




# Empowering a Relational Database with LSD: Lazy State Determination\*

Thales Parreira<sup>1</sup>, Tiago M. Vale<sup>1</sup>, Ricardo J. Dias<sup>1</sup>, and João M. Lourenço<sup>1</sup>

FCT—NOVA University Lisbon & NOVA LINCS, Portugal  
tv.parreira@campus.fct.unl.pt      joao.lourenco@fct.unl.pt

**Abstract.** Computing systems providing services to the final user, more often than not, involve some type of interaction with a database. It is of utmost importance that such systems are responsive, automatically adapting to different types of workload. When this does not happen, the service latency increases with considerable impact in the provided quality of service. In this paper, we propose the lazy evaluation of database SQL queries (using Futures/Promises and *JDBC*<sup>1</sup>) by empowering a relational database with Lazy State Determination (LSD). We observed that the use of Futures and LSD can improve the stability of the system.

**Keywords:** Concurrency · Relational Databases · JDBC · Futures/Promises · Concurrency Control.

## 1 Introduction

Computer systems are ubiquitous in today's world and Relational Database Management Systems (RDBMS) still remain one of the most popular database systems. These systems may have to process millions of operations in milliseconds with databases taking a massive role in the whole systems' throughput. Hence, it should be clear that efficient query processing in these databases is needed and, consequently, fast execution of transactions is a highly desirable property of any modern system. It is with this in mind that we propose empowering a relational database with Lazy State Determination (LSD) [5] by using Futures/Promises and *JDBC*. In the following sections we detail the underlying issue we will focus on and how using futures should improve a database performance.

### 1.1 Problem

The transaction abstraction in RDBMS is ACID abstraction, where transactions appear to execute atomically and without interference, despite being executed

---

\* Supported by Research Grants PTDC/CCI-COM/32456/2017 & LISBOA-01-0145-FEDER-032456.

<sup>1</sup> <https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/>

concurrently. A transaction that involves transferring money from one bank account to another will most likely involve multiple records that do not get updated atomically, even though the ACID abstraction entails that. Correctness here is guaranteed by the Concurrency Control (CC) system in the database, that allows transactions to run in parallel without them being aware of each other. The CC however, when met with high contention, loses performance, mainly due to the fact that conflicting transactions tend to execute sequentially, worsening the resource usage and lowering the system’s throughput.

Typical CC systems can only make conservative assumptions on what a transaction is doing. If two transactions conflict, they will most likely need their execution to be synchronized. For example, two transactions that increase the available balance of the same bank account would conflict because they would be writing to the same database tuple. To control such scenario, transactions are required to expose their concrete state to other transactions which, in the end, leaves them open to conflict. This, however, needs not always be the case as, provided that the aggregate effects of both transactions are preserved, we could view the transactions as non-conflicting.

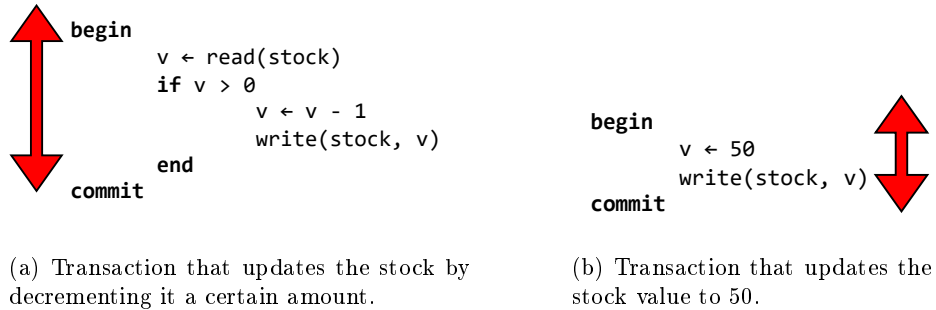


Fig. 1: Two transactions and their windows represented by the red arrows.

To better understanding this issue, we may think about a transaction as having an execution window, which starts when the transaction makes its first query or update to the database and ends when the transaction commits the transaction. (see Figure 1). The conflict window between two transactions is defined by the intersection of their execution windows. There is a possible conflict when such conflict window is not empty. From this observation results that, if we decrease the size of a transaction windows we also decrease the probability of a non-null conflicting window, which will increase the probability of successful commits and improve database throughput. It is based on this insight that we propose the lazy evaluation of database SQL queries leveraging the insights of Vale [5] using futures and JDBC.

