have suffered scaling problems due to the increase of *data contentions*, as solutions currently in use for CC suffer performance drops when having to deal with these data conflicts:

- *Pessimistic solutions*, which are based on locks, impede the progress of transactions that are unable to acquire proper access to the items they wish to view if the items in question are currently being accessed by another transaction. Such behaviour may lead to *deadlock* situations that must then be resolved, normally by aborting one of the culprit transactions.

- *Optimistic solutions*, on the other hand, always allow uncontrolled access to their data items, but then lose valuable processing time later when validating all the operations executed and rolling back transactions with conflicting operations.

## 1.2 Problem and Goals

In *Transactional Systems*, concurrency and parallelism aim to increase the throughput of the system by decreasing the amount of time required for each transaction to be committed. This is not possible without strict control over the *order of execution*, as data consistency must be maintained at any cost. The techniques that attempt to maintain said consistency in parallel environments may be classified under the CC umbrella.

There have been many propositions and solutions to improve the scalability of these systems over the years, but the core of our work will be with the Lazy State Determination (LSD) [45] technique, which proposes the *lazy evaluation and computing* of these transactions to reduce the chance of *data conflicts* between them.

So far, LSD, when used in Key-Value Stores, shows better performance, with higher throughput, lower average response time of the system and lower conflict rates, than the related solutions when operating in high contention environments. Now, our focus of research will be the implementation of LSD in Relational Database Management System (RDBMS) environments.

## 1.3 Approach and Contributions

Most of the work proposed so far has been to reduce the impact of data contention, through optimization of existing CC protocols or the implementation of new ones.

LSD, instead, proposes the reduction of the data contentions by reducing the amount of time transactions execute in isolation. In this way, the system can further increase the concurrent execution of transactions, as these performance roadblocks now happen less frequently.

In order to enable LSD for RDBMSs, we must address any compatibility issues that may appear between LSD and SQL.

Once these problems have been resolved, a new prototype driver must be developed, followed by testing and evaluation under different workloads. For these tests we will make use of the TPC Benchmark C (TPC-C) [40] standard benchmark.

More succinctly, our contributions will be:

- An in-depth evaluation of the *State of the Art*.
- A LSD Application Programming Interface (API) for RDBMS environments.

- An implementation of a driver which enables interaction with RDBMSs through LSD.
- A correctness and performance analysis comparing our newly-implemented solution against regular ODBC drivers.

## 1.4 Document Layout

Following is a brief presentation for the following chapters in this report:

- In Chapter 2, we will explore basic concepts and general ideas that have been previously worked on in this field of research and offer insights as to what advantages our different perspective might bring.
- In Chapter 3, we will discuss how we have implemented our prototype driver and discuss the challenges that have arisen.
- In Chapter 4, we offer our analysis and discuss the results we have obtained with our new LSD-based ODBC driver.
- Finally, in Chapter 5, we review the work that we have done and how we can improve in the future.

# 2 Concepts and Related Work

In this chapter, we will explore the basic concepts of *Asynchronous Programming* (Sec. 2.1) and *Transactional Systems* (Sec. 2.2), followed by a brief study of how interaction with a DBMS is made possible (Sec. 2.3) and, finally, a study of the *State-of-the-Art* and Related Work (Sec. 2.4).

## 2.1 Basic Concepts of Asynchronous Programming

### 2.1.1 Concurrency and Parallelism

*Concurrency* [12, 26] can be defined as the ability to execute *(some or all)* steps of an algorithm out of order and achieve the same output as if they were executed sequentially. More formally, it is a property that refers to the decomposability of an algorithm into units that are order-independent or partially-ordered.

*Parallelism* is the simultaneous execution of processes or computations. It is a property of *multi-processor* or *multi-core systems* that drive higher performance and throughput.

Both parallelism and concurrency are concepts that are usually mentioned together due to their definitions being intertwined, but they are distinct concepts. It is possible to have concurrency without parallelism but there can be no parallelism without concurrency.

In the context of this essay, we will be more interested in discussing concurrency and its many nuances and obstacles, with the ultimate goal of being able to process transactions in parallel reliably and consistently.

### 2.1.2 Asynchronous Programming

*Asynchronous Programming* is a programming paradigm that focuses in maximizing application performance through the use of *multiprocessors* by splitting the work into different parts that can be run *concurrently* and independently of each other[28].

Many techniques for Asynchronous Programming have been developed and explored. Following is an outline of some of these techniques.

#### 2.1.2.1 Coroutines

*Coroutines* [24] are computer program components that allow for non-preemptive concurrent execution of other components, such as subroutines. According to Knuth, the "coroutine" term was coined by Melvin Conway when he first implemented them in an Assembly program.

When compared to threads, coroutines are simpler to implement since they don't need synchronization primitive types such as semaphores, for example, to guard against critical sections. However, coroutines do not provide parallelism due to their cooperative multitasking nature.

#### 2.1.2.2 Event-Driven Programming

*Event-driven programming* [7] is a paradigm that bases the flow of the program's execution upon events (user actions, sensors' outputs, received network messages, etc). It is widely used in Graphical User Interface (GUI) development, distributed systems and anywhere that is centred on performing certain actions in response to inputs, such as device drivers.

In Event-Driven applications, there is a main loop listening for events, which triggers callback functions when one of these events is detected. For concurrency, this main loop can be a dispatcher, passing the callback function to some thread that can execute it in parallel.

#### 2.1.2.3 Actor Models

Highly used in Event-driven applications and Distributed Systems, *Actors* [19] are objects that are capable of receiving, processing and creating messages. Actors listen for "events", specific messages, and then trigger an appropriate response based on the messages they have received. Each actor keeps its own internal state, can spawn and communicate with other actors and can execute their functions anywhere, as long as all the system's elements are found in the same network.

Their main advantages for a system are the ease of scalability, heterogeneity and simplification of communications and executions.

#### 2.1.2.4 Futures

*Futures* [17], also known as *Promises*, is a technique that provides lazy or delayed evaluation of expressions by deferring or executing in parallel the requested computation until absolutely necessary, instead creating an object that represents the future computation's output. More simply, they provide the program with an "empty box" whose contents will eventually arrive. As such, the program can then continue execution up until the point where it requires the box's contents. When it does, if the box's contents already arrived, execution can proceed, or if they did not arrive, the program can either wait for them (by blocking execution) or continue some other task until notified of their arrival (non-blocking execution).

Several advantages can be found when building applications with futures. It enables high degrees of parallelism, as the main process can defer the computation of an expression to a thread and only checking and possibly waiting for its conclusion when desired. Additionally, computations that end up not being used by the main process can be discarded, reducing wasted Central Processing Unit (CPU) cycles, which may also have a positive impact in the system.

## 2.2 Transactional Systems

*Transactional Systems* are database systems that rely on *transactions* as units of work. In this section, some concepts about transactional systems are given, increasing our focus of discussion on DBMSs, particularly RDBMSs, and on Online Transaction Processing (OLTP).

### 2.2.1 Introduction

Transactional systems gain their name thanks to their reliance on transactions, which are units of work composed by one or more instructions. These units are then treated independently of one another by the system. They also represent a state change.

Transactions have four important properties, known as Atomicity, Consistency, Isolation, Durability (ACID) [18]:

- *Atomicity*, either a transaction is fully executed or not at all.
- *Consistency*, if a transaction finds the database in a consistent state (which can be guaranteed by the system through a set of invariants), then it must leave the database in a consistent state after its execution.
- *Isolation*, concurrent transactions must execute independently of each other. This means that concurrent transactions must behave as if they were being executed serially. This is the property we are most concerned about in this study.
- *Durability*, after a successful transaction execution, its changes must be permanent in the database.

Database systems using the “Relational Data” model are defined as RDBMSs. Many of these systems can be interacted with through the use of SQL [29] (more about it on Sec. 2.3), a query language made to exploit the “Relational Data” model. These systems are the focus of our investigation.

### 2.2.2 Isolation Levels

As mentioned previously, maintaining isolation while maximizing concurrency is the main focus of our work. SQL (and Berenson *et al.* [3]) currently defines four isolation levels (each stronger than the previous):

- *Read Uncommitted*, where changes made by other running transactions are visible. Virtually, there isn't any isolation here.
- *Read Committed*, where transactions can only see data that has been confirmed as committed.
- *Repeatable Read*, where transactions are guaranteed to see the same result if the same read operation is performed repeatedly within the same transaction.
- *Serializable*, where the output of a set of concurrent transactions' is guaranteed to be the same as a serial execution of the same set of transactions. This is the strongest and most complex isolation level.

Each isolation level has its strength measured by the number of anomalies by which they are vulnerable. "Repeatable Read" is weaker than "Serializable" because it is vulnerable to *Phantom Reads*, which occurs when, in the same transaction, two identical queries return two sets of rows where one of them is a subset of the other.

"Read Committed" is weaker than "Repeatable Read" because, in addition to *Phantom Reads*, it is vulnerable to both *Lost Updates*, where two transactions attempt to update the same row and column, both being unable to see the other's changes, and *Non-repeatable reads*, which happens when, in a transaction, two identical queries receive different sets of rows whose values have changed. "Read Uncommitted" is vulnerable to all the above and *Dirty Reads*, where a transaction can read data that won't be committed.

Table 2.1 depicts the relation between each isolation level and its anomalies, with (✓) representing the possibility of the anomaly happening and (✗) representing the contrary.

Table 2.1: Anomalies per Isolation Levels, as per Berenson *et al.* [3]

| | **Dirty Reads** | **Lost Updates** | **Non-repeatable reads** | **Phantom Reads** |
|---|---|---|---|---|
| **RU** | ✓ | ✓ | ✓ | ✓ |
| **RC** | ✗ | ✓ | ✓ | ✓ |
| **RR** | ✗ | ✗ | ✗ | ✓ |
| **SR** | ✗ | ✗ | ✗ | ✗ |

*RU: Read Uncommitted, RC: Read Committed, RR: Repeatable Read, SR: Serializable*

### 2.2.3 Concurrency Control Techniques

Many techniques have been developed over the years of research on concurrency control, each having their advantages and disadvantages. In this section, we will go over some of the most relevant methods employed today.

#### 2.2.3.1 Two-Phase Locking

Locks are an object which are associated with a shared resource (in our case, data items). Transactions wishing to perform certain operations upon any data item must gain access to the item's respective lock before doing so.

There are two types of locks:

- *Read-lock (or shared lock)*, which must be acquired before reading the associated data item. It is "shared" because, for each row, many transactions can hold one of these locks for the same row.

- *Write-lock (or exclusive lock)*, which must be acquired before writing changes to the data item, such as inserting, modifying or deleting it. It's "exclusive" in the sense that, for each row, only one transaction at a time can hold this type of lock. This lock blocks all other operations by other transactions and can only be acquired after all read-locks for the row have been relinquished.

One of the techniques developed that makes use of locks is Two-Phase Locking (2PL). As described by Eswaran *et al.* [15], it is a mechanism for concurrency control based on locks that ensures serializability. Each transaction, during its execution, acquires an increasing number of locks *(Growing Phase)* until it first releases a lock, upon which time it can only relinquish its held locks and cannot acquire new ones *(Shrinking Phase)*. When transactions acquire locks in this fashion, changes made to the database are considered serializable because only a single transaction can make changes per row at any given moment of time. In other words, all changes can be seen as a single timeline of events.

Methods relying on locks are vulnerable to *Deadlocks*, which occur when two concurrent transactions are waiting to acquire locks the other is holding. There are two ways to deal with deadlocks: either through detection or prevention. To prevent deadlocks, the system tests both involved transactions. This test consists in checking if the system will block if the incoming transaction is allowed to wait. If the test is passed, then the transaction may wait, otherwise one of them is aborted. The system can be characterized as *preemptive* if an incoming transaction immediately requests access to the CPU, making older transactions block and *non-preemptive* if an incoming transaction is forced to wait for older transactions [39].

Another technique for prevention is *predeclaration*, where transactions must obtain the set of locks that they require before executing. This completely prevents deadlocks, but leads to the starvation of transactions that have a relatively broad lock set, as the chance that they may be blocked increases.

With deadlock detection, transactions are allowed to block in an uncontrolled manner. The system only intervenes when a deadlock is detected, with it being possible thanks to a *Waits-for* graph, where nodes represent transactions and edges represent the "Waiting" relation. When the graph develops a cycle it means a deadlock has been found, and the system will choose to abort one of them, with this decision taking into account the progress made and the amount of used resources. Predeclaration can also be used here, with all of its already stated downsides.

Another alternative is using timeouts: a transaction has a maximum Time-To-Live (TTL). If this TTL is exceeded, the transaction is considered in a deadlock and is aborted.

#### 2.2.3.2 Optimistic Concurrency Control

Another method for concurrency control is to allow for uncontrolled access, only checking if any incoming changes have conflicts. This is the Optimistic Concurrency Control (OCC) approach, which is further elaborated on by Kung *et al.* [25].

The process can be structured down into three phases:

- *Read* phase, where transactions copy values from the required data items into their private workspaces.

- *Write* phase, optional, where transactions update the values in their private workspace as per their instructions.

- *Validation* phase, where the system checks if the incoming changes have conflicts with other transactions' updates. Normally, this verification is done resorting to timestamps which are attributed to transactions upon entering the system. If a transaction fails the validation test, then it is aborted and restarted.

#### 2.2.3.3 Multi-Version Concurrency Control

A different angle on this study of concurrency control methods is the Multi-Version Concurrency Control (MVCC), which can be found more in depth in Wu *et al.* [48]. This method works by keeping track of multiple versions of each tuple. How a transaction interfaces with these multiple versions depends on the *flavour* of MVCC that is implemented on the system, two of which are the most interesting here: Multi-Version Two-Phase Locking (MV2PL) and Multi-Version Optimistic Concurrency Control (MVOCC).

MV2PL is a MVCC variant which makes use of locks (see Section 2.2.3.1) for controlling accesses to different versions. Every transaction must acquire the appropriate lock before being able to access any version of a tuple. To perform a read, the DBMS searches for a visible version of a tuple. A version is considered visible if it has been successfully committed to the database. If a visible version is found, then its read lock is incremented and access is given to the transaction. The process for update operations is similar, the only difference being that instead of incrementing the read lock counter, the transaction must wait until the read-lock counter turns to zero before being able to acquire the exclusive lock to the row.

MVOCC implements MVCC with a slightly different Optimistic protocol. Transactions, upon entering the system, are given a unique timestamp and must then go through three execution phases:

- *Read* phase, where transactions invoke read and update operations. Before executing a read, the transaction must fetch the most up-to-date visible version of the chain. Updates are permitted as long as they are not write-locked by another transaction *and* that the tuple it is trying to update is the most recent version.

- *Validation* phase, which occurs when the transaction wishes to commit. A new timestamp is issued for the serialization of operations between transactions, followed by a verification by the system that no conflicts were present during the transaction's execution, aborting the transaction if any conflict occurs.

- *Write* phase, which happens if the validation phase is successful, where the transactions updates are made visible to other transactions.

Independently of the protocol running, MVCC DBMSs have other important characteristics. Maintaining multiple versions of each tuple has an obvious storage and processing overhead, so every MVCC implementation must also have features that mitigate these problems.

One of these features is the *Version Storage Scheme*, which dictates how the version chain is organized. There are three possible schemes in use today:

- *Append-only*, where tuple updates happen by first acquiring an empty slot in storage for the new tuple, then copying the content of the current version to the new version and applying the modifications to the tuple in the newly allocated version slot. Since latch-free doubly-linked lists aren't possible to maintain, the main design choice for an append-only version chain is the direction from which the version chain is traversed:

  - Oldest to Newest (O2N), where new versions are added to the tail of the version chain. This scheme has the advantage of removing the necessity to have frequent index updates, since indices point to the *HEAD* of the chain. However, traversing the version chain becomes a very costly task. Normally, databases that use O2N have mechanisms in place to keep these version chains as small as possible, thus reducing the travel overhead.

  - Newest to Oldest (N2O), where new versions are added at the head of the version chain. Its advantages and disadvantages are the opposite of O2N, it has more index updates but doesn't have the travel overhead.

- *Time-Travel*, similar to append-only except the master version is kept in the table while other versions are kept in a "time-travel" table. This circumvents the index update and the traversal overhead problems, but incurs extra maintenance when pruning versions.

- *Delta*, where the master version is always in the table with "delta" version kept in a separate "delta table". Older versions simply keep the modifications done to the master version to enable in-place updates. It is great for write-intensive workloads thanks to less memory allocations, but suffers in read-intensive workloads due to increased overhead because, to read a specific version, all the transformations between the master and the selected version have to be applied.

