Università
Ca'Foscari
Venezia

Master's Degree
in **Computer Science - Software Dependability and Cyber Security**
Second Cycle (D.M. 270/2004)

—
Ca' Foscari
Dorsoduro 3246
30123 Venezia

Computer Science Master's Thesis

Adaptive query handler for ORM technologies

Supervisor
Andrea Marin

Graduand
Paolo Quartarone,
859724

Year
2019/2020

Abstract

In modern software engineering, the mapping between the software layer and the persistent data layer is handled by the Object Relational Mapping (ORM) tools. These transform the operations on objects into DBMS CRUD queries. The problem of formulating the query associated with the operations in the most efficient way has been only partially solved by static code annotations. This implies that the programmer must guess the behavior of the software once it is deployed in order to choose the best configuration. In this work, we make a step toward the dynamic configuration of the queries. The ORM we propose aims to improve the overall system performance monitoring and adapting the behavior of the query. The solution achieves the result by pruning the query in two steps. In the first step, the ORM chooses the columns to fetch, taking into account the system load and usage frequencies. In the second step it exploits the join elimination optimization. This is a feature implemented by some DBMS that removes unnecessary tables from a query, avoiding useless scans and join operations. Then, the ORM proposed applies together eager and lazy strategies. It loads the expected data eagerly, and it loads lazily the data not expected but subsequently requested. The efficiency of the proposed solution is assessed through customized tests and through the Tsung benchmark tool, comparing the ORM developed with a simple JDBC connection and the Hibernate ORM service.

Keywords: ORM, Java, performance, queueing systems, database, software engineering

Contents

1 Introduction **5**
- 1.1 Problem description . . . 5
- 1.2 State of art solutions . . . 6
  - 1.2.1 Evaluation strategy . . . 6
  - 1.2.2 ORM tools . . . 8
  - 1.2.3 Auto-Fetch . . . 11
- 1.3 Proposed solution . . . 12
- 1.4 Document content . . . 13

2 Theoretical Background **15**
- 2.1 Database . . . 15
  - 2.1.1 Relational database . . . 16
  - 2.1.2 NoSQL . . . 16
  - 2.1.3 SQL vs NoSQL . . . 17
- 2.2 Object Oriented Programming . . . 17
  - 2.2.1 Patterns . . . 18
- 2.3 Multitier Architecture . . . 19
- 2.4 Queueing system . . . 20
  - 2.4.1 Little's Law . . . 22
  - 2.4.2 PASTA Property . . . 23
- 2.5 Queueing networks . . . 23
  - 2.5.1 Open Network . . . 24
  - 2.5.2 Closed Network . . . 24
- 2.6 Performance testing . . . 26
- 2.7 Statistical inference . . . 27
  - 2.7.1 Hypothesis Testing . . . 27

3 Implementation of AdaORM 31
3.1 Software . . . 31
3.1.1 Programming language . . . 31
3.1.2 Database . . . 31
3.1.3 Frameworks . . . 32
3.1.4 Libraries . . . 32
3.1.5 Patterns . . . 32
3.1.6 Main functionalities . . . 34
3.1.7 Conclusion . . . 40

4 Experiments and Analysis of AdaORM 41
4.1 Network architecture . . . 41
4.1.1 Application Server . . . 42
4.1.2 Database Server . . . 43
4.2 Databases Benchmark . . . 44
4.2.1 Join Elimination optimization . . . 45
4.2.2 MySQL . . . 45
4.2.3 PostgreSQL . . . 49
4.2.4 DB2 . . . 50
4.2.5 Conclusion . . . 52
4.3 Application Benchmarks . . . 53
4.3.1 Configuration . . . 54
4.3.2 Database . . . 55
4.3.3 Custom benchmark . . . 56
4.3.4 Web Server benchmark with Tsung . . . 61
4.3.5 Comparison of `AdaORM` and Hibernate . . . 72
4.4 Queueing system analysis . . . 76
4.4.1 Adaptive Heavy Query . . . 76
4.4.2 JDBC Heavy Query . . . 80
4.4.3 Conclusion . . . 84
4.5 Prototype limits . . . 84
4.5.1 Concurrency . . . 85
4.5.2 System database . . . 85
4.5.3 Usability . . . 85