## 2 Lazy State Determination

Lazy State Determination (LSD) [5] presents itself as an innovative mean to increasing concurrency of transactions. The LSD is an API for conveying semantics to the database while improving concurrency and ensuring serializability of operations. For this, LSD adapts the traditional transactional API to be lazily evaluated and changes its operations to return *futures* instead of concrete values. Each future is then resolved, one by one, at commit time, which consequently reduces the conflict window of transactions and increases the system's throughput. The commit time in the end should not be the same, as any computation the client needed to make before commit, will long have been computed.

### 2.1 Futures

A *future*, also known as *promise*, is an abstraction used to refer to a proxy of a value that, as the name implies, will only be available at a future time. In other words, *futures* encapsulate asynchronous tasks and their result. In doing so, they provide means for deferring computations, allowing programs to create a *future* and only retrieving its result when necessary. It is based on this abstraction that LSD transactions execute.

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

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

### 2.2 The LSD API

The traditional transactional API is evaluated immediately and so, returns concrete results, which is not compatible with LSD requirements of returning lazy objects, i.e., futures. This forces LSD to have its own API for transactions. Hence, clients that wish to reap the benefits of lazy evaluation must use a LSD specific API. It is worth noting that if the client still needs to use the traditional API, the benefits of delayed execution and short isolation periods will not be present in such operations.

LSD operations are implemented as follows:

`READ(key) → □` — Returns the future `□`, which is an opaque representation of the actual value held by `key`. From the database perspective the future is represented by some function to obtain the value. From the client perspective it only knows the opaque representation of the future. Hence, other interactions with the database that need value in `□` need to use it as is, and it is up to the database to decide when and how to resolve it. The database on the other hand promises to lazily resolve it when necessary.

- READ( $\Delta$ )**  $\rightarrow$   $\square$  — Returns a future  $\square$  held by future key  $\Delta$ . This operation is similar to **READ(key)**  $\rightarrow$   $\square$ , however, in this case,  $\Delta$  future value needs to be resolved before returning future  $\square$ . This is necessary to avoid dealing with “futures of futures”.
- WRITE(key,  $\square$ )** — Writes the future value  $\square$  to value held by key. From the application perspective, it is as if the operation has executed in the database and the value has been updated. On the other hand, from the database perspective this is an indication of an operation to be eventually executed, not necessarily an immediate execution. The value of  $\square$  will only be resolved in the commit phase and, hence, the write operation will only take place at commit time as well.
- WRITE( $\Delta$ ,  $\square$ )** — Writes the future value  $\square$  to the value held by future key  $\Delta$ . This operation is similar to **WRITE(key,  $\square$ )**, however in this case the key is also a future, and, consequently,  $\Delta$  needs to also be resolved in the commit phase (before  $\square$ ).
- IS-TRUE( $\square$ )**  $\rightarrow$  **boolean** — Conditional check for future  $\square$ . This operation allows transactions to operate based on abstract database states. As was the case for other LSD transactions, it does not expose the database state to the transaction. The idea here is to expose an abstract state instead of the concrete state. This allows for more concurrency and safeguards isolation because this operation is only executed in the commit phase, after  $\square$  is resolved.
- COMMIT**  $\rightarrow$  **boolean** — Resolves all futures generated along the transaction execution and, if the transaction validation succeeds, commits to the database.

### 3 JDBC API

The JDBC is an API that allows Java applications to interact with database systems. The API defines a set of Java interfaces that encapsulate database functionality such as executing database queries, updating or inserting data, or even managing configuration information. To any applications that needs to interact with the database, only this high level interface is provided, while internally the specific database driver performs database-specific translations from the high-level interface to the database system. This separation (high-level JDBC API vs. database driver) makes the JDBC API vendor-agnostic, allowing multiple databases to be accessible via a single common interface, and even a single application.

The most relevant interfaces and operations of the JDBC API are the following:

- DriverManager** — Used by the application to create and retrieve the desired JDBC driver.
- Driver** — Used by the application to create connections to a database.
- Connection** — This interface represents the connection to a database.
- prepareStatement(sql: String): PreparedStatement** — Creates a query object represented by an instance of a **PreparedStatement**.

**PreparedStatement** — Represents a SQL statement that can be used to execute some operation in the database. It holds parameters that are to be used in the query and also has the ability to batch multiple update/insert queries which result in a single hop to the database.