Another important feature MVCC DBMSs implement is Garbage Collection (GC), essential for efficient version pruning. Two archetypes of GC exist today:

- *Tuple-level*, which attempts to prune individual tuple versions based on their visibility (or lack thereof) to transactions. Two techniques in this archetype exist:
  - Background Vacuuming (VAC), where background threads are used to periodically scan the database. This is the most common technique, as it is easy to implement. However, this mechanism doesn't scale well with big databases. Thus, a variant exists where transactions themselves register expired tuples in a latch-free data structure, which can then be claimed by the GC threads.
  - Cooperative Cleaning (COOP), only usable in append-only databases, where the DBMS, while traversing the version chain, marks the irrelevant information which can be claimed in the future. This scales well, since GC threads are no longer required, but has a flaw: it doesn't prune tuples that are rarely accessed by a transaction. This *'dusty corners'* problem is dealt with by using a GC thread periodically, similar to VAC.

- *Transaction-level*, where GC is done at the granularity of transactions. A transaction is considered expired by the DBMS when none of its generated versions are visible to other transactions. These versions are then erased at the end of an epoch, which is when their removal can be considered safe. It is simple and works well with transaction-local storage, but has the downside of having to keep track of the read/write sets of transactions in an epoch.

#### 2.2.3.4 Discussion

Wu *et al.* [48] have shown that the most important factor in a DBMS' performance (aside from having in-memory capabilities) is the combination of MVCC configurations, with *MV2PL/Delta Storage/Transaction-level GC* being the most performant, with Delta Storage being an enabler of high concurrency in multiprocessor environments, although it has the slowest response times, and systems using Append-Only O2N being the slowest, due to being harder to scale, thanks to longer version chain traversals.

## 2.3 Interacting with the Database

### 2.3.1 Structured Query Language

SQL[30] is a language used in RDBMS and Relational Data Stream Management System (RDSMS) to perform queries and operations. It was the first commercial language to utilize the *"Relational Model"* proposed by Codd *et al.* [6].

SQL syntax can be subdivided into different categories:

- *Clauses*, the main components of *Queries* and *Statements*, normally designate operations.
- *Expressions*, a string which produces a scalar or table value, with tables being composed of rows and columns of data.
- *Predicates*, a string that represents a condition that evaluates to a boolean value. Used to limit the effects of statements and queries or to change program flow.
- *Queries*, a combination of the previous elements that results in data being returned.
- *Statements*, a combination of the previous elements that may result in data or schemata alterations.

A small example of a SQL statement is provided in Fig. 2.1.

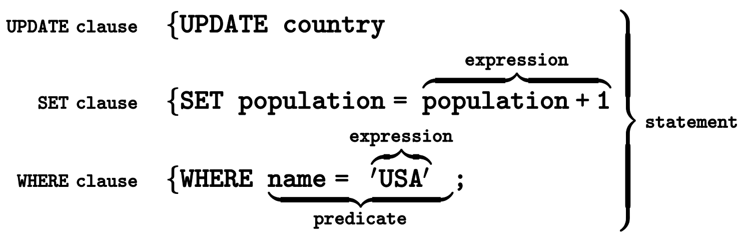


Figure 2.1: Simple SQL Syntax Example [31]

Even though SQL has been declared a standard by both the International Organization for Standardization (ISO) and the American National Standards Institute (ANSI), different RDBMSs have different *dialects* of SQL, thus sometimes some SQL code must be changed between these systems.

### 2.3.2 Java Database Connectivity

Java Database Connectivity (JDBC)[20] is an API, provided to Java applications, which specifies how a connection to a RDBMS can be made to access its data. This is done by providing methods to applications that enable queries and updates to be executed upon the database. These methods are divided into different interfaces (which can be seen in Fig. 2.2), with the main ones being the following:

- *Driver*, the base interface, is used by JDBC's *Driver Manager* to create connections to any given database.
- *Connection*, the interface that represents a connection to a RDBMS, with its methods specifying the different interactions an application can have with the database.
- *Statement*, the interface representing a SQL statement. Statements can be executed within the database with the methods specified therein. A *Statement* object can be used to execute different instructions many times.
- *Prepared Statement*, similar to the *"Statement"* interface, whose only difference is that an object implementing this interface can only be used for executing the *same* instruction many times with greater performance.
- *Callable Statement*, similar to previous *"Statement"* interfaces, but only executes *stored procedures*.
- *Result Set*, the interface which specifies how to interact with the produced results of an executed statement (if they exist).

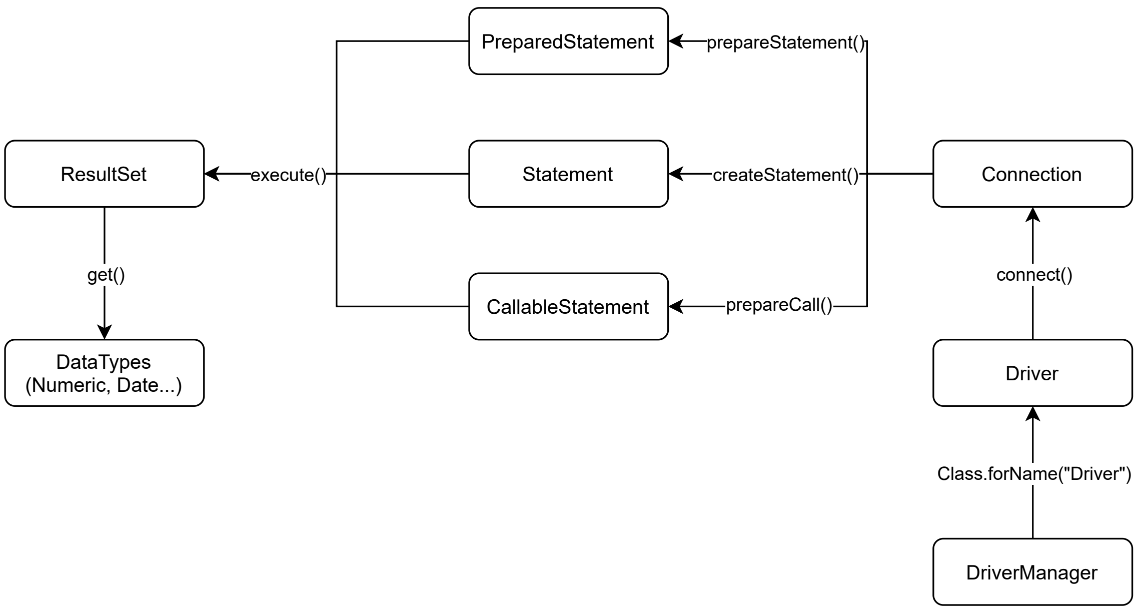


Figure 2.2: Diagram of JDBC Interfaces

#### 2.3.2.1 JDBC Driver

To be able to access a database through JDBC, an application must use a *JDBC driver*.

There are four main types of these drivers:

1. *Type 1: JDBC-ODBC Bridge* is a database-agnostic driver that makes connections through Open Database Connectivity (ODBC) [46]. Very useful when it was first released, but not recommended any more today due to poor performance.

2. *Type 2: Native-API*, partly written in Java, is a platform-specific driver that translates the client's requests into C/C++ native to the database. This driver type has better performance than type 1.

3. *Type 3: Network-protocol*, fully written in Java, is a driver that makes use of a middleware proxy that executes requests to the database on behalf of the client. The middleware itself can communicate with the database using type 1, 2 and 4 JDBC drivers.

4. *Type 4: Native-protocol*, fully written in Java, is a driver that establishes communication with the database purely through sockets. It is the most performant of all driver types, and is usually provided by the platform's vendor.

### 2.3.3 Benchmarking

In this study, we are more concerned with the performance of OLTP RDBMSs, and the industry leading organization in these matters is the Transaction Processing Performance Council (TPC)[41, 23].

According to the TPC, TPC Benchmark C (TPC-C) is an appropriate tool[1] to measure the performance of OLTP databases. TPC-C is a benchmark more complex than its predecessor TPC Benchmark A (TPC-A) due to multiple transaction types and a more complex database schema.

#### 2.3.3.1 TPC-C Specifications

TPC-C simulates a wholesale supplier company with geographically distributed sales districts and associated warehouses. Each regional warehouse is responsible for supplying 10 districts, with each district serving 3000 customers. Each warehouse maintains the 100,000 items the company sells. Fig. 2.3 depicts the business's structure.

As such, the database (see Fig. 2.4) is separated into 9 different tables:

1. *Item*, which holds all the information about the 100,000 products the company sells.

2. *Warehouse*, contains the information about all warehouses the company holds. The number of warehouses is configurable (normally 10) and is represented by $W$.

[1] Although TPC Benchmark E (TPC-E) is a more recent addition to this family of transaction processing benchmarks.

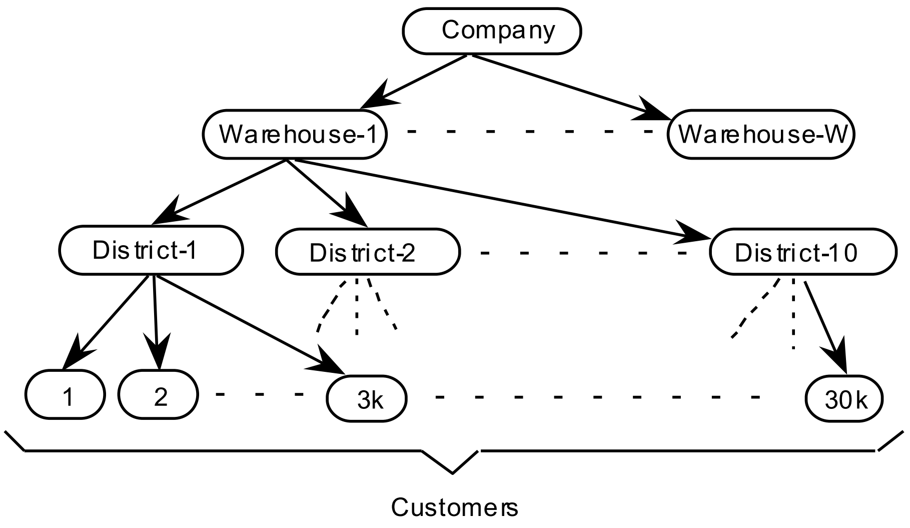


Figure 2.3: TPC-C Business Structure, from TPC-C specifications [42]

3. *Stock*, holding information about the available stock of all products in each warehouse. It contains $W * 100{,}000$ records.

4. *District*, maintains records about the different districts each warehouse supplies. It holds $W * 10$ tuples.

5. *Customer*, contains information about all of the company's $W * 30{,}000$ customers.

6. *History*, maintains the purchase history of each customer. Contains $W * 30{,}000$ records (small variations in the initial database population according to certain parameters).

7. *Order*, which holds the orders that have been received by the company. Contains $W * 30{,}000$ tuples, subject to small variations, as previously mentioned.

8. *New-Order*, maintains the information about new orders that have been placed by customers. Starts with $W * 9000$ records, subject to the same small variations as before.

9. *Order-Line*, which contains each line from each order that has been made. Contains $W * 300{,}000$ records, subject to small variations.

According to the TPC-C specifications [42], there are five types of transactions:

1. *New-Order*, the backbone of the benchmark, inserts an entire new order into the database. It is a mid-weight, read-write transaction and has the most stringent response time requirements.

2. *Payment*, represents an update to a customer's balance. It is a light-weight, read-write transaction and is tied with the *New-Order* transaction for the most stringent response time requirements.

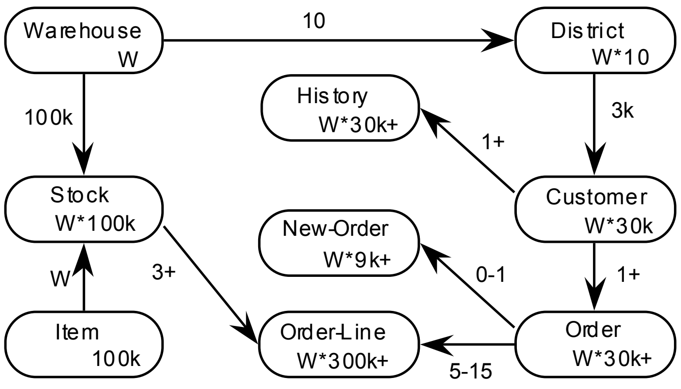


Figure 2.4: TPC-C Tables, from TPC-C specifications [42]

3. *Order-Status*, a mid-weight read-only transaction that queries the status of a customer's last order. Has low response time requirements.

4. *Delivery*, a transaction type that is intended to be executed in a deferred mode, processes a batch of 10 yet-to-be-delivered orders, with each order being delivered within the scope of this read-write transaction.

5. *Stock-Level*, calculates the number of recently sold items that have a stock level below a specified threshold. It is a heavy read-only transaction and with relaxed response time and consistency requirements.

The benchmark measures the performance of the database in Transactions per Minute (TPM). Since, by design[2], most transactions will be *New-Orders*, Orders per Minute (tpmC) can be used instead.

## 2.4 Related Work in High Performance OLTP Databases

In this section, we will explore other approaches that attempt to solve the same problem we have described in High Performance OLTP Databases environments.

### 2.4.1 Using Optimistic Concurrency Control Variants

**Silo [43]** is an in-memory database, developed by Tu *et al.*, designed with efficient memory and cache usage to allow for high performance and scalability in multicore nodes.

Its high performance is assured through the usage of an OCC commit protocol that provides serializability while being able to bypass shared-memory writes for records that didn't suffer any updates. Serializability is guaranteed by linking epochs with the commit protocol.

[2]Although it can be configured otherwise by altering the probability that each transaction type can occur.

This new OCC protocol uses Transaction IDs (TIDs), which are basically timestamps but, unlike other OCC implementations, these are generated in a distributed manner *and* only after verification can that transaction be committed. This TID must be the smallest number that obeys the following conditions:

- Larger than the TID of any record read or written by the transaction.
- Larger than the worker's most recently chosen TID.
- In the current global epoch.

The protocol is serializable because:

- All written records are locked before validating TIDs.
- It treats locked records as dirty, aborting the transaction upon encountering such tuples.
- The system guarantees that, for each requested data item at any time, the most up-to-date version is received by the transaction.

This new protocol provides *Silo* with scalability and better performance, since it reduces the bottlenecking of timestamp management and minimizes the latency in responses. The tests, performed by Tu *et al.* by making use of the Yahoo! Cloud Serving Benchmark (YCSB) and TPC-C benchmarks, show that *Silo* is capable of achieving 700,000 transactions per second with linear scalability according to the number of cores. However, these results have been obtained in a *"no contention"* setting through a single-threaded client and, as such, do not correspond to the real values had *Silo* been in a high-contention environment, as was shown by Yu *et al.* [51].

**TicToc [51]** is a new OCC protocol that aims to avoid the scalability and bottlenecking problems of prior timestamp algorithms through a new timestamp management protocol that assigns these to operations instead of transactions, using said timestamps to lazily compute a valid commit timestamp in a distributed manner.

Since timestamps are attributed to operations which, in a horizontally scaled database, are executed in different nodes, this reduces the bottlenecking of centralized timestamp management of previous OCC implementations. This allows the protocol to scale really well and, additionally, the laziness component allows the DBMS to further exploit parallelism, which greatly improves its performance.

To better understand *TicToc*, a short analysis of the following example is required. Two transactions (*A* and *B*) have the following interweaving of instructions:

$$A{:}read(x) \rightarrow B{:}write(x) \rightarrow B{:}commit \rightarrow A{:}write(y)$$

Although this line of execution is serializable, as $A$ can be ordered before $B$, ordinary OCC protocols would abort this execution because $A$'s read value for $x$ had been updated by $B$. *TicToc* is able to allow this interweaving due to being aware of *when* $A$ read $x$, which then allows for the aforementioned serialization.

Yu *et al.* report, according to their benchmarks obtained via TPC-C, almost twice the performance with a third of the abort rate in low-contention when compared to previous OCC implementations, with behaviour in high-contention reducing the gap, but still showing an improvement. When compared with *Silo*, *TicToc* has achieved better throughput of operations with similar abort rates.

**ROCOCO** [32] is a distributed concurrency control protocol that executes transactions as a collection of atomic pieces, with each piece being executed by a different server for execution, with reordering of operations if conflicts are found.

This division of transactions into different pieces requires *a priori* knowledge of how the system is partitioned (to correctly send the piece to the responsible partition) and which transactions the system will execute (since static analysis needs to be performed to be able to divide it correctly).