5 Conclusions and Future Works 87
5.1 Data prediction . . . 88
5.2 Cost impact . . . 88

6 Acknowledgements **91**

Chapter 1

Introduction

1.1 Problem description

Nowadays, exist some interesting methodologies to allow communication between software layer and data layer. Avoiding verbose code to fetch data from the last layer, it is possible to use the Object Relational Mapping (ORM) tools. These tools implement ORM programming paradigm to favor the integration among object oriented programming languages (OOP) and relation database management system (RDBMS). ORM tools try to solve the problem of formulating the query associated with the operations in the most efficient way by *static* code annotations. This implies that the programmer must guess the behavior of the software once it is written in order to choose the best configuration. A wrong *static* choice will lead to an unnecessary waste in terms of computation time and resources used, these degrade performance. For example, the developer chooses statically a query that loads a big result set but the user always uses only some values. In this case, where the user only needs a small amount of data, our application, set statically, loads a lot of them anyway. With this strategy our developer has a heavy application, which response time is higher than the optimal time. Then, the developer chooses a cheaper static approach, at least at the beginning. He chooses to load only values that user asks. This can improve response time at the beginning, but if the user runs a routine that requires all information he will load each parameter individually, giving a big waste of time. Then he decides to study the entire system to decide how and where the application must load a bigger result set and where a smaller one. But the system is

too complex and in continuous evolution. In this case, the developer must analyze and maintain too many cases, increasing developing time and costs.

In this thesis we present `AdaORM` , an ORM prototype that aims to improve data fetch from database using a new strategy that handles the query after the observation of its behavior. `AdaORM` also allows to reduce the development and maintenance times of the application, thus makes the development of the application cheaper. `AdaORM` automatically chooses the most probable query to submit to RDBMS according to observed behaviors of the query and parameters utilization. To achieve the goal, `AdaORM` performs optimization in two steps. In the first step, it collects statistics over query columns usage and chooses dynamically what of them can be removed from the query statement. In the second step, after the submission of the new edited statement to the DBMS, `AdaORM` exploits the `join elimination` optimization, a feature implemented by some DBMS that removes not useful tables from the query, avoiding unnecessary scans and join operations. `AdaORM` is able to improve query execution times because DBMS works with less tables, it improves system response time because `AdaORM` handles less data and, at the end, it reduces the system load.

1.2 State of art solutions

In this section we describe firstly *state-of-art* strategies that ORM tools, persistence framework and active record database pattern implement. For simplicity, we always talk about these three solution as ORM tool. In fact we are interested to understand how they works and how they try to optimize data fetch. The strategies are described by treating their strengths and weaknesses, demonstrating on which cases they are the best choice and in which the worst one. Then, we talk about the most famous ORM solutions. ORMs are described according to their purpose, their strategies and their functions used to achieve it. So let's describe their strengths and weaknesses. At the end, we talk about an interesting technique called `AutoFetch` that generates automatically prefetching using transversal profiling.

1.2.1 Evaluation strategy

Evaluation strategy changes the behavior of execution flow according to the evaluation chosen. An evaluation strategy decides when and how evaluate

an expression that is bound to a variable. Applying an evaluation strategy to an ORM changes its behavior according to the static setting defined by the developer. To understand better how a ORM tool works is important to understand how the applied evaluation strategy works.

Strict evaluation: called also *eager evaluation*, is an evaluation strategy which evaluates as soon as possible the expression that is bound to a variable. Using this strategy it improves code workflow and facility the debug. However, eager evaluation can *reduce performance* in case of too many and unnecessary evaluations.

An ORM tool that uses this kind of strategy loads immediately all values from the goal table and also loads immediately all values from relation `one-to-many`. Only `many-to-many` relations will be performed after requesting. An application that implements an ORM with this strategy provides slower response times following an object request, but it has no delay when asked to return the value of an object property with one-to-many relationship. Avoiding making new connections to the database, the application is much faster providing the required value. However, if the loaded values from relations are never used, we can interpret this situation as a waste of time and system resource.

Non-strict evaluation: also called *lazy evaluation*, is an evaluation strategy which evaluates expressions bound to a variable only in the moment they are required to complete the execution flow, correctly. This strategy allows to improve performance in the opposite situation of eager evaluation. In fact, in case we need the results of all evaluations and the strategy that we are using is *lazy*, we have worse performance.