**setInt(parameterIndex: Int, value: Int)** — Example of a parameter setter in the **PreparedStatement** API. In this specific example the operation adds an integer parameter to the **PreparedStatement** in the position **parameterIndex**. This parameter is then added to SQL statement when executing the statement in the database. Other setters exist for different types of data.

**executeQuery()** — Executes a query in the database and returns a **ResultSet** with the query results.

**executeUpdate()** — Executes an update or insert in the database and returns a integer representing the result of the operation.

**ResultSet** — Representation of the results that are returned by the database on a SQL query to the database. The results are represented in tabular form with a cursor position maintained pointed to the current row data.

**getInt(columnLabel: String): Int** — Example of a value getter from the **ResultSet**. In this specific example the operation returns the currently positioned item in the **ResultSet** from the column with label **columnLabel**. Other getters exist for different types of data.

The use of this interface enables to execute operations in the database and process its results. Listing 1.1 has an example of a transaction that decrements the stock number of an item using JDBC in *Kotlin*<sup>2</sup>. The first step (line 1) is to create a connection using the **DriverManager**, which internally uses the database specific **Driver** to create the connection. The connection is then used to create a **PreparedStatement** (lines 2–4) that is used to retrieve the stock of item with **id** of 1. The **PreparedStatement** is then executed (line 6) to get the stock value via a **ResultSet**. The **ResultSet** is then used as a parameter to a final update query that decrements the stock value of item with **id** of 1 (lines 8–14).

Listing 1.1: Transaction for update the stock number of item 1 in the a database using JDBC API.

```

1  val connection = DriverManager
2      .getConnection("jdbc:postgresql://localhost/database")
3  val stockStatement = connection.prepareStatement("SELECT stock
4      FROM items
5      WHERE id = 1 FOR UPDATE")
6
7  val resultSet = nextIdStatement.executeQuery()
8
9  val updateStock = connection.prepareStatement("UPDATE items
10     SET stock = ? - 1
11     WHERE id = 1")
12
13  updateStock.setInt(1, resultSet.getInt(1))
14

```

<sup>2</sup> <https://kotlinlang.org/>

```

15 updateStock.executeUpdate()
16
17 connection.commit()

```

## 4 JDBC on LSD

Empowering the database with lazy evaluation requires changes to the JDBC API, such as, needing to understand futures, working with future parameters and, finally, knowing how and when to resolve the future (at commit time). This means that the previously presented interface must change, namely, the `Driver` has to be re-written to know how to support the creation of LSD connections; the `Connection` needs to change to hold futures and at the same time resolve them, one by one, at commit time; the `PreparedStatement` must know how to resolve its future parameters and at the same time know to resolve itself; and the `ResultSet` must be prepared to return future values.

We aimed at enabling all SQL operations supported by JDBC. To demonstrated that, listing 1.1 and listing 1.2 present the same transaction with in JDBC and JDBC with LSD. Over the course of the next sections we will go into more detail over each of the steps of the example, but when comparing both listings it is easy to see the ease of use of this API for programmers, as it is very similar to the standard JDBC.

Listing 1.2: Transaction for updating the stock number of item 1 in a database, using the LSD API.

```

1  val connection = DriverManager
2      .getConnection("jdbc:lsd.v2:postgresql://localhost/database")
3  val stockStatement = connection.prepareStatement("SELECT stock
4      FROM items
5      WHERE id = 1 FOR UPDATE")
6
7  val futureResultSet = stockStatement.executeFutureQuery()
8
9  val updateStock = connection.prepareStatement("UPDATE items
10     SET stock = ? - 1
11     WHERE id = 1")
12
13  val futureStockValue = futureResultSet.getFutureInt(1)
14
15  updateStock.setFutureInt(1, futureStockValue)
16
17  updateStock.executeFutureUpdate()
18
19  connection.commit()

```

### 4.1 Driver

The LSD JDBC driver is a database agnostic driver that proxies operations to a backing real JDBC SQL driver. This means that every component of the LSD driver leverages on a real database specific component, and frees the LSD driver from re-implementing the database specific operations. The LSD driver introduces the `lsd.v2` directive into the scheme component of the database connection URI. The LSD driver after being selected by the `DriverManager` proceeds

to remove the `lsd.v2` directive and then searches for a real backing driver. In the Listing 1.2, the connection `jdbc:lsd.v2:postgresql://localhost/database` is transformed into `jdbc:postgresql://localhost/database`, which is what is then used internally by the LSD driver to create a real connection to the database.