Mu *et al.* report that *ROCOCO*, when compared to both OCC and 2PL, has a higher throughput and a lower response time across all contention levels when executing the TPC-C benchmarks.

### 2.4.2 Using Hybrid Protocols

The following protocols are considered "hybrid" because they can use different CC strategies during runtime.

**Strife** [34] is a new CC protocol that exploits data contention to improve performance in multicore systems in high contention environments. It achieves its high performance by batching incoming transactions into clusters that can then be executed with uncontrolled parallelism without any data conflicts. Transactions that can't be put into clusters will be run concurrently with traditional CC protocols.

The protocol has three phases:

- *Analysis* phase, where the protocol takes the incoming transactions and batches them into clusters that are data-conflict free.This is done by using a *data access graph*, a bipartite graph where one side is the set of transactions and the other is the set of accessed data items, with edges representing the relation "*Transaction X accessed Data Item Y*". Transactions that were not clustered are then marked as residual for later processing.

- *Conflict-free* phase, where the protocol distributes the clusters of transactions by the CPU cores to be run in parallel in an uncontrolled fashion, with transactions of the same cluster being executed serially by the same core. Strife can guarantee the correctness of these executions because it makes sure every cluster doesn't share any data dependencies with the other clusters. This phase ends when all cores have finished processing their cluster of transactions. Cores that have finished earlier must wait for stragglers, otherwise correctness cannot be guaranteed.

- *Residuals* phase, where the transactions that could not be clustered are processed. The protocol ensures that these are run in parallel with CC. Any CC protocol can be used for this phase, but Prasaad *et al.* implemented 2PL with a *No Wait* policy, meaning that any transaction that fails to acquire a lock is aborted and must be restarted.

Results show that *Strife* achieves twice better throughput both on YCSB and TPC-C benchmarks when compared to other CC protocols under normal circumstances, and up to four times better when under high-contention environments, with the ability of improving its performance if the number of *"hot items"* (records that are heavily requested by different transactions) that are independently accessed increases. However, *Strife* underperforms in low-contention environments when compared to those same protocols, due to having an *"Analysis"* phase that takes longer than the time it would take had the system simply executed the transactions when received.

**Salt [50]** is a distributed database that offers transactions the possibility of using either the ACID API or the Basically Available, Soft state, Eventual consistency (BASE) [35]. The BASE API weakens the consistency and isolation constraints that ACID requires to be able to exploit greater performance.

It does so by allowing programmers to create a subtransaction from a normal ACID transaction that runs at a BASE level. Correctness during the interaction between BASE and ACID transactions is maintained by the *Salt Isolation* property, which permits BASE transactions to only see other BASE transactions' internal states, and even still, only at well-defined spots. ACID transactions retain their full isolation to themselves.

Presented results show, when compared to a MySQL distributed cluster (from which *Salt* was derived from), a performance improvement of over six times on the TPC-C benchmark and of six and half times higher when on Fusion Ticket.

**Callas [49]** sharing some ancestry with *Salt*, is a distributed ACID database which employs a new kind of concurrency control protocol, Modular Concurrency Control (MCC). MCC works by "decoupling the ACID abstraction from its implementation", meaning it partitions the transactions into groups (similarly to *Strife*) and each group has its own optimized concurrency control mechanisms. The groupings are made by a *chopping tool* that uses heuristics to statically analyse the transactions' workload and identify the groupings that increase concurrency.

MCC guarantees isolation intra-groupings and uses *nexus locks*, a new lock type made specifically for *Callas*, that guarantees isolation for transactions that belong to different groups.

Xie *et al.* report, when compared to a MySQL distributed cluster (from which both *Salt* and *Callas* were derived from), a performance improvement of over eight times when running the TPC-C benchmark, nearly six times better performance on *Fusion Ticket* and over six times better performance on Front Accountant, achieving very similar results to those of Salt.

### 2.4.3 Using New Hardware and/or Software Capabilities

**Star™ [10, 9, 11]** is a tool for Software Transactional Memory (STM) Java applications that statically detects if any pair of transactions may cause a *write skew* anomaly, which leads to unpredictable behaviours.

A *write skew* is a serializability anomaly that occurs in Snapshot Isolation. An example of this anomaly is when two transactions $X$ and $Y$ try to execute the following statements:

$$X := x + y \tag{2.1}$$

$$Y := y + x \tag{2.2}$$

For this example, it is possible to find a trace of execution that is not serializable and yields unexpected results. These anomalies occur when two transactions are writing on different memory addresses (in the previous example, $x$ and $y$) but are also reading data that is being modified by the other.

Star™prevents these anomalies from happening by making use of *Separation Logic*[36] to perform shape analysis[13], producing memory locations for a transaction's *Read* and *Write* sets, enabling the execution of said transaction in Snapshot Isolation or, when Snapshot Isolation is not possible, the enforcement of full serializability semantics.

**Fast Remote Memory (FaRM) [14, 37]** is a distributed transactional system that exploits Remote Direct Memory Access (RDMA) and non-volatile Dynamic Random-Access Memory (DRAM) to reduce the network bottlenecks, when compared to main memory systems built on top of the Transmission Control Protocol / Internet Protocol (TCP/IP), and storage bottlenecks by keeping information in-memory. Its main advantages are the performance gains from the exploited hardware features and a new OCC protocol based on timestamps.

Thanks to these improvements, advantages that other systems may be able to exploit as well due to their physical nature, Shamis *et al.* have been able to report 5.4 million *New-Order* operations per second with the TPC-C benchmark.

**Hardware Transactional Memory (HTM)** [27, 33] is a commercial solution used in multi-core processors such as the *IBM POWER8**. It consists of a set of *Power ISA* instructions to implement a best-effort and isolated transactional memory system, with hardware-based mechanisms for checkpointing and rollbacks provided for efficient creation, restoration and committal of transactions.

This enables the usage of two features: Transactional Lock Elision (TLE) and Thread-Level Speculation (TLS).

TLE allows for the execution of critical sections to be concurrent across many threads without needing a lock. In other words, it converts these critical sections into transactions that can be dynamically analysed for conflicts.

TLS enables concurrent execution of serial operations in a program (such as loop iterations) even if compilers cannot detect the existence of data inter-dependencies by enclosing these operations in a transaction where, in a similar fashion to TLE, the system can dynamically analyse them for conflicts, rolling back and re-executing the operation as necessary. A shared control variable is used to guarantee the serial semantics of the original operations, but this can create conflicts which increase the abort rate of transactions. To combat this, *suspend regions* are used to be able to read this variable without said read becoming part of the transaction's footprint, thus eliminating these kinds of conflicts.

When testing the benefits of TLE, results have shown that, with a modified *Pthreads* library, the speedup experienced by the system in *"high lock contention, no conflict"* environments is significant, being directly proportional to the number of active threads. This is, however, a favoured case, as the results from other conditions show increased performance to a much lesser degree, with the *"high lock contention, no conflicts, and a data set that overflows the transactional memory footprint capacity"* yielding virtually no benefits.

For TLS, results show that making use of suspend regions increase the system's speedup significantly in single-threaded executions with transaction failures eliminated.

**Snapshot Isolation Hardware Transactional Memory (SI-HTM)** [16] is an improvement for HTM proposed by Felipe *et al.* that implements Snapshot Isolation through a software layer. SI-HTM exhibits improved scalability, achieving speedups of up to 300% relatively to HTM on in-memory database benchmarks.

### 2.4.4 Using Futures

LSD [45] is an API that implements the notion of *futures* for clients requesting operations to databases, thus bringing all the previously mentioned benefits of *lazy computing* without radical changes in how programmers deal with transactions.

#### 2.4.4.1 Brief Introduction

LSD aims to reduce transaction conflicts by reducing the amount of time that a transaction requires isolation. Isolation is only required during the *Commit* phase, when the database will resolve the futures developed during the transaction's execution. This approach means that, while executing, transactions don't (normally) observe specific database states but rather abstract states, relying only on the futures given by LSD to execute their operations.

The way LSD achieves such goals is by modifying the traditional *Transactional API*, through the modification of the *WRITE* and *READ* operations, enabling them to return and use futures, and by implementing a new operation *IS-TRUE*, whose function is to return whether a conditional expression (which can include futures) is true or false.

Table 2.2: Traditional API *vs* LSD API

| **Operation** | **Traditional API** | **LSD** |
|---:|---|---|
| *BEGIN* | Starts a new transaction | Same as before |
| $READ(key)$ | Returns value $X$ held at $key$ | Returns future □ that represents the value held in the future-key $key$ |
| $READ(\triangle)$ | Does not exist | Returns future □ that represents the value held at △ |
| $IS-TRUE(\square)$ | Does not exist | Returns the boolean value of the condition □ in the database |
| $WRITE(key, X)$ | Writes the value $X$ into $key$ | Does not exist |
| $WRITE(key, \square)$ | Does not exist | Writes the future value □ into $key$ |
| $WRITE(\triangle, \square)$ | Does not exist | Writes the future value □ into the future-key △ |
| *COMMIT* | Attempts to commit the transaction | Same as before |
| *ABORT* | Aborts the transaction | Same as before |

The main benefits of using LSD are increased throughput and reduced latency, which are achievable thanks to the reduced contention window. This reduction is possible because isolation is required for smaller lengths of time and through a relaxation of the set of transactions that must be forced to abort to maintain said level of isolation. It is important to note, however, that transactions that require a specific database state must fallback to the traditional *READ* instructions, not benefiting from these improvements, but these transactions are expected to be a minority.

LSD can work with both OCC and 2PL techniques.

#### 2.4.4.2 Model and Design

LSD follows the classic *Client-Server* architecture, with the *Client* being responsible for running the application code and the *Server*, which can be partitioned or replicated, being responsible for managing the DBMS. Both of these components can be run in the same machine or in different locations and communication between them is done through the LSD API.

Operations exposed by the LSD API (briefly described in Table 2.2) are the following:

- $BEGIN$, which begins the transaction.
- $READ(key)$ or $READ(\triangle)$, returns a future, which is an abstract representation of the requested object. The transaction will continue its execution, applying transformations to this future instead of the real value, which only later will be resolved by the database.
- $IS-TRUE(\square)$, for conditional branching inside the transaction. This new operation is necessary to be able to evaluate expressions with future values, as their real values are unknown to the transaction. It returns whether the passed condition $P$ is true, but does it over an abstract state, meaning the futures aren't resolved and, as it does, maintains the isolation between transactions. This operation has different behaviours depending on the CC protocol used: either the condition is validated during commit, if using OCC, or locks are put in place to ensure the output of the condition doesn't change until the commit phase, in case of 2PL.
- $WRITE(key, \square)$ or $WRITE(\triangle, \square)$, instructs the database to write into a future tuple $\triangle$ or to a specific tuple *key* the future value $\square$. The future(s) $\triangle$ and $\square$ are only evaluated by the database during the commit phase.
- $COMMIT$, instructs the database that the transaction wishes to commit. All the futures generated during its execution must be resolved at the beginning of the commit process.
- $ABORT$, which simply aborts the transaction.

To better understand the differences between both APIs, consider the example found in Fig. 2.5, where it can be seen that the general structure of the transaction remains the same, which shows the user-friendliness of LSD, with the only major change being the introduction of the futures.

Of course, this begs the question of how does LSD deal with concurrency and correctness since futures aren't immediately computed. It depends on which protocol is being used, but there is some common ground. In addition to the previously required *read* and *write* sets already maintained by the DBMS, LSD requires the maintenance of three new sets: a *future_reads*, to maintain the unresolved read sets, *future_writes*, analogous to *future_reads* set but for writes, and *future_conditions*, where the conditions used in *IS-TRUE*

**Algorithm 1** Traditional API

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

**Algorithm 2** LSD API

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

Figure 2.5: Traditional API vs. LSD API example, from Vale *et al.* [45]

operations are kept to be computed later. From this point onwards, LSD's behaviour changes depending on the protocol being used.

For OCC, each data item has an associated version. When the database receives read or write operations, they are registered into their respective future sets. When a transaction wishes to commit, the DBMS resolves and locks the tuples found in *future_reads*, followed by locking of *future_writes* and of *writes*. It then validates the *reads*, *future_reads* and *future_conditions* sets. If validation is passed, it resolves the *future_writes* and updates the database state with the values from the resolved *future_writes* and *writes* set, releasing all the locks in the end.

For 2PL, all objects that transactions wish to access must be locked. In addition to previously established locks, a new lock was implemented specifically for the *IS-TRUE* operation: a *conditional lock*, which supports the traditional read-write modes plus two more:

- *Read condition*, which associates an item with a condition $\square$.
- *Write value*, which is acquired by a transaction that wishes to update the locked item with a new value that respects all the *read condition* locks imposed on it.

### 2.4.5 Discussion

LSD's improvements, which aim to reduce the possible contention window of transactions, make it an interesting solution in the sense that it is compatible with the previously mentioned solutions. In other words, LSD, by acting as an extra layer between the client and the DBMS, can be used in conjunction with other solutions that optimize the functioning of the database, which can vary from protocols such as *Silo*, *TicToc* and *Strife*, to hardware improvements such as FaRM and HTM. This compatibility stems from attempting to improve the client-server communication instead of simply improving the server's performance. In other words, it attempts to solve the problem from another angle, thus it doesn't interfere with how other propositions deal with the difficulties.

# 3 A JDBC Driver for LSD

In this chapter, we will learn how to implement a JDBC driver (Sec. 3.1) and how a LSD-JDBC driver was created, as well as what challenges presented themselves during its development and how they were addressed (Sec. 3.2).

## 3.1 How to implement a JDBC driver?

In order to implement a JDBC driver, two things are necessary: registering a driver implementation with the *"Driver Manager"* and implement the JDBC interfaces.

To register the driver with the *"Driver Manager"*, the driver must be configured properly as to be discoverable by the *Java Virtual Machine (JVM)*. To do this, a Java Archive (JAR) containing the driver must have a *manifest file* and a *"services/"* directory explicitly declaring what functionalities are being exposed to the JVM (see Fig. 3.1).

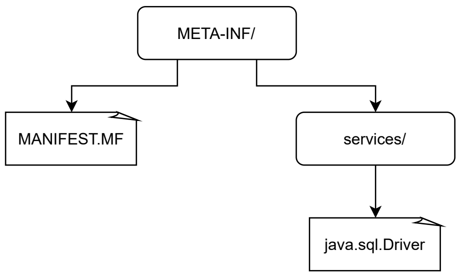


Figure 3.1: The *"META-INF"* directory and its contents

The manifest file is named *"MANIFEST.MF"* and is located under the *"META-INF"* directory in the JAR, containing a list of key/value pairs grouped into sections. These pairs supply metadata that help us describe aspects of our JAR such as the versions of packages, what application class to execute, the classpath, among others (see Listing 3.1).

A *service provider* is identified by placing a provider-configuration file in the *"services/"* directory (see Listing 3.2). In our case, since we are writing a new driver, we must reference what Java interface we are implementing. In this case, we're implementing the *"java.sql.Driver"* API and so, we have our service name (and, by consequence, the file

```
Manifest-Version: 1.0
Main-Class: lsd.util.LSDJDBCMain
Automatic-Module-Name: lsd.driver.jdbc
Implementation-Vendor-Id: lsd
Implementation-Version: 1.0
Implementation-Title: LSD JDBC Driver
Implementation-Vendor: LSD Development Group
Specification-Vendor: Oracle Corporation
Specification-Title: JDBC
Specification-Version: 4.2
```

Listing 3.1: The *MANIFEST.MF* file contents

```
lsd.Driver
```

Listing 3.2: The *java.sql.Driver* service file contents

name). The contents of the file must point to the class that implements the API we're using. In our case, *"lsd.Driver"* is that class.

### 3.1.1 JDBC Interfaces and their Roles

The second thing needed to implement a JDBC driver is, as previously stated, the implementation the JDBC interfaces to create our own Java classes.

Let's start by exploring more in-depth the primary interfaces of the JDBC API (see Sec. 2.3.2.1) [38].

#### 3.1.1.1 Driver

The *"java.sql.Driver"* is the simplest of the interfaces to implement, since it has the least number of methods that need implementation, since it mainly acts as the *"Main"* for the driver, with its most important method being *"connect"*, as it returns a *Connection* object that establishes the communication between the client and the database.

#### 3.1.1.2 Connection

The *"java.sql.Connection"* interface manages the connection between the client and the database through the creation of *"Statement"* objects that execute operations upon the database.