An ORM tool that uses this kind of strategy will load immediately all values from the goal table and, when requested, the values from other table with relations `one-to-many` or `many-to-many`. ORMs tool that works with this strategy responds faster when a client requests the object fetch, but it responds slower when it asks to return values from `one-to-many` relations. The performance in this case degrades because ORM tool must pay some fixed time in database connection, even if there is little data to load. However, if the loaded values from relations will be never used, we can interpret this situation as a gain of time and system resources.

Non-deterministic evaluation In this evaluation strategy arguments are loaded after heuristics evaluations at run-time. We cannot know how the workflow will be. These family of evaluation strategies can give high performance improvement, but its result might provide unexpected values. Debugging can be complicated and the execution flow is decided when choosing whether to evaluate an expression or not.

`AdaORM` implements a predictive evaluation strategy that can be mapped in this macro strategy.

1.2.2 ORM tools

Object-Relational Mapping (ORM) is a programming technique that aims to improve integration among software system, that uses object oriented paradigm and RDBMS systems, creating a virtual object database. A ORM tool solves the problem of translating the information to be stored in a relational database, preserving the properties and relationships that involve object in OOP paradigm. Those tools load data from databases according to the chosen *evaluation strategy*.

Advantages: by introducing this kind of technology, we obtain some advantages. As the *reduction of the code to write*. The less we write, the less error we make. Also, development time is reduced and we are able to *avoid boilerplate code*. Using an ORM *improves the portability* over the DBMS used. Only changing some lines of code and importing the correct driver we can use one specific language instead of SQL.

Disadvantages: there are some unfavorable points. The higher level of abstraction *doesn't show what happens inside*. Sometimes, it can be useful making the process transparent. Other times, *it doesn't give enough information and/or control* to improve the behavior of an application. The last disadvantage is the main point that we want to improve with `AdaORM` .

Also, an ORM tool helps the developer with other features like automatic object graph loading, concurrency management, caching support and improvement on DBMS communication.

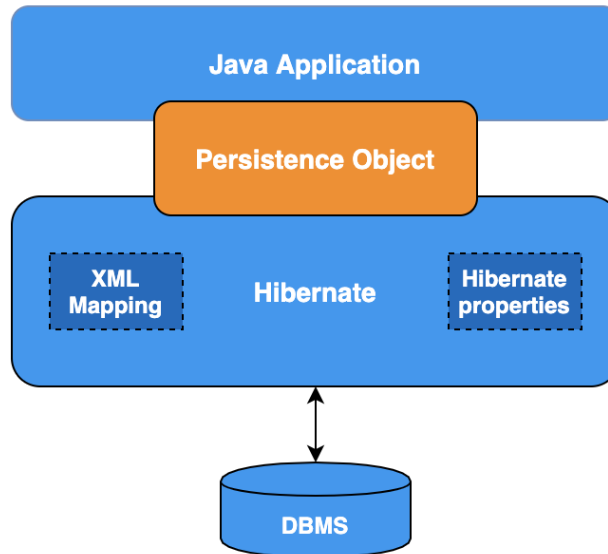

Figure 1.1: Hibernate architecture

Hibernate

Hibernate (H8) is an object-relational mapping framework that gives support to manage data persistence into databases, representing and maintaining Java object in relational databases. H8 gives mapping between Java classes and relational database tables. Also, it executes the queries that the developer needs to update the objects. In this way, the developer does not need to write any line of code to assure data persistence. Hibernate is an ORM *object-centric*. It gives more importance to the object that uses data from the database than the database itself.

In Figure 1.1 we can observe at high level Hibernate architecture. We can see that model objects (called persistence objects) are the communication bridge between Hibernate and our application. Also, our application *never* communicates directly with DBMS, but only through Hibernate commands (if we implement a homogeneous solution).

Hibernate became so famous, also for the previous ORM features, thanks to its personal features as dual-layer cache architecture, custom query language, session managed and general CRUD functionality.

H8 uses `xml` files, called mapping files (format name: `class_name.hbm.xml`)

to describe the structure of persistent Java object and a configuration file that contains information about database connections and mapping files. Therefore, in order to use the previous data, we must write a class which takes care of saving the persistent object in the db. The SQL code to make the schema is created by mapping files.