**Futures** To represent futures in the LSD API we propose the interface presented in Listing 1.3. This interface has a single method *resolve*, which is responsible for resolving and returning the actual value of the future.

The future representation in Listing 1.3 provides a clear interface for the connection when interacting with the future, while also enabling the addition of a new operation in the LSD connection for creating futures statements. New interfaces in the LSD API rely heavily on this interface and are what enable the driver to operate over a future state. The method `resolve` computes the actual value of the future while `dispose` is used to clear any unnecessary info after future resolution and database commit.

Listing 1.3: The representation of a Future in Kotlin.

```

1 interface Future<T> {
2     fun resolve(): T
3     fun dispose()
4 }
```

**Connection** Having a lazy state means that the connection must hold future statements and must have the ability to resolve them. We achieved this by modifying the connection to create future statements, and store them local to the connection. These statements are held for execution until the commit operation itself is executed. When it is time to commit, the LSD connection will resolve all futures it holds and only commit the transaction if it succeeds to resolve all futures. If it fails to resolve any future, the transaction fails and aborts.

The operation for creating future statements is presented in Figure 2. The parameter and return value are very similar to the original JDBC API, however, a new data type is returned which implements the already presented future interface (see Listing 1.3).

```

prepareStatement(sql: String): PreparedStatement
      ↓
prepareFutureStatement(sql: String): PreparedFutureStatement
```

Fig 2: New `prepareFutureStatement` operation added to the LSD connection for creating Future statements.

**Future Statements** Using parameters in statements is a major functionality of the JDBC API. This is also supported by the new `PreparedFutureStatement` interface. However, working with futures opens up the possibility that one of such parameters is also a future. This is the exact case of line 14 of the Listing 1.2. Hence, it is important that the `PreparedFutureStatement` interface also supports future parameters. This is achieved by introducing new parameter setter operations that take a future interface as parameter. Figure 3 shows an example of a setter for a future integer value.

```
setFutureInt(parameterIndex: Int, value: Future<Int>)
```

Fig. 3: Example of a new operation for setting future integer parameters in the `prepareFutureStatement` interface. The actual value will only be resolved at commit time.

Statement execution is no longer the same when working in the future state. For this, we add two new methods to the `PreparedFutureStatement` that represent an eventual future execution. These methods are very similar to the ones already existing in the JDBC API, and are as follows:

`executeFutureQuery()` — Creates a future internal to the statement that will execute the JDBC operation `executeQuery()`. This operation returns a `FutureResultSet` which contains a reference to the statement that created it.

`executeFutureUpdate()` — Creates a future internal to the statement that will execute the JDBC operation `executeUpdate()`.

As previously mentioned, a `PreparedFutureStatement` is itself a future, and because of that, must implement the *resolve* operation. In this case, the resolve operation works by resolving any reference to a future parameter it holds and then by executing the internal future that gets created when the `executeFutureQuery()` or `executeFutureUpdate` is executed.

**Future Results** Since statements are no longer executed immediately, a replacement for the `ResultSet` must exist. This is addressed by having a `FutureResultSet`. This new interface has a reference to the `PreparedFutureStatement` that created it and knows how to resolve the statement in order to return values from a `ResultSet`.

The implementation of this new interface serves as a proxy to a real `ResultSet` that has not yet been resolved. Standard operations of the `ResultSet` are no longer supported and are replaced with operations that return future values that comply with the interface presented in Listing 1.3. This is exactly what happens in the line 12 of Listing 1.2. In that case, a future of an integer value is returned. This is achieved by having the operation `getFutureInt` create a future that will first resolve the `PreparedFutureStatement`, access the statement's `ResultSet` and then return the actual value.



**Conditions** The code is not always as simple as the one presented in Listings 1.2. It may become necessary to execute some operation if a certain condition is met. In the example of Listing 1.2, we may want to restock the item if it is below a certain threshold. This is also supported by the LSD driver, and is done with the LSD connection operation presented in Figure 4. The operation gets a condition as parameter, with its syntax being the same as the one used to the `WHERE` clause of any SQL statement. The return interface of the operation is a `FutureStatementCondition` which is a representation of a real SQL statement with only the condition in the statement and no table reference. This enables us to leverage the expressiveness of SQL conditions without having to parse the actual result.

```
isTrue(condition: String): FutureStatementCondition
```

Fig. 4: LSD driver support for conditions based on a future state of the database.