Important methods include:

- *"setAutoCommit"*, a method that sets the behaviour of the JDBC connection to either commit after every single *Statement* execution if set to *TRUE*, or only commit when instructed by the application if set to *FALSE*.

```
val sql1 = ''SELECT d_next_o_id '' +
           ''FROM bmsql_district ''

val sql2 = ''SELECT d_next_o_id '' +
           ''FROM bmsql_district '' +
           ''WHERE d_w_id = 7 '' +
           ''AND d_id = 1 '' +
           ''FOR UPDATE''

val stmt = conn.prepareStatement()

stmt.execute(sql1) // Execute sql1
stmt.execute(sql2) // Execute sql2
```

Listing 3.3: A simple *Statement* example

- *"createStatement"* (and its variants[1]), which returns a *"Statement"* object to be manipulated by the application.
- *"commit"*, which executes the database's commit protocol for the current transaction. Cannot be executed if the *"Connection"* is configured to automatically commit after each *"Statement"* execution.
- *"rollback"*, which undoes the active transaction's modifications (either all of them or up until a *savepoint* that can be supplied as an argument). As with the *"commit"* method, *"rollback"* can only be executed if *"autoCommit"* parameter is set to *FALSE*.

#### 3.1.1.3 Statements

The *"java.sql.Statement"* is an interface that allows for the parametrization and execution of SQL statements.

There are three types of statements:

***Statement*** is the standard go-to class for executing SQL instructions. A single *Statement* object can be used for many different instructions, since its *execute* method receives as a parameter the SQL instruction. Listing 3.3 depicts a simple Kotlin example where a *Statement* is created and executed.

***Prepared Statement*** is an interface that offers similar functionalities as the *Statement* interface, but is designed to instead run the same SQL instruction (with possibly different parameters) many times, with higher performance than a normal *Statement*. As such, the instruction is passed as a parameter when invoking the *prepareStatement* function. Listing 3.4 depicts a simple Kotlin example where a *Prepared Statement* is created and executed.

[1] *"prepareStatement"* for Prepared Statements and *"prepareCall"* for Callable Statements.

```
val sql = ``SELECT d_next_o_id '' +
                   ``FROM bmsql_district '' +
                   ``WHERE d_w_id = ? '' +
                   ``AND d_id = ? '' +
                   ``FOR UPDATE''

val stmt = conn.prepareStatement(sql)
var wID = 7
var dID = 1
stmt.setInt(1, wID) // This would set d_w_id as 7
stmt.setInt(2, dID) // This would set d_id as 1
stmt.execute() // Execute sql with first set of parameters

wID = 3
dID = 6

stmt.setInt(1, wID) // This would set d_w_id as 3
stmt.setInt(2, wID) // This would set d_id as 6
stmt.execute() // Execute sql with second set of parameters
```

Listing 3.4: A simple *Prepared Statement* example

```
val sql = ``{call getWID (?)}''
val stmt = conn.prepareCall(sql)
val wID = 5
stmt.setInt(1, wID) // This would set ID as 102
stmt.execute()
```

Listing 3.5: A simple *Callable Statement* example

***Callable Statement*** like the *Prepared Statement* interface, is designed to execute a single SQL instruction many times, with possibly different parameters. However, instructions passed onto a *Callable Statement* object must be a call to a *stored procedure* of the database. Listing 3.5 depicts a simple Kotlin example where a *Callable Statement* is created and executes the stored procedure *getWID*.

#### 3.1.1.4 Result Set

The *"java.sql.ResultSet"* interface represents an object that is returned by all *Statement* classes. This object's methods expose the results of an executed SQL instruction, serving as a wrapper to be able to parse the database's response.

The object itself acts as an iterator, exposing a single row at a time. To advance to the next row, a *"next"* method invocation is required. If there aren't any more rows when *"next"* is executed, an exception is thrown, but this can be avoided by using the *"isLast"* method, which returns a boolean value that determines if the current row is indeed the last one returned.

```
val sql = ‘‘SELECT d_next_o_id ’’ +
                    ‘‘FROM bmsql_district ’’

var prepStmt = conn.prepareStatement(sql)
val rs = prepStmt.executeQuery() // execute and get ResultSet

val oIDs = mutableListOf<Integer>()
while (!rs.isLast()){
    rs.next() // move to the next row of results

    // get current d_next_o_id
    oIDs.add(rs.getColumn("d_next_o_id"))
}
```

Listing 3.6: A simple *Result Set* example

Each row is divided into columns that can be accessed by using the *"getColumn"* method. Its value is then returned as the Java primitive type that correctly represents its type in the database table (i.e. a "numeric" is converted to "Integer", *etc.*).

A simple example of a *Result Set* and how it is used can be seen in Listing 3.6.

#### 3.1.1.5 JDBC in action

After having a basic understanding of how JDBC works, a simple example showing how an application would interact with the driver is in order. This is what we will explore with the help of Listings 3.7 and 3.8.

These listings show us a simple Kotlin[2] application that reads from a database a value (*Next Order ID*, analogous to *"d_next_o_id"*) and increments it by one in a single transaction.

First, the application must fetch the *properties* file (see Listing 3.9) that holds the necessary information to establish a connection to the database (lines 2-3). Then, it registers the JDBC driver with the *"Driver Manager"* (line 6) and establishes a connection configured for manual commits (lines 9-10).

Afterwards, it generates two parameters, *"dWID"* and *"dID"*, analogous to *"d_w_id"* (district warehouse ID) and *"d_id"* (district ID) respectively (lines 14-15) and creates the first *Statement* for the transaction, where the *"d_next_o_id"* field referring to both previous parameters is read (lines 17-30). Upon completion of this first statement, both itself and its *"ResultSet"* are closed, to be later cleaned by Java's GC (lines 33-34).

Finally, for the last part of the transaction, the update is created and executed in a similar fashion to the previous statement (lines 36-47), the transaction is committed (line 50) and the connection is closed (line 51), signalling the end of the application's execution.

[2]Note that Kotlin can be compiled to Java bytecode, hence it can be run in a JVM and can make use of Java libraries, thus JDBC can be used to its full effect in Kotlin applications.

```
// Get connection properties
val props = Properties()
props.load(
    this.javaClass.getResourceAsStream("conn.properties")
)

// Register driver with the DriverManager
DriverManager.registerDriver(Driver())

// Create connection and set its behaviour to
// 'autoCommit = off'
val conn = DriverManager.getConnection(
                props.getProperty("connectionString")
            )
conn.autoCommit = false

val dWID = 2
val dID = 1

// Read 'd_next_o_id'
val readNOID = ``SELECT d_next_o_id '' +
                    ``FROM bmsql_district '' +
                    ``WHERE d_w_id = ? '' +
                    ``AND d_id = ? '' +
                    ``FOR UPDATE''

var prepStmt = conn.prepareStatement(readNOID)
prepStmt.setInt(1, dWID) // set first parameter
prepStmt.setInt(2, dID) // set second parameter
val rs = prepStmt.executeQuery() // execute and get ResultSet

rs.next() // move to the first row of results
val oID = rs.getColumn("d_next_o_id") // get d_next_o_id
```

Listing 3.7: A Simple JDBC transaction in Kotlin

```
rs.close()
prepStmt.close()

// Update 'd_next_o_id'
val updateNOID = ``UPDATE bmsql_district '' +
                 ``SET d_next_o_id = ? + 1 '' +
                 ``WHERE d_w_id = ? '' +
                 ``AND d_id = ? ''

prepStmt = conn.prepareStatement(updateNOID)
prepStmt.setInt(1, oID)
prepStmt.setInt(2, dWID)
prepStmt.setInt(3, dID)
prepStmt.executeUpdate()
prepStmt.close()

// commit transaction, composed from both previous statements
conn.commit()
conn.close() // close connection
```

Listing 3.8: A Simple JDBC transaction in Kotlin (cont.)

```
db=postgres
driver=lsd.Driver
conn=jdbc:lsd:database://0.0.0.0/example_bd
user=user
password=beriÇkureP4ss
```

Listing 3.9: A Simple *Properties* file

## 3.2 The JDBC Driver for LSD

Now that we have a basic understanding of how a JDBC driver works, we are ready to implement a JDBC driver that can use both the traditional API and the LSD API (recall Sec. 2.4.4) for communication with the database.

### 3.2.1 Extending SQL with LSD operations

The first step to create a LSD-JDBC driver is to consider how an LSD instruction should be written. In other words, how would LSD influence SQL's syntax.

Since we wish to maintain both the traditional SQL and LSD APIs exposed to the application so that it could use them both how it sees fit, we chose to implement new instructions that closely follow SQL's already existing syntax instead of overriding our counterparts.

Recalling Table 2.2, where the possible LSD instructions were listed, Table 3.1 shows how each LSD instruction correlates to our LSD-SQL API (omitting the already extensive, but still available, SQL instructions).

Table 3.1: Pure LSD syntax *vs* LSD-SQL syntax

| **Pure LSD** | **LSD-SQL** |
|---:|:---|
| $BEGIN$ | $BEGIN$ |
| $READ(key)$ | $SELECT\_LSD$ |
| $READ(\triangle)$ | $SELECT\_LSD$ |
| $IS-TRUE(\square)$ | $IF\_LSD$ |
| $WRITE(key,\square)$ | $INSERT\_LSD$ if value does not exist yet in the database, $UPDATE\_LSD$ otherwise. |
| $WRITE(\triangle,\square)$ | $INSERT\_LSD$ if value does not exist yet in the database, $UPDATE\_LSD$ otherwise. |
| $COMMIT$ | $COMMIT$ |
| $ABORT$ | $ROLLBACK$ |

Table 3.2 shows a few examples of how a normal SQL statement would be translated to LSD-SQL[3].

Table 3.2: SQL instructions and LSD-SQL's counterparts

| **SQL** | **LSD-SQL**[4] |
|---|---|
| SELECT d_city FROM bmsql_district | SELECT_LSD d_city FROM bmsql_district |
| SELECT COUNT (d_city) FROM bmsql_district | SELECT_LSD COUNT (d_city) FROM bmsql_district |
| UPDATE bmsql_district SET d_next_o_id = d_next_o_id + 1 WHERE d_w_id = ? AND d_id = ? | UPDATE_LSD bmsql_district SET d_next_o_id = □ WHERE d_w_id = ? AND d_id = ? |
| INSERT INTO bmsql_item (i_name, i_price, i_data, i_im_id) VALUES (?, ?, ?, ?) | INSERT_LSD INTO bmsql_item (i_name, i_price, i_data, i_im_id) VALUES (?, ?, ?, ?) |

Listing 3.10 and show an example of an *IS-TRUE* statement. An *IS-TRUE* statement is comprised of a Boolean expression which is divided into three parts:

1. *Expression 1*, which can be a simple parameter like an integer or a future.

2. *Logical Operator*

[3]Although normal SQL instructions can still be used if futures are not needed.

Listing 3.10: IS-TRUE instruction

```
// IS-TRUE Template
IS-TRUE expression1 logical_operator expression2

// IS-TRUE Example
IS-TRUE ? > 10;
```

Listing 3.11: IF_LSD instruction

```
// IF_LSD Template
IF_LSD 'condition' ;; thenBranch ;; elseBranch;

// IF_LSD Example
IF_LSD IS-TRUE ? > 10 ;;

    UPDATE_LSD bmsql_stock
        SET s_quantity = ? - ?, s_ytd = ? + ?,
        s_order_cnt = ? + 1,
        s_remote_cnt = ? + ?
        WHERE s_w_id = ? AND s_i_id = ? ;;

    UPDATE_LSD bmsql_stock
        SET s_quantity = ? + 91, s_ytd = ? + ?,
        s_order_cnt = ? + 1,
        s_remote_cnt = ? + ?
        WHERE s_w_id = ? AND s_i_id = ?";
;
```

3. *Expression 2*, analogous to *Expression 1*.

Listing 3.11 shows an example of an *IF_LSD* instruction. An *IF_LSD* statement is comprised of three parts:

1. *Condition*, which must be an instruction that returns a Boolean value.
2. *Then Instruction*, which contains the instruction to be executed if the *Condition* is resolved to 'true'.
3. *Else Instruction*, same as *Then Instruction*, but executed when the *Condition* is resolved to 'false'.

### 3.2.2 Driver Architecture

Now we will explore the architecture of the driver we have developed, as well as the reasons for some of the choices we have taken along our path.

First and foremost, we have identified two main components we need to create: a parser and the JDBC interfaces. However, there is a first step we must consider: how will the communication with a database be established?

#### 3.2.2.1 Communicating with the database

To maximize compatibility with different databases, we have opted to create a *Type 3 JDBC driver* (recall Sec. 2.3.2.1), acting as a middleware between the application and an underlying *Type 4 JDBC driver*. The underlying driver to use is chosen automatically by the *"DriverManager"* when our middleware requests a connection between it and the database the application requested.

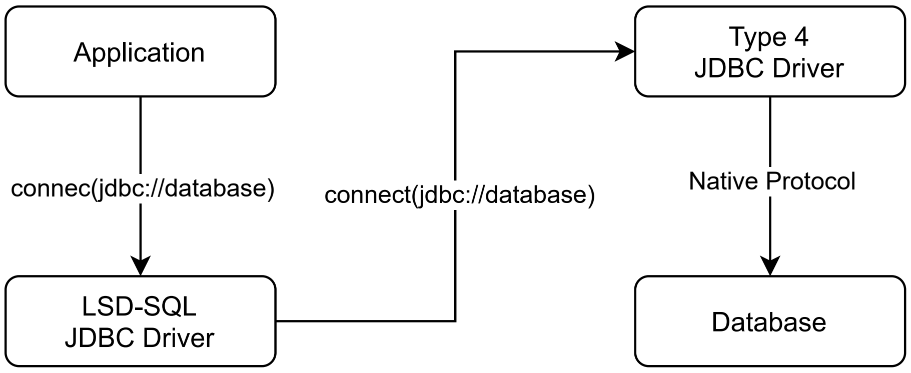


Figure 3.2: LSD-SQL JDBC Driver network architecture

This enables our driver to execute *Statements* on behalf of the application or delay certain executions, returning instead a *future* that the application can use until it becomes absolutely necessary to know the exact value, i.e. when the application attempts to commit a transaction.

#### 3.2.2.2 Parser

The parser's function is to take the application's SQL input, evaluate it, convert it (if necessary) and transform it into an *"Operation"* class to be easily altered for the driver's needs. This subsystem can be subdivided into two categories:

- The *Parser* itself, responsible for reading the input and creating the appropriate *"Operation"* object.
- The *"Operation"* class and its derivatives. Each different class represents a different kind of LSD-SQL operation supported by the driver, similar to how an Abstract Syntax Tree (AST) [21] works. The available operations are:
  - *StandardSQL*, an operation whose instruction has been shown to be just normal SQL. The Parser also defaults to this operation when it cannot recognize if the instruction belongs to the LSD-SQL API.

- *LSDOperation*, an abstract class representing an operation that belongs in the LSD API. Operations that implement this class are:
  - *SelectLSD*, used for SQL queries that return a future.
  - *InsertLSD*, used for SQL insertions that are only executed when committing the transaction.
  - *UpdateLSD*, used for SQL updates that are only executed when committing the transaction.
  - *IsTrue*, used for conditional execution of an instruction depending on the state of the database during the commit phase.
  - *IFLSD*, used for branching execution of conditions depending on the state of the database during the commit phase. This is a *"new"* LSD operation but, in actuality, it executes one of two different instructions depending on the result of an *IsTrue*.

These operations are then used by the rest of the driver to execute the *Statements* in different ways, which will be explored in the following section.

#### 3.2.2.3 JDBC

From the JDBC API, our driver only implements the following interfaces:

- *Driver*, which we are required to do so that the driver is discoverable by the *"Driver Manager"*.

- *LSDConnection*, which implements a fairly standard *Connection* object, mostly invoking the counterpart methods of the underlying driver, with the only exceptions being the commit protocol and its supporting data structures (these will be further discussed in Sec. 3.2.3).

- *LSDStatement*, which implements a *Statement* object. The only noteworthy differences from a standard *Statement* implementation are the *"execute"* methods, since they request the Parser for the *Operation* they are about to execute and, depending on the type of operation, the *Statement* executes in different ways.

- *LSDPreparedStatement*, which implements a *PreparedStatement* object, has similar behaviour to the *LSDStatement*, but the Parser is invoked during the initialization of the object instead of when the *execute* method is called.

- *LSDResultSet*, which serves only as a wrapper for a *LSDFuture* object that contains the future that the application must use for its execution and will contain the actual value produced by the database during the commit phase.

Other instructions that our driver cannot serve or understand are sent to and executed by the underlying driver.

### 3.2.3 Implementation Details

This section mirrors the structure of the previous section (Sec. 3.2.2), as we will elaborate about the implementation details of each of the main components of our driver.