Hibernate is a complete, stable and affordable solution suitable for complicated systems with persistent domain. However, these features have the downside in decreasing performance when running large SELECT operations than the next solution, *MyBatis*.

MyBatis

MyBatis isn't an ORM framework, it is instead a persistence framework. Developed by Apache, it automates the mapping between objects in Java (and `.NET`). It is the evolution of iBatis, that was dismissed since 2010. A persistence framework maps SQL code into Java methods, while an ORM framework maps Java classes to database tables.

MyBatis implements some interesting features as *lazy loading* and *aggressive lazy loading*. The classical lazy loading loads all values from a table. Aggressive lazy loading loads *only* required values. Besides, this framework provides a caching system to improve performance. MyBatis is preferable then Hibernate when our approach is *database-centric* and we want to use analytic queries.

MyBatis works in reverse way respect to Hibernate. In fact, the developer starts writing SQL code into XML files to create a database schema, and just at the end, MyBatis creates the Java objects and methods. Also, MyBatis centers around XML files that contains SQL parameters to database connection. The mappings are decoupled from the application logic to XML configuration files, packaging the SQL statements into them. In this case the developer does not have to develop low level SQL queries.

MyBatis is the best choice when we work with a database where you need to write fairly complicated SQL queries. It gives high throughput but it isn't suitable for a large object-centric application.

jOOQ

jOOQ is a software library that implements active record database pattern. It provides a language very similar to SQL to perform queries. This one

allows to implement some functionalities that cannot be used with other ORM, staying very light. jOOQ has a database-centric approach, as we could imagine from its syntax. The syntax allows to standardize the language, in this way it will be independent from under RDBMS layer. Also, jOOQ is multi-tenancy, it works using many instance of the same service in a shared environment.

jOOQ abstracts SQL through some function. In this way we become independent from the under DB and we are less exposed to risk. It supports many SQL features that cannot be used with other ORM. In fact, ORM such as Hibernate, are expensive resources and they don't permit all SQL operations. jOOQ is very different than Hibernate and MyBatis. It gives to the developer a lot of control, which in inexperienced hands can lead to serious performance problems. jOOQ implements eager loading by default. This means that if you are using a large database and you are loading a lot of data, which of then might not be all used, it will lead to an unnecessary waste of time and resources.

ORM quick matching

In Table 1.1 we can see a comparison of the main functionalities between the ORMs seen previously. The comparison is done in terms of strategies, caching, the philosophy to which it is oriented and the amount of resources it needs.

Tool	Strategies	Caching	Oriented to	Resources
Hibernate	Lazy/Eager	Yes	Object	Expensive
MyBatis	(Aggressive)Lazy/Eager	Yes	DB	Cheap
jOOP	Lazy/Eager	Yes	DB	Cheap

Table 1.1: Matching principal ORMs

1.2.3 Auto-Fetch

`Auto-Fetch` is a technique that aims to automate prefetch. In fact, to improve performance many architecture supports query prefetch associated to an object as query result. Unfortunately prefetch must be done statically

in the code and sometimes might be difficult to guess the correct query of maintain it, mostly in modular system. `Auto-fetch` achieves this result through the traversal profiling[1] in object persistence architecture. This technique based its prefetch over previous query executions of similar queries. `Auto-fetch` records traversed associations when we submit a query. Then, recorded information are aggregated to profile a statistical application behavior. It can prefetches arbitrary transversal patterns. In this way, the application performs less queries, improving performance. `Auto-fetch` maps execution flow with a graph, because is the natural representation of relations in most object persistence architectures. `Auto-fetch` understands only after one execution how to prefetch correctly a query. In fact, from the second execution it is already able to execute the best prefetch. However, it classifies only on the criteria, without distinguish the different query utilization. In this case the classification is too coarse. Furthermore, `Auto-fetch` does not implement the feature to perform lighter prefetch if the load on the system is higher. It also can possibly aggravating the system load.

1.3 Proposed solution

The contribution of this thesis is the development of an ORM prototype, `AdaORM` , that aims to improve system response time and system load. `AdaORM` uses a predictive strategy that exploits collected statistics and a feature implemented in some DBMS, `join elimination` optimization. `join elimination` prunes query statement removing useless tables. In this document we assess through customized benchmark test and using Tsung benchmark tool, how the system response time, the complexity of executed query and system load decrease exploiting the features offered by `AdaORM` .