The `FutureStatementCondition` interface then has two operations, `whenTrue` and `whenFalse`, where both get as parameter a future to be executed. One limitation here is that it is not possible to execute any statement inside `whenTrue` and `whenFalse` operations.

## 4.2 Other features

Similarly to the JDBC API, the LSD JDBC API also has support for batching multiple inserts and updates into a single SQL statement to the database. This is done by introducing two distinct operations in the `PreparedFutureStatement` interface. The first operation is similar to the one existing in the JDBC, which is `addFutureBatch`. This operation indicates to the statement that it should hold the values of the current batch and that subsequent parameters are set to not replace already present ones. The second operation is `executeFutureBatch`, and is used to indicate to the `PreparedFutureStatement` that its internal future action is to execute the operation `executeBatch` of the JDBC API.

## 5 Evaluation

In order to evaluate the work presented, we have executed two different test scenarios with the *New-Order* transaction present in the TPC-C-C benchmark. We chose this transaction because of its ability to capture different contention levels. The first test scenario, represented the high contention case, was executed using 10 terminals and 1 warehouse, while the second scenario, represented by the low contention case, used 10 terminals and 10 warehouses. Both scenarios were executed for 2 minutes, after which we analyzed the throughput and the percentile 99 of the latency. We validated this work with 5 distributed nodes, each with a *2xAMD Opteron 2376 CPU@2.30GHz*, *16GB* of RAM and *2x1Gbps*

network connections. One node hosted a *PostgreSQL* database while the others acted as clients.

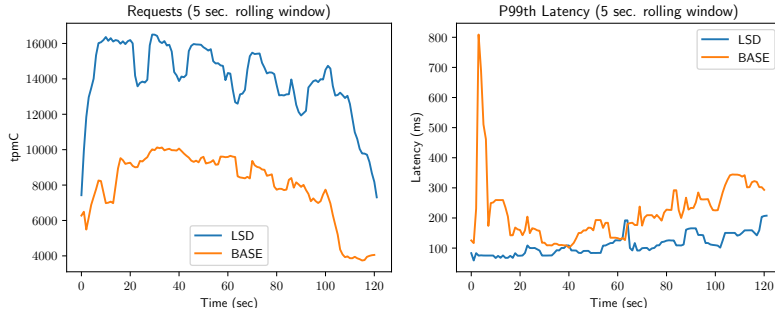


Fig. 5: *New-Order* transaction benchmark between JDBC LSD (LSD) and Standard JDBC (BASE) for a scenario where contention is high, i.e., 1 warehouse and 10 terminals. The left plot compares throughput over a 5 seconds rolling window, while the right plot shows latency.

The high contention scenario is plotted in Figure 5. The plot shows us that the throughput of the *BASE* (Standard JDBC) is not good when compared to the throughput of *LSD*. This happens because, as contention increases, so do the chances of conflicts and risk abort/rollbacks of transactions. The same happens to the latency of *BASE*, which is a lot higher than the *LSD*. As for *LSD*, the throughput is a lot higher, with some cases where it goes down, with this being likely caused by cases where multiple transactions have their window intersected, which is still possible to happen.

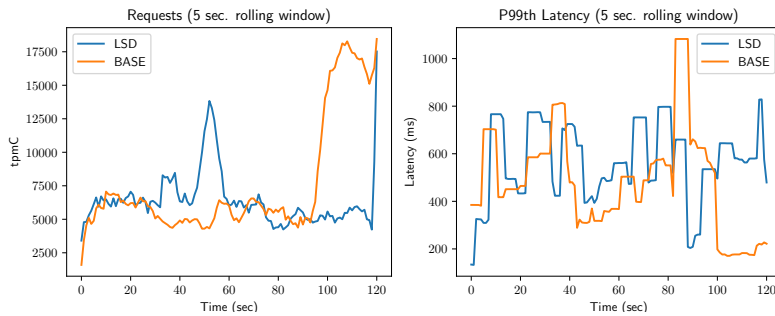


Fig. 6: *New-Order* transaction benchmark between JDBC LSD (LSD) and Standard JDBC (BASE) for a scenario where contention is low, i.e., 10 warehouses and 10 terminals. The left plot compares throughput over a 5 seconds rolling window, while the right plot shows latency.