#### 3.2.3.1 Communicating with the database

The two main classes mostly responsible for establishing the communication with the database are the *"Driver"* and the *"LSDConnection"*.

**The *Driver*,** before creating a new *"LSDConnection"* object, first checks if it can accept the Uniform Resource Locator (URL) passed by the application. As of the time of writing, the driver can only accept JDBC URLs that start with "$jdbc : lsd : \{database\}$". This differentiates from the established URL "$jdbc : \{database\}$" format because regular drivers would be chosen by the *"Driver Manager"* instead of our own.

Figure 3.3 shows how a connection is made. The application requests a connection to "$jdbc : lsd : \{database\}//0.0.0.0/example\_bd$". Our driver thus requests a connection to "$jdbc : \{database\}//0.0.0.0/example\_bd$"[5], to which the underlying driver acknowledges and proceeds to establish a raw connection to the database, wrapping it into a JDBC *"Connection"* object and returning it to our driver. Finally, a *"LSDConnection"* is returned by our driver to the application.

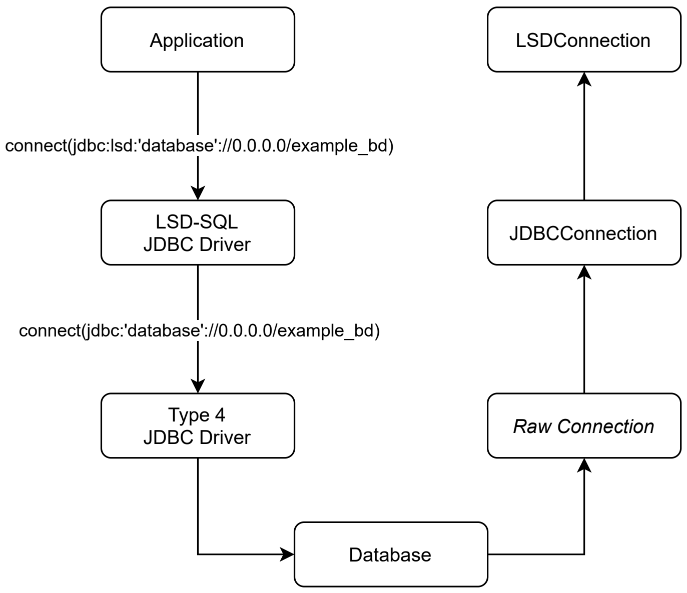


Figure 3.3: How a connection with the database is established.

The *"LSDConnection"*, upon initializing, creates a *"Parser"*, which will be used for all incoming instructions, and also creates two data structures:

[5]Note the URL conversion.

- *operations*, a linked list of *"LSDOperations"*, representing the sequence of LSD instructions present in the current transaction.
- *readMap*, an hash map of *"$String \rightarrow LSDFuture$"*, which contains all futures generated by the driver that will be resolved during the commit phase.

Other important features of the *"LSDConnection"* will be further elaborated in Sec. 3.2.3.4.

#### 3.2.3.2 Parser

The *Parser* subsystem, as stated in Sec. 3.2.2, can be subdivided into the *"Parser"* and the *"Operation"* classes.

**The *Parser*,** when receiving *"parse"* requests, takes the string input and splits it into different tokens and matches them into one of the supported *"Operation"* classes.

Additionally, when initializing, it creates a *"JexlEngine"*[5] object so that it can evaluate Boolean expressions that a string variable might contain.

Other important functions of the *"Parser"* will be discussed in Sec. 3.2.3.4

**The *Operation*** classes represent all operations that can be executed by the driver. These operations receive and store the SQL instruction into a *"query"* variable. Any *"LSDOperation"* keeps track of all parameters and their values from its query in a tree map. Before executing, any *LSDOperation* must be prepared, i.e., its parameters must be set into the instruction that will be sent unto the underlying driver. This is the job of the *"prepareForLSDExecution"* method.

Additionally, different *"LSDOperation"* classes have other uses for this variable:

- *"SelectLSD"* extends the *"prepareForLSDExecution"* to also generate the future's Identification (ID).
- *"IsTrue"* uses the aforementioned *"JexlEngine"* to parse the *"query"* variable and return its boolean value.
- *"IFLSD"* uses an *"IsTrue"* instance to decide whether to execute the first half (the *"then"* block) or the second half (the *"else"* block) of the *"query"*.

Fig. 3.4 illustrates how a *"Statement"* object interacts with our *"Parser"* .

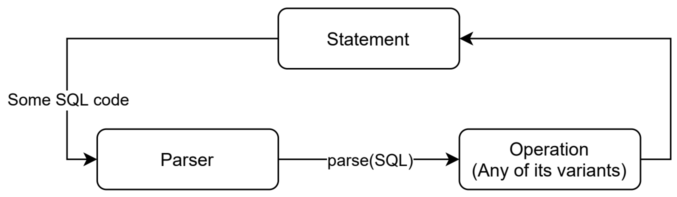


Figure 3.4: How a *"Statement"* interacts with the *"Parser"*

#### 3.2.3.3 JDBC

The main objects that any JDBC application interacts with the most are the various *Statement* objects and their *ResultSet* instances. As previously stated, only the "*LSDStatement*" and "*LSDPreparedStatement*" were implemented.

***LSDStatement,*** keeps track of what is the current "*Operation*" ("*activeOp*"), the current underlying "*Statement*" ("*activeStmt*") and the active "*ResultSet*" ("*activeRS*"). These are necessary because each "*execute*" invocation may have vastly different instructions[6], each with different "*ResultSet*" objects to iterate and/or other properties the JDBC API exposes and, thus, so should we.

Additionally, the primary methods for executing SQL instructions, "*executeQuery*" and "*executeUpdate*", may, instead of sending these instructions to the underlying "*Connection*", only update the "*LSDConnection*" state and possibly return a "*LSDResultSet*".

Figs. **??** and **??** shows the difference between the execution of a normal SQL instruction *versus* the execution of an LSD instruction. As it can be seen, during a normal SQL execution, the database is queried and a normal "*ResultSet*" is returned to the "*Statement*". In case of a LSD instruction, the driver simply returns a future.

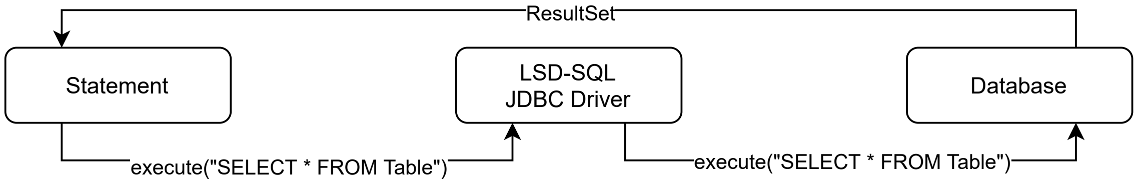

ResultSet
Statement
LSD-SQL
JDBC Driver
Database
execute("SELECT * FROM Table")
execute("SELECT * FROM Table")

Figure 3.5: Normal SQL Statement behaviour

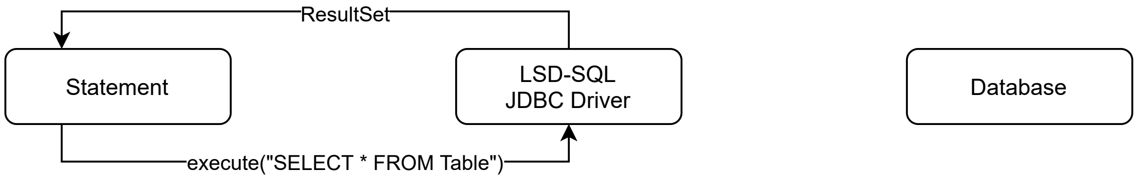


Figure 3.6: LSD Statement behaviour

***LSDPreparedStatement*** is much like a "*LSDStatement*" because, even though its base instruction does not change in the course of its life, its parameters and, therefore, results may change. Its behaviour is equivalent to the one illustrated in Fig. 3.5.

***LSDResultSet,*** as previously stated, serves only as a wrapper for "*LSDFuture*" objects, whose only available method to the application being the "*get*" method, which returns the string representing the future's ID.

[6]The same *LSDStatement* may be used first to execute a *SELECT* query and an *UPDATE* instruction later.

***LSDFuture,*** besides exposing the future ID of the executed query, it also contains a method that is only used internally: *“resolve”*, which executes the query it contains and saves it into a *“result”* variable.

#### 3.2.3.4 Savepoints, Rollbacks and Commits

Some of the more important methods a *“Connection”* exposes to an application are the *”setSavepoint”*, *”rollback”* and *”commit”* methods.

**Savepoints** are fairly simple to implement, as the only requirement from our *“Connection”* is to save the current state. To do this, we make use of two auxiliary variables which store our current state and then invoking the underlying connection’s *“setSavepoint”*, which does most of the necessary work.

**Rollbacks** are also simple, as all that is needed to be done is restore the data that is in the auxiliary variables and then invoking the underlying connection’s *“rollback”* method, similarly to how savepoints function.

**Commit** is more complex, due to having to execute all of the LSD operations that the transaction contains.

Listings 3.12 and 3.13 show the commit protocol[7] in Kotlin.

The first step of the protocol is to lock the entire connection (line 2) to prevent any alterations to its state. Then, each operation in the current transaction is matched to one of the supported *“Operation”* types[8] (lines 4-6):

- *“SelectLSD”* operations update their “future” value in the *readMap* (lines 7-18).
- *“IsTrue”* operations evaluate their expressions and abort execution if found to be false (lines 19-30).
- *“InsertLSD”* or *“UpdateLSD”* operations are sent to the underlying driver for execution (lines 32-37).
- *“IFLSD”* operations evaluate their internal *“IsTrue”* and return the instruction of the correct execution branch to the underlying driver for execution (lines 38-47).

If an operation is not matched to one of these, then an Exception is thrown and the transaction is rolled back (lines 48-53).

After issuing all the necessary instructions to the underlying driver, a *“commit”* is issued for the already processed transaction (line 57) and, assuming nothing goes wrong, the *“LSDConnection”* resets its state (*readMap* and *operations*) in preparation for a new transaction (lines 60-61).

[7]Only the essential excerpts.

[8]All supported operations, before execution, have their “future” parameters resolved. This is done in the *“prepareForExecution”* method (found in lines 68-75) found in Listing 3.13

```
// Lock connection
synchronized(this) {
    // For each operation found in 'operations' linked list
    for (op in operations) {
        // Check which operation type it is
        when (op) {
            is SelectLSD -> {
                // If it is a SelectLSD
                // resolve futures found inside the query
                prepareForExecution(op)
                // fetch future and update its query with
                // all the new values
                val future = readMap[op.future]!!
                future.query = op.activeQuery
                // finally resolve query and assign its
                // value to the future
                future.resolve(this)
            }
            is IsTrue -> {
                // If it is a IsTrue
                // resolve futures found inside the query
                prepareForExecution(op)
                if (!op.evaluate()) {
                    // make sure 'IsTrue' evaluates to TRUE
                    dbConn.rollback()
                    throw LSDException(
                            "IsTrue evaluated to false!"
                        )
                }
            }
            is UpdateLSD, is InsertLSD -> {
                // If it is a UpdateLSD or an InsertLSD
                // resolve futures found inside the query
                prepareForExecution(op)
                dbConn.prepareStatement(op.activeQuery)
                                        .executeUpdate()
            }
```

Listing 3.12: The Commit protocol

```
            is IFLSD -> {
                // If it is a IFLSD
                // resolve futures found inside the query
                prepareForExecution(op)
                // Prepare statement and execute it,
                // IFLSD returns the branch of execution that
                // must be run already parametrized
                val str = parser.convert(op.resolve(this))
                dbConn.prepareStatement(str).executeUpdate()
            }
            else -> {
            // If it is an unsupported operation, rollback
            // and abort, something has gone wrong
                dbConn.rollback()
                throw LSDException("Unrecognized!")
            }
        }
    }

    dbConn.commit() // Issue commit on the connection

    // Reset state in preparation for next transaction
    readMap.clear()
    operations.clear()
}

/**
 * Prepare query by resolving its futures.
 * @param op the operation
 */
private fun prepareForExecution(op: LSDOperation) {
    val pair = parser.resolveFutures(op.activeQuery)
    op.activeQuery = pair.first
    for (i in pair.second.indices) {
        op.parameters[i] = readMap[pair.second[i]]!!.result
    }
    op.prepareForLSDExecution()
}
```

Listing 3.13: The Commit protocol (cont.)

#### 3.2.3.5 Class Diagram

In Fig. 3.7, we can see a simplified version of the class diagram of our driver. A more detailed diagram can be found in Appendix A.

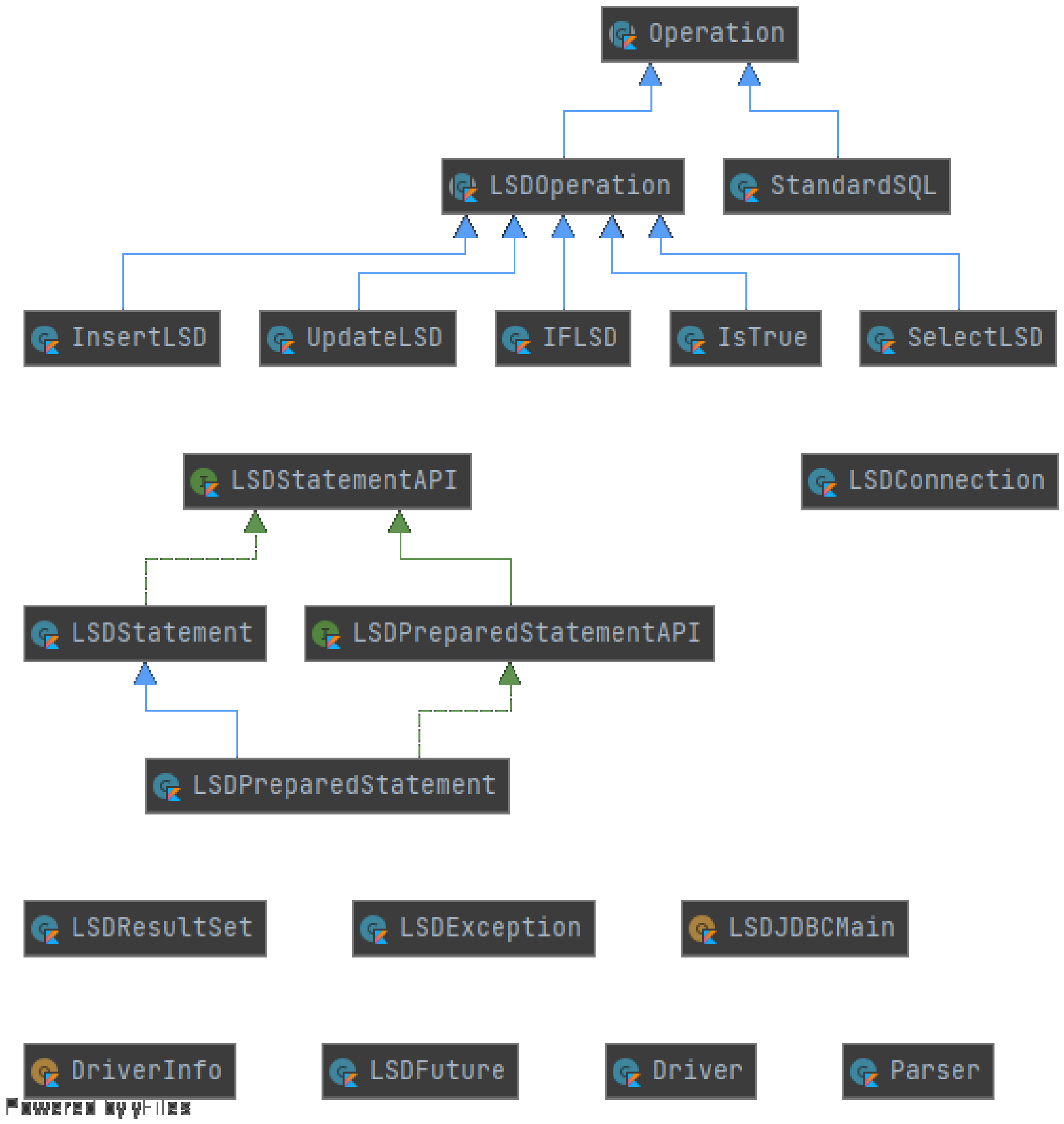


Figure 3.7: LSD-JDBC Driver Simple Class Diagram, made with IntelliJ IDEA's UML plugin[44]

# 4 Validation

In this chapter, we will elaborate on what our testing environments were (Sec. 4.1), which tests we have performed (Sec. 4.2) and discuss our findings (Sec. 4.3).