Achieving the fixed goals is possible to exploit a particular feature implemented from a some DBMS, the `join elimination` optimization. This optimization removes unnecessary tables from a submitted query. In this way, the DBMS performs less operation, decreasing the query cost and improving the response time. Before `AdaORM` , the developer was forced to hard code different query for different execution contexts to improve performance, handling the query each time that the system behavior changes. Then, maintaining system efficient and performing each time that the behavior changes, has an expensive cost in term of maintenance and complexity. Now, thanks

[1]transversal profiling is a technique to collect statistics tracking the control flow.

to AdaORM we are able to solve these problems. It works observing the behavior of requested query and the mapped column utilization. By the using of collected statistics on the application behavior, AdaORM formulates the query statement by removing the columns which are not evaluated as interesting. Ranges are set up monitoring the current system load. When we reload the interrogation we have to check the fixed system load. If a column frequency usage is enough high and according to the full system usage, it is loaded into the final query.

The tables are not removed by AdaORM but only from DBMS that implements join elimination .

1.4 Document content

In the first chapter we described the problems that we aim to face, how to improve application response time and save resources. We described the state-of-art solutions and how they try to solve the problem or some shades of it. Then, we briefly described AdaORM , the proposed solution to previous problems.

In the second part we give some theoretical knowledge about databases, that are the heart of the problem. Then we describe the Object Oriented Programming, a programming paradigm to develop structured and modular applications. After that, we talk about Object Relational Mapping, a programming technique to improve integrity and develop time among relational database and Object Oriented Programming, with some principal solutions. Then, we introduce the multitier architecture, a way to describe a particular type of client-server architecture, to pass after that to talk about queueing networks and how to perform performance testing. At the end, we spend some words to describe the statistical methods that we want to use to assess the performance improvement given by AdaORM .

In chapter 3 we describe how we implemented AdaORM , describing software and hardware components. This chapter is necessary to understand better the subsequent one.

In chapter 4 we show the experiments done with different databases to assess the performance improvement gained having the join elimination . Then, we comment some plots obtained from benchmark results of AdaORM . Benchmarks have been done with a custom benchmark and with Tsung in server configuration. We compare AdaORM with Hibernate and we show some

cases of study where the proposed solution overcome Hibernate execution time. At the end, we give some limits found in my prototype.

In the last chapter we give a last observation over the project and we propose interesting possible future works to be implemented on `AdaORM`, focusing on improving data prediction and reducing the computation cost of the application layer.

Chapter 2

Theoretical Background

To make this self-contained thesis we decided to theoretically introduce the main topics that will be covered. In the next sections we will talk about *databases*, that are a key point of our research. We will introduce the *Object Oriented Programming* (OOP) paradigm and we will describe principal pattern used. After that, we will talk about *Object Relational Mapping* (ORM), a technique that allows us to link together two different paradigm as RDBMS and OOP. Then, we will describe *Multi-tier architecture* to represent a particular type of client-server architecture. Studying and testing the adopted system will be possible thanks to *Queueing system network*, described below. At the end, we will give some theoretical definition over statistics methods that we will use to assess the obtained improvements.

2.1 Database

In this section, we will talk about the two families of databases and their principal features.

A database is a collection of data, organized and electronically stored over a computer system.[1] Interacting with a database is possible thanks to the *database management system* (DBMS). A DBMS is a software layer that allows users and applications to interact with data layer. According to *database model*, that determines the database logical structure and defines how data can be stored, organized and manipulated, we split DBMS in two families: *Relational database* and *NoSQL*.

2.1.1 Relational database

Relational database is a database structured over the concept of relation. It is possible to interact with a relational database thanks to *Relational Database Management System* (RDBMS). Data are presented to the end user and application as relations of *tables*. Each table consists of a set of rows, called also records or tuple; and columns, that are the attribute which value describes rows. Each row is uniquely identifiable into the system through the use of a *primary key* (PK), a particular attribute or set of attributes that are unique in the table. PK can be used also as *foreign key* (FK) to link together rows and to make relation among them.

SQL: acronym of *Structured Query Language*, is the query language used to interact with a RDBMS. A RDBMS can extend SQL with many other features as new commands or attribute type. This extension takes the name of *dialect*.