The low contention scenario is plotted in Figure 6. The plot shows a very similar throughput between *LSD* and *BASE*, with both having spikes in throughput around the same time. A similar behavior is seen in the latency, with it not being stable for both *LSD* and *BASE*. At the time of writing of this article it was not clear the reasons, hence, more study on this matter is required.

## 6 Related Work

Improving performance of database transactions has been an area of study for a long time, with many approaches [2, 4, 8, 9, 1, 7, 6]. STRIFE [2] being one of them, proposes to improve transactional performance by introducing transaction analysis and aggregation to avoid data conflicts. This is achieved by aggregating committing transactions into batches. The batches are then partitioned into conflict-free and residual groups. The conflict-free group is able to execute without any concurrency control, while the residual group needs to execute using concurrency control. Although its ability to improve performance of some transactional workloads, in the end a new step for analysis and aggregation is added, which can hamper performance of the database system.

Another approach is the one followed in the Salt database [8]. This database stems from the idea that only a few transactions influence the total system performance, and hence, only those need to be optimized. To optimize such transactions, they propose the rewriting and partitioning of such transactions into sub-transactions with a relaxed isolation between themselves. In the end, this relaxed isolation adds different types of locks that stem from the idea of *BASE* [3] transactions. A limitation of this approach is that it requires a complex analysis and rewriting of transactions by its developers in order to partition it into sub-transactions, and in doing so, bugs may be introduced.

Callas [7] takes Salt database [8] a step further by automating the process of partitioning transactions through static analysis and through iterative processes for finding good transaction decomposition, assigning groups and a specific CC for each group. This, however, still adds a step of analysis which in the end can hamper performance of the database system.

In the end, the work presented here takes the task of improving a database system performance from a different perspective, more specifically from the client perspective, which differs from the approaches followed by most literature. In doing so, it leaves open the possibility of even combining LSD with some work in the literature.

## 7 Conclusion

This paper presents an implementation of the work of Vale [5]. Although different from the original work that implemented the lazy state directly in the database, this work focused on bringing the lazy state to the client using JDBC with the idea that it would make the use of futures more explicit to the client and at the same time without requiring changes to the database implementations of

different vendors, which in the end enables the lazy state to be used with other solutions as mentioned in Section 6.

The driver was tested using the TPC-C benchmark and, achieved a greater performance in the database in high contention scenarios and similar performance in the low contention scenario. Although, the low contention scenario has some shortcomings in the end we believe the work still relevant as it presents an alternative approach with little overhead for a programmer implementing a client. To deal with that we would do an evaluation of actual size of transaction windows and the conflict rate of transactions.

## References

- [1] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. “Cicada: Dependably Fast Multi-Core In-Memory Transactions”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD ’17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 21–35. ISBN: 9781450341974. DOI: 10.1145/3035918.3064015.
- [2] Guna Prasaad, Alvin Cheung, and Dan Suciu. “Improving High Contention OLTP Performance via Transaction Scheduling”. In: *CoRR* abs/1810.01997 (2018). arXiv: 1810.01997.
- [3] Dan Pritchett. “Base an acid alternative”. In: *Queue* 6 (3 May 2008), pp. 48–55. ISSN: 15427730. DOI: 10.1145/1394127.1394128.
- [4] Eduardo Subtil. “Lazy State Determination for SQL Databases”. MA thesis. NOVA School of Science and Technology, 2021.
- [5] Tiago Marques do Vale. “Executing requests concurrently in state machine replication”. Nova School of Science and Technology, 2019. URL: <https://run.unl.pt/handle/10362/71218>.
- [6] Yingjun Wu et al. “An Empirical Evaluation of In-Memory Multi-Version Concurrency Control”. In: *Proc. VLDB Endow.* 10.7 (Mar. 2017), pp. 781–792. ISSN: 2150-8097. DOI: 10.14778/3067421.3067427.
- [7] Chao Xie et al. “High-performance ACID via modular concurrency control”. In: *SOSP 2015 - Proceedings of the 25th ACM Symposium on Operating Systems Principles* (Oct. 2015), pp. 279–294. DOI: 10.1145/2815400.2815430.
- [8] Chao Xie et al. “Salt: Combining ACID and BASE in a Distributed Database”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI’14. Broomfield, CO: USENIX Association, 2014, pp. 495–509. ISBN: 9781931971164.
- [9] Xiangyao Yu et al. “TicToc: Time Traveling Optimistic Concurrency Control”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1629–1642. ISBN: 9781450335317. DOI: 10.1145/2882903.2882935.