## 4.1 Testing Environments

The RDBMS we have chosen for these tests is PostgreSQL, for being a Free and Open-Source Software database that is robust and sees extensive and popular use in many areas of computing.

These tests were run (in a distributed fashion) between Nodes 19 through 23 of the *NOVALINCS Cluster*[8], machines whose specifications include (for each node) *2 x AMD Opteron 2376 CPU @ 2.30GHz*, *16GB* of RAM and *2 x 1 Gbps* network connections. One node would host the database while the rest acted as different clients. Additionally, some of these tests were also run locally, in a *Lenovo Thinkpad P1 Gen 1*, with a *Intel Core i7-8750H CPU @ 2.20GHz* and *16GB* of RAM.

The testing candidates are our newly-implemented LSD-JDBC driver and PostgreSQL's JDBC driver (v42.2.16). We will be comparing both of these drivers both in terms of correctness and performance.

## 4.2 Tests

### 4.2.1 Implementation Correctness

The objective of these tests is to make sure that both drivers have the same observable behaviours given some input. In other words, we want to guarantee both drivers return the same answer when given the same question.

The PostgreSQL driver will serve as a baseline, as we assume that it functions correctly. To test the correctness of our solution, we will split these tests into two categories:

- *Parser Tests*, which test if our Parser can correctly identify which kind of operation it is processing and if it can do the required transformations upon the query.

- *Driver Tests*, which runs a given set of queries with both drivers and tests if their output is the same.

#### 4.2.1.1 Parser Tests

The *Parser tests* are simple: instantiate a *Parser* object, give it some input and check if the output corresponds to the expected result.

Our tests can be divided into the following categories:

- *Query Detection*, where some *String* input is given and the *Parser* returns a *Operation*[1].
- *Conversion*, where some *String* representing an LSD instruction is given and its SQL counterpart is expected[2].
- *Future Creation*, where some *String* representing an LSD instruction is given, and a *String* representing the query's future is returned[3].
- *Future Resolution*, where a *String* representing an LSD instruction is given and whose expected output is a set of values which represent the futures that are present within the input query[4].

#### 4.2.1.2 Driver Tests

The *Driver tests'* function is to test the various operations our driver exposes through the JDBC API and comparing their results with the results given by another JDBC driver *(the test driver)* that is assumed to work correctly. Before this can be done, however, three databases must be setup and initialized to the exact same state.

First, it tests if our driver can establish a connection to the first database and if it can execute statements upon said connection.

Afterwards, it prepares a second and third connections, one managed by our driver and the other managed by the aforementioned test driver and prepares a transaction that will be executed by both connections. During this transaction's execution, any values that are returned by the drivers are compared. To assume a transaction was well executed, any values outputted by the drivers must match, and so must the final database states.

These tests are run once with the *'autoCommit'* property set to 'true' and once with it set to 'false'. In other words, two tests are executed with instructions being treated as transactions and two tests are executed with instructions being treated as *part* of a transaction.

Our new LSD driver passed all of our correctness tests.

[1]Example input is 'SELECT * FROM table' with expected output 'StandardSQL'.
[2]Example input is 'SELECT_LSD * FROM table' with expected output 'SELECT * FROM table'.
[3]Example input is 'SELECT_LSD tax FROM products' with expected output 'products_lsd'.
[4]Given a map with the entry ('products_tax', 0.12) and the example input 'UPDATE_LSD tax = {products_tax} FROM products WHERE id = 1', expect *Parser*'s output to be an array containing (0.12).

### 4.2.2 Performance

#### 4.2.2.1 Benchmark

As previously mentioned, we will make use of the TPC-C benchmark, more specifically, a Java implementation of the TPC-C benchmark[22] modified by us to include some LSD transactions.

To optimize our research efforts, we have adjusted the weights of the instructions present in the benchmark to guarantee that only *New-Order* operations are executed.

A single *New-Order* transaction behaves as follows (see Listings 4.1 and 4.2):

1. The *Tax* and the *Next Order ID* are queried and locked (lines 3-6).
2. Some of the customer and warehouse's data is also queried, but not locked (lines 9-12).
3. An increment to the *Next Order ID* is issued (lines 15-17).
4. A new order is inserted into the *Order* table (lines 20-25).
5. A new order is inserted into the *New Order* table (lines 28-30).
6. For each item present in the new order, the transaction:
    a) Inserts a new order line into the respective table (lines 34-40).
    b) Reads and locks the stock of the current item (lines 43-49).
    c) Updates the stock of said item (lines 52-56).
7. The transaction commits (line 59).

We have designed two test scenarios:

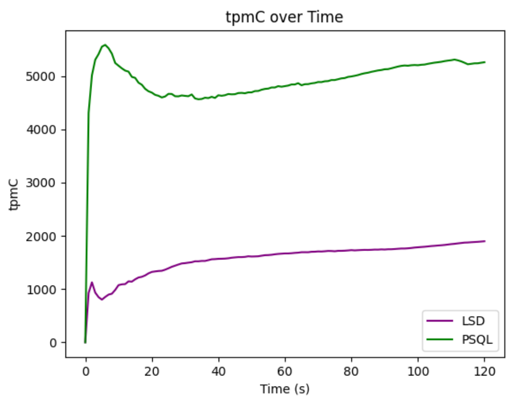
- A database configured with 10 Warehouses that will have to serve 10 Clients, which is a very common benchmark.
- A database configured with 1 Warehouse that will *still* have to serve 10 Clients, which should generate enough conflicts to lower the overall throughput of the system due to transactions aborting.

Each of these scenarios are run for 2 minutes and each benchmark is run 10 times. Fig. 4.1 shows the average tpmC values measured during these tests.

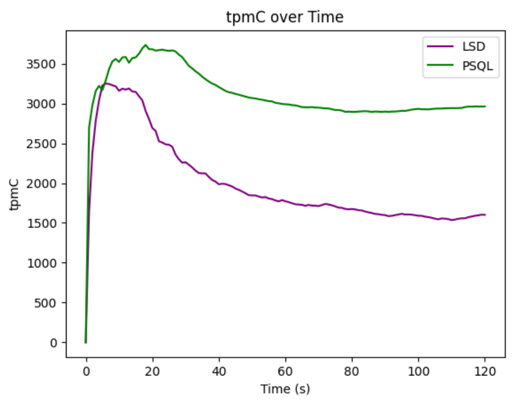
As it can be seen, our driver has lower performance when compared to PostgreSQL's values, even though it seems both drivers follow the same performance trends. Thus, we must analyse why our driver underperforms when compared to PostgreSQL.

```
// Retrieve the required data from DISTRICT

SELECT d_tax, d_next_o_id
    FROM bmsql_district
    WHERE d_w_id = ? AND d_id = ?
    FOR UPDATE

// Retrieve the required data from CUSTOMER and WAREHOUSE
SELECT c_discount, c_last, c_credit, w_tax
    FROM bmsql_customer
    JOIN bmsql_warehouse ON (w_id = c_w_id)
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?

// Update the DISTRICT bumping the D_NEXT_O_ID
UPDATE bmsql_district
    SET d_next_o_id = d_next_o_id + 1
    WHERE d_w_id = ? AND d_id = ?

// Insert the ORDER row
INSERT INTO bmsql_oorder
    (
        o_id, o_d_id, o_w_id, o_c_id, o_entry_d,
        o_ol_cnt, o_all_local
    )
    VALUES (?, ?, ?, ?, ?, ?, ?)

// Insert the NEW_ORDER row
INSERT INTO bmsql_new_order
    (no_o_id, no_d_id, no_w_id)
    VALUES (?, ?, ?)
```

Listing 4.1: The ‘NEW-ORDER’ transaction



(a) 10 Warehouses, 10 Clients



(b) 1 Warehouses, 10 Clients

Figure 4.1: LSD: TPC-C Benchmark Results

```
// For each ORDER_LINE:
    // Insert new ORDER_LINE
    INSERT INTO bmsql_order_line
        (
            ol_o_id, ol_d_id, ol_w_id, ol_number,
            ol_i_id, ol_supply_w_id, ol_quantity,
            ol_amount, ol_dist_info
        )
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)

    // Fetch the current item's stock
    SELECT s_quantity, s_data,
        s_dist_01, s_dist_02, s_dist_03, s_dist_04,
        s_dist_05, s_dist_06, s_dist_07, s_dist_08,
        s_dist_09, s_dist_10
        FROM bmsql_stock
        WHERE s_w_id = ? AND s_i_id = ?
        FOR UPDATE

    // Update the current item's stock
    UPDATE bmsql_stock
        SET s_quantity = ?, s_ytd = s_ytd + ?,
        s_order_cnt = s_order_cnt + 1,
        s_remote_cnt = s_remote_cnt + ?
        WHERE s_w_id = ? AND s_i_id = ?

// Commit the transaction
COMMIT
```

Listing 4.2: The ‘NEW-ORDER’ transaction (cont.)

#### 4.2.2.2 Analysing the Performance

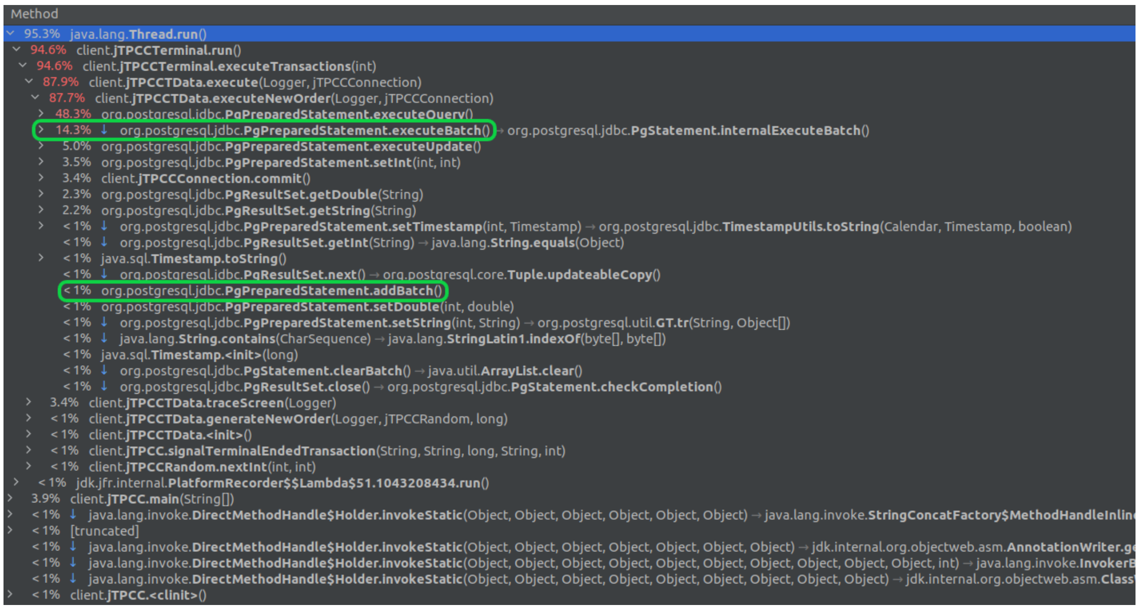
By making use of IntelliJ IDEA’s profiler, we can analyse how much CPU time is being used for each component present in the call stack of each benchmark run. For this analysis, each run consisted in having 10 clients each execute and commit 1000 transactions. Essentially, we are evaluating how each driver deals with the same[5] workload.

Fig. 4.2 shows the call tree of the benchmark run with PostgreSQL’s driver, with the percentage values representing the amount of CPU time a given method spent. Something that is of note here is that the benchmark makes use of *Statement Batches*[6], something that our driver, at the time of writing, does not support, and would be a great boon to reducing our overheads when executing in a networked environment.

Similarly to Fig. 4.2, Fig. 4.3 shows the call stack for the same benchmark making use of the LSD driver. Things of note included here are:

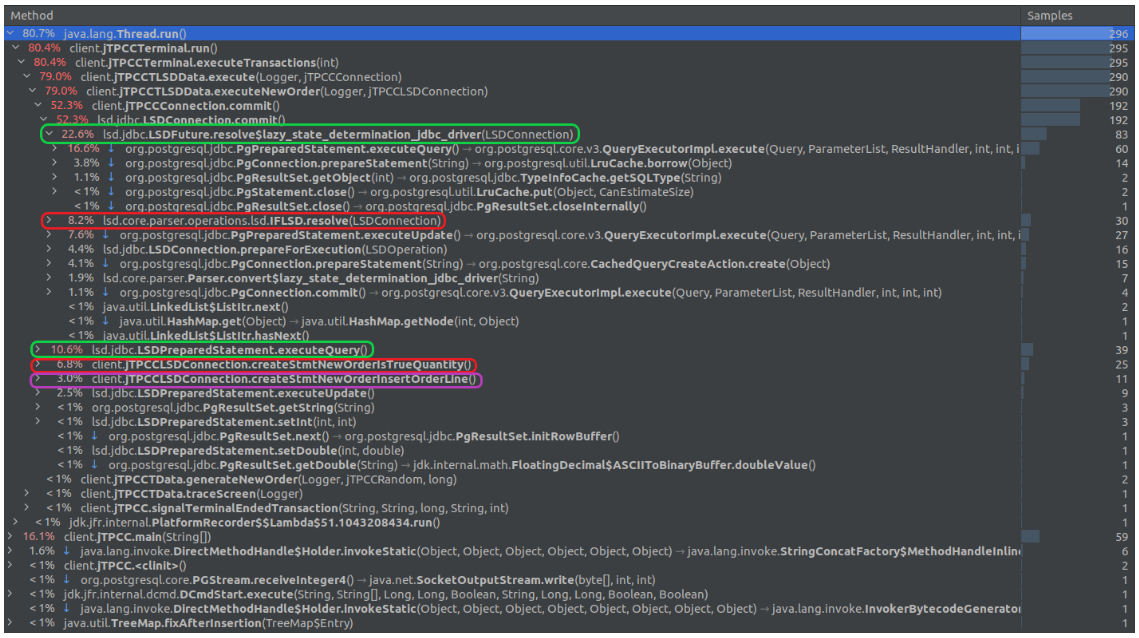
[5]Approximate, due to the randomness of the order generation process.

[6]Denoted by the green bubbles in Fig. 4.2



```
Method
⌄ 95.3%  java.lang.Thread.run()
 ⌄ 94.6%  client.jTPCCTerminal.run()
  ⌄ 94.6%  client.jTPCCTerminal.executeTransactions(int)
   ⌄ 87.9%  client.jTPCCTData.execute(Logger, jTPCCConnection)
    ⌄ 87.7%  client.jTPCCTData.executeNewOrder(Logger, jTPCCConnection)
     › 48.3%  org.postgresql.jdbc.PgPreparedStatement.executeQuery()
     › 14.3% ↓ org.postgresql.jdbc.PgPreparedStatement.executeBatch() → org.postgresql.jdbc.PgStatement.internalExecuteBatch()
     ›  5.0%  org.postgresql.jdbc.PgPreparedStatement.executeUpdate()
     ›  3.5%  org.postgresql.jdbc.PgPreparedStatement.setInt(int, int)
     ›  3.4%  client.jTPCCConnection.commit()
     ›  2.3%  org.postgresql.jdbc.PgResultSet.getDouble(String)
     ›  2.2%  org.postgresql.jdbc.PgResultSet.getString(String)
     ›  < 1% ↓ org.postgresql.jdbc.PgPreparedStatement.setTimestamp(int, Timestamp) → org.postgresql.jdbc.TimestampUtils.toString(Calendar, Timestamp, boolean)
        < 1% ↓ org.postgresql.jdbc.PgResultSet.getInt(String) → java.lang.String.equals(Object)
     ›  < 1%  java.sql.Timestamp.toString()
        < 1% ↓ org.postgresql.jdbc.PgResultSet.next() → org.postgresql.core.Tuple.updateableCopy()
        < 1%  org.postgresql.jdbc.PgPreparedStatement.addBatch()
        < 1%  org.postgresql.jdbc.PgPreparedStatement.setDouble(int, double)
        < 1% ↓ org.postgresql.jdbc.PgPreparedStatement.setString(int, String) → org.postgresql.util.GT.tr(String, Object[])
        < 1% ↓ java.lang.String.contains(CharSequence) → java.lang.StringLatin1.indexOf(byte[], byte[])
        < 1%  java.sql.Timestamp.<init>(long)
        < 1% ↓ org.postgresql.jdbc.PgStatement.clearBatch() → java.util.ArrayList.clear()
        < 1% ↓ org.postgresql.jdbc.PgResultSet.close() → org.postgresql.jdbc.PgStatement.checkCompletion()
    ›  3.4%  client.jTPCCTData.traceScreen(Logger)
    ›  < 1%  client.jTPCCTData.generateNewOrder(Logger, jTPCCRandom, long)
    ›  < 1%  client.jTPCCTData.<init>()
    ›  < 1%  client.jTPCC.signalTerminalEndedTransaction(String, String, long, String, int)
    ›  < 1%  client.jTPCCRandom.nextInt(int, int)
 ›  < 1%  jdk.jfr.internal.PlatformRecorder$$Lambda$51.1043208434.run()
›  3.9%  client.jTPCC.main(String[])
›  < 1% ↓ java.lang.invoke.DirectMethodHandle$Holder.invokeStatic(Object, Object, Object, Object, Object, Object) → java.lang.invoke.StringConcatFactory$MethodHandleInlin
›  < 1%  [truncated]
   < 1% ↓ java.lang.invoke.DirectMethodHandle$Holder.invokeStatic(Object, Object, Object, Object, Object, Object, Object) → jdk.internal.org.objectweb.asm.AnnotationWriter.ge
   < 1% ↓ java.lang.invoke.DirectMethodHandle$Holder.invokeStatic(Object, Object, Object, Object, Object, Object, Object, Object, Object, int) → java.lang.invoke.InvokerB
   < 1% ↓ java.lang.invoke.DirectMethodHandle$Holder.invokeStatic(Object, Object, Object, Object, Object, Object, Object, Object, Object) → jdk.internal.org.objectweb.asm.Class
›  < 1%  client.jTPCC.<clinit>()
```

Figure 4.2: PostgreSQL Profiler Results



```
Method                                                                                                          Samples
⌄ 80.7%  java.lang.Thread.run()                                                                                     296
 ⌄ 80.4%  client.jTPCCTerminal.run()                                                                                295
  ⌄ 80.4%  client.jTPCCTerminal.executeTransactions(int)                                                            295
   ⌄ 79.0%  client.jTPCCTLSDData.execute(Logger, jTPCCConnection)                                                   290
    ⌄ 79.0%  client.jTPCCTLSDData.executeNewOrder(Logger, jTPCCLSDConnection)                                       290
     ⌄ 52.3%  client.jTPCCConnection.commit()                                                                       192
      ⌄ 52.3%  lsd.jdbc.LSDConnection.commit()                                                                      192
       ⌄ 22.6%  lsd.jdbc.LSDFuture.resolve$lazy_state_determination_jdbc_driver(LSDConnection)                       83
        › 16.6% ↓ org.postgresql.jdbc.PgPreparedStatement.executeQuery() → org.postgresql.core.v3.QueryExecutorImpl.execute(Query, ParameterList, ResultHandler, int, int, i   60
        ›  3.8% ↓ org.postgresql.jdbc.PgConnection.prepareStatement(String) → org.postgresql.util.LruCache.borrow(Object)    14
        ›  1.1% ↓ org.postgresql.jdbc.PgResultSet.getObject(int) → org.postgresql.jdbc.TypeInfoCache.getSQLType(String)     2
        ›  < 1% ↓ org.postgresql.jdbc.PgStatement.close() → org.postgresql.util.LruCache.put(Object, CanEstimateSize)       2
           < 1% ↓ org.postgresql.jdbc.PgResultSet.close() → org.postgresql.jdbc.PgResultSet.closeInternally()              1
       ›  8.2%  lsd.core.parser.operations.lsd.IFLSD.resolve(LSDConnection)                                          30
       ›  7.6% ↓ org.postgresql.jdbc.PgPreparedStatement.executeUpdate() → org.postgresql.core.v3.QueryExecutorImpl.execute(Query, ParameterList, ResultHandler, int, int, i   27
       ›  4.4%  lsd.jdbc.LSDConnection.prepareForExecution(LSDOperation)                                             16
       ›  4.1% ↓ org.postgresql.jdbc.PgConnection.prepareStatement(String) → org.postgresql.core.CachedQueryCreateAction.create(Object)   15
       ›  1.9%  lsd.core.parser.Parser.convert$lazy_state_determination_jdbc_driver(String)                            7
       ›  1.1% ↓ org.postgresql.jdbc.PgConnection.commit() → org.postgresql.core.v3.QueryExecutorImpl.execute(Query, ParameterList, ResultHandler, int, int, int)   4
          < 1%  java.util.LinkedList$ListItr.next()                                                                    2
          < 1% ↓ java.util.HashMap.get(Object) → java.util.HashMap.getNode(int, Object)                                1
          < 1%  java.util.LinkedList$ListItr.hasNext()                                                                 1
     › 10.6%  lsd.jdbc.LSDPreparedStatement.executeQuery()                                                           39
     ›  6.8%  client.jTPCCLSDConnection.createStmtNewOrderIsTrueQuantity()                                           25
     ›  3.0%  client.jTPCCLSDConnection.createStmtNewOrderInsertOrderLine()                                          11
     ›  2.5%  lsd.jdbc.LSDPreparedStatement.executeUpdate()                                                           9
     ›  < 1%  org.postgresql.jdbc.PgResultSet.getString(String)                                                       3
     ›  < 1%  lsd.jdbc.LSDPreparedStatement.setInt(int, int)                                                          3
        < 1% ↓ org.postgresql.jdbc.PgResultSet.next() → org.postgresql.jdbc.PgResultSet.initRowBuffer()               1
        < 1%  lsd.jdbc.LSDPreparedStatement.setDouble(int, double)                                                    1
        < 1% ↓ org.postgresql.jdbc.PgResultSet.getDouble(String) → jdk.internal.math.FloatingDecimal$ASCIIToBinaryBuffer.doubleValue()   1
      < 1%  client.jTPCCTData.generateNewOrder(Logger, jTPCCRandom, long)                                             2
   ›  < 1%  client.jTPCCTData.traceScreen(Logger)                                                                     1
   ›  < 1%  client.jTPCC.signalTerminalEndedTransaction(String, String, long, String, int)                            1
 ›  < 1%  jdk.jfr.internal.PlatformRecorder$$Lambda$51.1043208434.run()                                               1
› 16.1%  client.jTPCC.main(String[])                                                                                  59
›  1.6% ↓ java.lang.invoke.DirectMethodHandle$Holder.invokeStatic(Object, Object, Object, Object, Object, Object) → java.lang.invoke.StringConcatFactory$MethodHandleInlin   6
›  < 1%  client.jTPCC.<clinit>()                                                                                      2
   < 1% ↓ org.postgresql.core.PGStream.receiveInteger4() → java.net.SocketOutputStream.write(byte[], int, int)       1
›  < 1%  jdk.jfr.internal.dcmd.DCmdStart.execute(String, String[], Long, Long, Boolean, String, Long, Long, Boolean, Boolean)   1
   < 1% ↓ java.lang.invoke.DirectMethodHandle$Holder.invokeStatic(Object, Object, Object, Object, Object, Object, Object, Object) → java.lang.invoke.InvokerBytecodeGenerato   1
›  < 1%  java.util.TreeMap.fixAfterInsertion(TreeMap$Entry)                                                           1
```

Figure 4.3: LSD Profiler Results

- 19%[7] of the time is used during the *'resolveFutures'* phase, which is when our driver executes queries to the database. This is (at a first glance) better than the previous 46.4% spent by PostgreSQL during its *'executeQuery'* phase, even when accounting for the 9.3% overhead our solution inserts during its own *'executeQuery'* phase.
- (8.5 + 13.3)%[8] of the time is used to prepare and execute *IFLSD* instructions. When considering that these instructions are only required once per *New Order*, this shows that these operations are our biggest bottleneck and must see greater optimization efforts. The necessity for these efforts will be further justified later in this chapter.
- 7.5%[9] of the time was used during the *Insert New Order Line* phase, which is an area that could see further improvements with the previously mentioned implementation of *Statement Batches*.

Of course, this analysis would be naive without taking into account *how long* each benchmark ran for, and so, we have taken note of them and done some further analysis[10]:

Table 4.1: Time elapsed for each run, in seconds

| **PostgreSQL** | **LSD** |
|---|---|
| 70.61 s | 259.15 s |

Table 4.2: LSD Parser and Preparation Overheads

| **Phase** | **Time** |
|---|---|
| Prepare For Execution | 4.5 % |
| Parser.convert | 1.9 % |
| Parser.parse | 1.2 % |
| Total Percentage | 7.60 % |
| Real Time | 19.695 s |

Table 4.1 shows, for each driver, how long each benchmark run took, with its data being taken into consideration when calculating values for the next tables.

Table 4.2 shows the overheads our LSD driver requires when preparing statements compared to PostgreSQL and how it translates into wasted time. As it can be seen, we lose approximately *twenty seconds* of execution to these overheads, which is nearly *one third* of the total execution time for the PostgreSQL driver.

[7]Denoted by the green bubble in Fig. 4.3.

[8]Denoted by the red bubbles.

[9]Denoted by the magenta bubble.

[10]There are slight variations in the percentages for this analysis when compared to the previously shown due to using a more recent dataset.

Table 4.3: Read Phase Analysis

| | PostgreSQL | LSD |
|---|---|---|
| Execute Query | 47 % | 8.7 % |
| Resolve Futures | 0 % | 24.3 % |
| Total Percentage | 47 % | 33 % |
| Real Time | 33.188 s | 85.519 s |

Table 4.4: Write Phase Analysis

| | PostgreSQL | LSD |
|---|---|---|
| Execute Update | 5.2 % | 7.6 % |
| LSD Execute Update | 0 % | 2.5 % |
| Execute Batch | 13.2 % | 0 % |
| IfLSD.resolve | 0 % | 8.2 % |
| Create NewOrder IsTrue | 0 % | 6.8 % |
| Create NewOrder InsertOrderLine | 0 % | 3 % |
| Total Percentage | 18.4 % | 32.7 % |
| Real Time | 12.993 s | 84.742 s |

Tables 4.3 and 4.4 show the percentages of work of each driver for the *'Read'* and *'Write'* phases respectively and how long each phase lasted. We can see that, for the *'Read'* phase, the LSD driver uses too much time resolving all futures, taking nearly twenty-one seconds[11].

#### 4.2.2.3 Performance of Instructions by Section

For this analysis, we are interested in seeing the decay of performance for both benchmarks if we change which types of instructions are executed. In other words, we change the *New Order* transaction to only execute some of its steps (see Listings 4.1 and 4.2).

For this analysis, we consider the default configuration of the benchmark: each run lasts two minutes, and each driver tries to commit as many transactions as possible.

This analysis is useful because we can visually see and confirm which parts of the benchmark are most taxing, performance-wise.

Figures 4.4 through to 4.8 show the performance of each benchmark run up until a certain point of its execution, with Figure 4.9 representing the full benchmark run. Figure 4.10 represents the full benchmark run but only executing its queries, no updates

[11] 24.3% of 85.519s = 20.781s

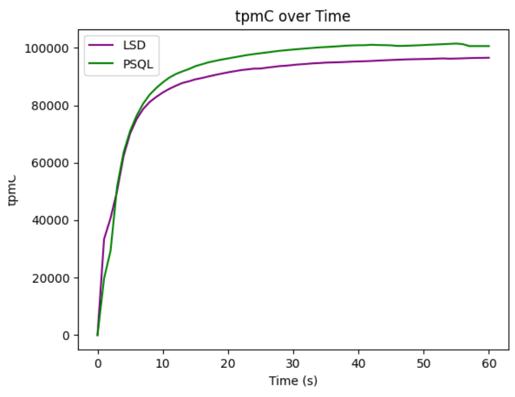


Figure 4.4: Executing up to the *'Tax'* step (including)

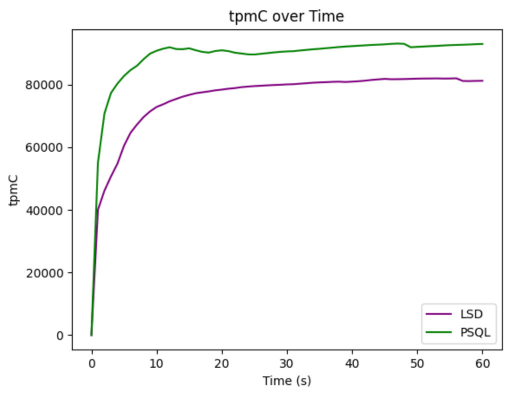


Figure 4.5: Executing up to the *'Warehouse'* step (including)

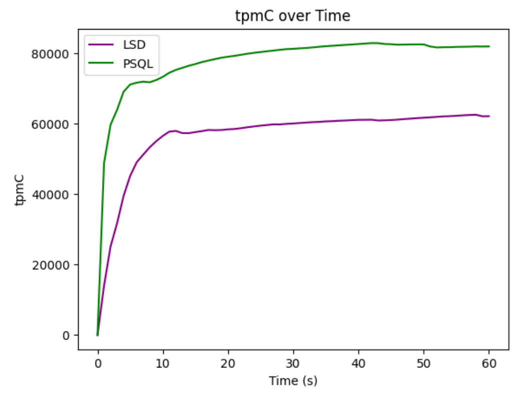


Figure 4.6: Executing up to the *'Update New Order ID'* step (including)

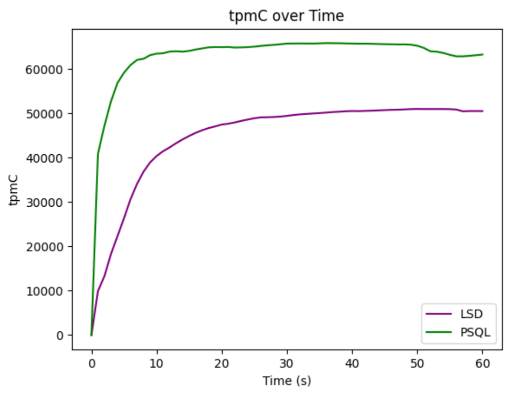


Figure 4.7: Executing up to the *'Insert Order'* step (including)

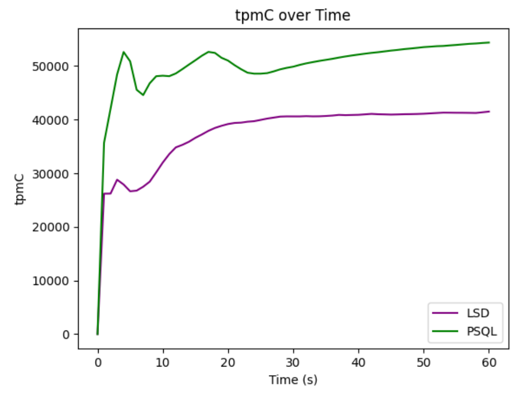


Figure 4.8: Executing up to the *'Insert New Order'* step (including)

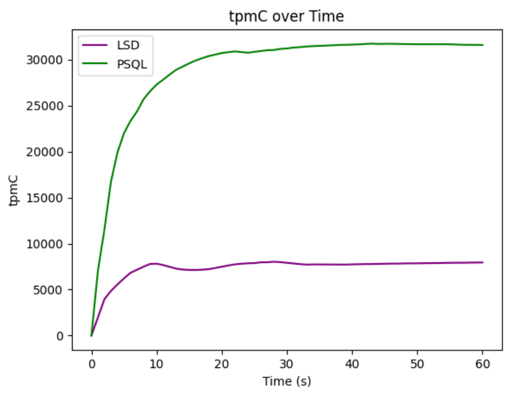


Figure 4.9: Normal and complete benchmark execution

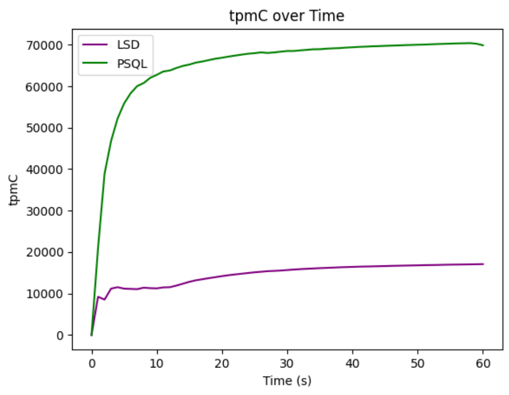


Figure 4.10: Executing benchmark, but only queries

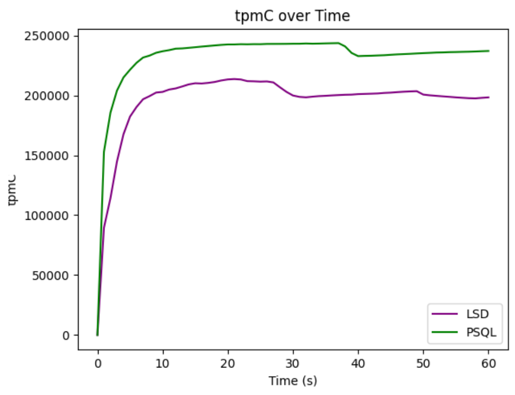


Figure 4.11: Executing up to the *'Insert New Order Line'* step (excluding)

are sent to the database, and Figure 4.11 shows the performance of a benchmark run fully executed except the final loop which can be found in lines 34-56 in Listing 4.2.

As it can be seen, the LSD performance takes its biggest hit between Figures 4.8 and 4.9. This points us to believing our lack of statement batching is the biggest culprit for the lack of performance. Other performance dips seen in the other figures can be attributed to our overheads, which have been discussed previously.

## 4.3 Discussion

Armed with the previous analyses, we can derive some conclusions about our lack of performance.

First and foremost, reducing the impact of our overheads would be helpful, and it can be done by improving our *Parser*, since all of the three main overheads are dependant on it. Its costly performance is due to an inherent dependency in the usage of Java's *'String'* library, which is used to parse and manipulate the instructions received by the application at various points.

Additionally, implementing the batching of statements in a similar fashion to what PostgreSQL employs would greatly help in networked environments.

Secondly, we can clearly see that the creation and execution of *'IF_LSD'* instructions is too taxing. Recalling Section 3.2.3.2 in page 37, this instruction carries with it at all times three sub-instructions:

- The *'IS-TRUE'* instruction.
- The instruction to be executed if the condition is found to be true.
- The analogous if the condition is false.

Keeping in mind that each of these instructions must be evaluated by our *Parser*, since any of them can have futures and parameters that must be inserted into the statement, we can see how costly this single instruction can be.

Further discussion of improvements can be found in Section 5.2.

# 5 Conclusions

In this chapter, we will give some final considerations about what we have accomplished so far in Section 5.1 and what can be done in the future of this project in Section 5.2.

## 5.1 Final Considerations

When we began this project, we had set to:

- Do a detailed evaluation of the *State of the Art*, which we have produced in Chapter 2.
- A proposal for the LSD-SQL API, which we have presented in Chapter 3 and used for the first prototype of the LSD-JDBC driver.
- A prototype database-agnostic JDBC driver which makes use of LSD, which we have developed for this project.
- A correctness and performance evaluation of our newly developed driver vs a standard PostgreSQL driver, which was discussed in Chapter 4.

Out of all of these goals, only the performance was disappointing, as we have found from our different analyses, with the LSD driver being capable of approximately one fourth of the throughput of the PostgreSQL driver. However, we still believe that what we have developed holds some promise, and we have new ideas on how to improve our prototype.

## 5.2 Future Work

Some things that have been briefly mentioned in Section 4.3 can definitely be done:

- Improving our *Parser* by reducing its dependency on the *'String'* library. There are two ways to achieve this goal:

  - The *Parser* module can be done outside of the driver. Ideally, its functions could be implemented into a database's own *Parser* subsystem *but* this solution is very complex and would need to be developed for each database.
  - Alternatively, re-implementing the *Parser* subsystem to make use of an *Abstract Syntax Tree*[1] should greatly reduce our overheads at the expense of extra memory usage.

- Removing the *IF_LSD* instruction and, instead, changing how our *"IS-TRUE"* works:
  - First, the application should execute a *"IS-TRUE"*. This operation would be executed by our driver on a separate JDBC connection immediately, and it should return whether the condition this operation holds is true or false. With this information, the application can correctly execute the desired branch of operations.
  - Finally, during the *'Commit'* phase, the *'IS-TRUE'* is executed again to check if its Boolean value remains the same and, if it does not, the transaction can be aborted.
- Implementing batching of statements. To do this, the driver should keep track of each operation the application adds to the batch into a data structure. Then, during the *'Commit'* phase, each of these "proto-batches" should be transformed into batches created by the database JDBC connection.

Additionally to these improvements, our *'Commit'* protocol can be further optimized. By making use of concurrency, we can *first* update the *'readMap'* in parallel and *then* execute the remaining operations also in parallel.

Updating the *'readMap'* in parallel is possible because there are no *'Read After Write'* dependencies inside the transaction. This, in turn, enables the execution of the remaining operations in parallel as well, since all of its correct and final values have already been read into the *'readMap'*.

# Bibliography

[1] *Abstract Syntax Tree - Free On-line Dictionary of Computing*. URL: http://foldoc.org/abstract+syntax+tree. (accessed: 22.06.2021) (cit. on p. 55).

[2] M. Armbrust et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58 (cit. on p. 1).

[3] H. Berenson et al. "A critique of ANSI SQL isolation levels". In: *ACM SIGMOD Record* 24.2 (1995), pp. 1–10 (cit. on p. 7).

[4] A. Castillo O'Sullivan and A. D. Thierer. "Projecting the growth and economic impact of the internet of things". In: *Available at SSRN 2618794* (2015) (cit. on p. 1).

[5] *Class JexlEngine*. URL: https://commons.apache.org/proper/commons-jexl/apidocs/org/apache/commons/jexl3/JexlEngine.html. (accessed: 20.03.2021) (cit. on p. 37).

[6] E. F. Codd. "A relational model of data for large shared data banks". In: *Software pioneers*. Springer, 2002, pp. 263–294 (cit. on p. 12).

[7] F. Dabek et al. "Event-driven programming for robust software". In: *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. 2002, pp. 186–189 (cit. on p. 5).

[8] *DI-Cluster*. URL: https://cluster.di.fct.unl.pt. (accessed: 21.05.2021) (cit. on p. 43).

[9] R. J. Dias, J. M. Lourenço, and N. M. Preguiça. "Efficient and correct transactional memory programs combining snapshot isolation and static analysis". In: *Proceedings of the 3rd USENIX conference on Hot topics in parallelism (HotPar'11), HotPar*. Vol. 11. 2011 (cit. on p. 20).

[10] R. J. Dias et al. "StarTM: Automatic Verification of Snapshot Isolation in Transactional Memory Java Programs". In: (2011) (cit. on p. 20).

[11] R. J. Dias et al. “Verification of snapshot isolation in transactional memory Java programs”. In: *European Conference on Object-Oriented Programming*. Springer. 2012, pp. 640–664 (cit. on p. 20).

[12] E. W. Dijkstra. “Solution of a problem in concurrent programming control”. In: *Pioneers and Their Contributions to Software Engineering*. Springer, 2001, pp. 289–294 (cit. on p. 4).

[13] D. Distefano, P. W. O’hearn, and H. Yang. “A local shape analysis based on separation logic”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2006, pp. 287–302 (cit. on p. 20).

[14] A. Dragojević et al. “FaRM: Fast remote memory”. In: *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 2014, pp. 401–414 (cit. on p. 20).

[15] K. P. Eswaran et al. “The notions of consistency and predicate locks in a database system”. In: *Communications of the ACM* 19.11 (1976), pp. 624–633 (cit. on p. 8).

[16] R. Filipe et al. “Stretching the capacity of Hardware Transactional Memory in IBM POWER architectures”. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 2019, pp. 107–119 (cit. on p. 21).

[17] D. P. Friedman and D. S. Wise. *The impact of applicative programming on multiprocessing*. Indiana University, Computer Science Department, 1976 (cit. on p. 5).

[18] J. Gray et al. “The transaction concept: Virtues and limitations”. In: *VLDB*. Vol. 81. 1981, pp. 144–154 (cit. on p. 6).

[19] C. Hewitt et al. “Actor induction and meta-evaluation”. In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1973, pp. 153–168 (cit. on p. 5).

[20] *JDBC Basics*. URL: `https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html`. (accessed: 29.07.2020) (cit. on p. 12).

[21] J. Jones. “Abstract syntax tree implementation idioms”. In: *Proceedings of the 10th conference on pattern languages of programs (plop2003)*. 2003, pp. 1–10 (cit. on p. 34).

[22] *jTPCC*. URL: `https://github.com/petergeoghegan/benchmarksql`. (accessed: 21.05.2021) (cit. on p. 45).

[23] S. Kabangu. “Benchmarking Databases”. In: (2009) (cit. on p. 14).

[24] D. E. Knuth. *The art of computer programming*. Vol. 3. Pearson Education, 1997 (cit. on p. 5).

[25] H.-T. Kung and J. T. Robinson. “On optimistic methods for concurrency control”. In: *ACM Transactions on Database Systems (TODS)* 6.2 (1981), pp. 213–226 (cit. on p. 9).

[26] L. Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 179–196 (cit. on p. 4).

[27] H. Q. Le et al. “Transactional memory support in the IBM POWER8 processor”. In: *IBM Journal of Research and Development* 59.1 (2015), pp. 8–1 (cit. on p. 21).

[28] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012 (cit. on p. 4).

[29] J. Melton and A. R. Simon. *SQL: 1999: understanding relational language components*. Elsevier, 2001 (cit. on p. 6).

[30] J. Melton and A. R. Simon. *Understanding the new SQL: a complete guide*. Morgan Kaufmann, 1993 (cit. on p. 12).

[31] J. Milton. “Database Language SQL Part 2: Foundation (SQL/Foundation)”. In: *ISO ISO/IEC* (1999), pp. 9075–2 (cit. on p. 12).

[32] S. Mu et al. “Extracting more concurrency from distributed transactions”. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 479–494 (cit. on p. 18).

[33] T. Nakaike et al. “Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8”. In: *ACM SIGARCH Computer Architecture News* 43.3S (2015), pp. 144–157 (cit. on p. 21).

[34] G. Prasaad, A. Cheung, and D. Suciu. “Improving High Contention OLTP Performance via Transaction Scheduling”. In: *arXiv preprint arXiv:1810.01997* (2018) (cit. on p. 18).

[35] D. Pritchett. “Base: An acid alternative”. In: *Queue* 6.3 (2008), pp. 48–55 (cit. on p. 19).

[36] J. C. Reynolds. “Separation logic: A logic for shared mutable data structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74 (cit. on p. 20).

[37] A. Shamis et al. “Fast general distributed transactions with opacity”. In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 433–448 (cit. on p. 20).

[38] *The “java.sql” package*. URL: https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html. (accessed: 09.03.2021) (cit. on p. 26).

[39] A. Thomasian. “Concurrency control: methods, performance, and analysis”. In: *ACM Computing Surveys (CSUR)* 30.1 (1998), pp. 70–119 (cit. on p. 8).

[40] *TPC-C OLTP Benchmark*. URL: https://tpc.org/tpcc/. (accessed: 08.07.2020) (cit. on p. 2).

[41] *TPC-C OLTP Benchmark*. URL: http://www.tpc.org. (accessed: 04.03.2021) (cit. on p. 14).

[42] *TPC-C OLTP Benchmark Specifications*. URL: http://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp. (accessed: 04.03.2021) (cit. on pp. 15, 16).

[43] S. Tu et al. “Speedy transactions in multicore in-memory databases”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 18–32 (cit. on p. 16).

[44] *UML class diagrams*. URL: https://www.jetbrains.com/help/idea/class-diagram.html. (accessed: 16.03.2021) (cit. on pp. 42, 61).

[45] T. M. Vale et al. *Lazy State Determination: More concurrency for contending linearizable transactions*. 2020. arXiv: 2007.09733 [cs.DC] (cit. on pp. 2, 21, 24).

[46] *What Is ODBC?* URL: https://docs.microsoft.com/en-us/sql/odbc/reference/what-is-odbc?view=sql-server-ver15. (accessed: 29.07.2020) (cit. on p. 14).

[47] D. Wischik, M. Handley, and M. B. Braun. “The resource pooling principle”. In: *ACM SIGCOMM Computer Communication Review* 38.5 (2008), pp. 47–52 (cit. on p. 1).

[48] Y. Wu et al. “An empirical evaluation of in-memory multi-version concurrency control”. In: *Proceedings of the VLDB Endowment* 10.7 (2017), pp. 781–792 (cit. on pp. 9, 11).

[49] C. Xie et al. “High-performance ACID via modular concurrency control”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015, pp. 279–294 (cit. on p. 19).

[50] C. Xie et al. “Salt: Combining {ACID} and {BASE} in a Distributed Database”. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 495–509 (cit. on p. 19).

[51] X. Yu et al. “Tictoc: Time traveling optimistic concurrency control”. In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 1629–1642 (cit. on p. 17).

A

# Detailed Class Diagram for the LSD JDBC Driver

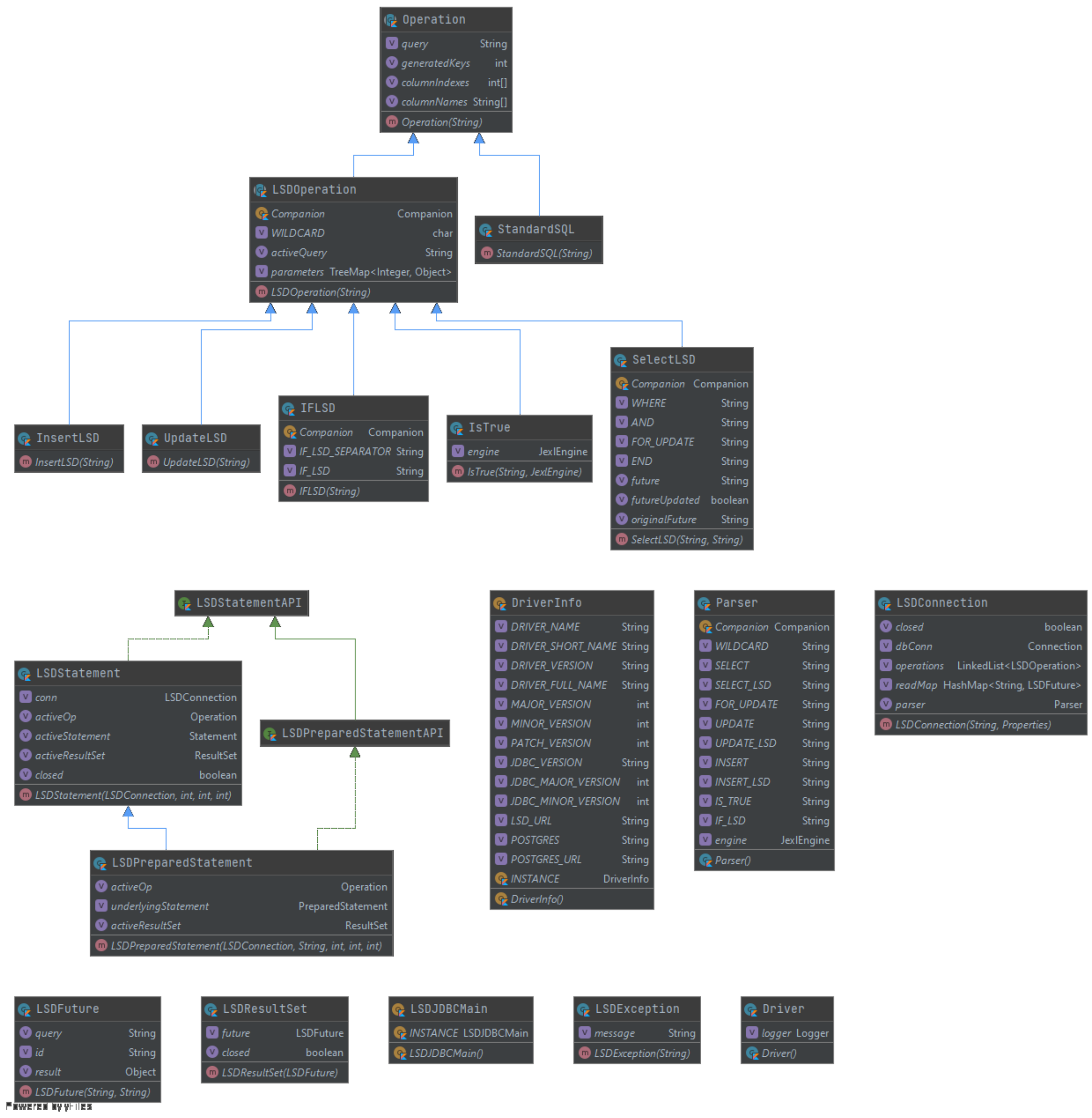


Figure A.1: LSD-JDBC Driver Detailed Class Diagram, made with IntelliJ IDEA's UML plugin[44]

2021 Eduardo Subtil LAZY STATE DETERMINATION FOR SQL DATABASES