2.1.2 NoSQL

A NoSQL database is a system that provides methods to store and to retrieve information in different ways which are not the *classic* relational models. NoSQL database can be split in different families that depend from the type of data model they work with. The most used data models are document graph, key-value and wide-column. Using NoSQL databases can offer some advantages, as design simplicity, flexibility working with unstrucutured data, simpler "horizontal" scaling to cluster, but NoSQL database has an high impact over the amount of memory to storage data. However, the cost to store data is a convenient cost because its simplicity decreases the development costs. Now we explain better the difference between data model used by NoSQL databases.

Document databases: data are stored into document as JSON. Into the document we can find a couple of *key-value*. Values have type, that can be primitive, or complex type as object. Sometimes, object/variable type into document are the same used by programming language used. This simplifies the mapping between data into document and classes.

Graph databases: data are stored as a graph, where node represent values and edges represent the relations among them. Graph representation is a convenient choice when we have to work with algorithms that nativity require handling graph.

Key-value databases: data are memorized in couples *key-value*. The key is used to retrieve information linked with it. Key-value is a convenient representation when we need to retrieve quickly a value, but we do not perform complex query over our database.

Wide-column stores: data are saved in tables, rows, and dynamic columns. Wide-column is like a relational database, but it provides a lot of flexibility because each row is not required to have the same columns. Wide-column is a convenient representation when you need to memorize large amounts of data and you can predict what your query patterns are.

2.1.3 SQL vs NoSQL

In Table 2.1 we recap the main features of each paradigm. There is no best paradigm. We must choose the best according to the problem we have to sort out.

Type	Cluster	Join	Representation	Memory	OOP use
SQL	Difficult	Provided	Relations	Low	Hard
NoSQL	Easy	Absent	Many type	High	Easy

Table 2.1: Comparison between SQL vs NoSQL

2.2 Object Oriented Programming

In this section we will see one of the most famous programming paradigm used, *Object Oriented Programming* (OOP). After a quick explanation of it, we will describe the pattern applied into our project.

Object Oriented Programming is a programming paradigm whose main core concept is the *Object*. An object is a wrapper that contains data structured in fields or properties, and some action, accessible through methods.

2.2.1 Patterns

We can define a pattern as a reusable solution to frequently current problems. Exploiting pattern is the best way to develop good codes. In this document we will see only pattern that we used developing `AdaORM` [7].

Strategy: is a behavioral pattern that allows to choose the right algorithm at runtime. Delegating the algorithm choice at runtime, it improves the code reusability. Validation algorithm and validating object are encapsulated separately. In this way we can validate the same object in different contexts without duplication codes.

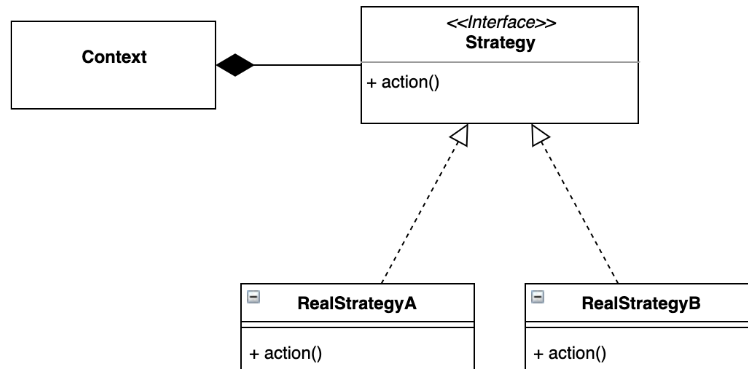

Figure 2.1: UML representation of Strategy pattern

In Figure 2.1 we can observe how the `Context` class does not implement directly any action, but delegates its implementation to others classes. In fact, implementing `Strategy` interface, the `Context` will be able to change its behavior dynamically, changing the referred strategy. Classes `RealStrategyA` and `RealStrategyB` implement the Strategy class, then the algorithm that will be executed.

Decorator: aims to solve the problem of how adding/removing object responsible dynamically or at runtime, avoiding subclassing explosion.

Decorator does not change the behavior of original class, but `wraps` it. How we can see in Figure 2.2, Decorator class has an attribute that is of the same type as the class that extends. In this way Decorator, also appears as

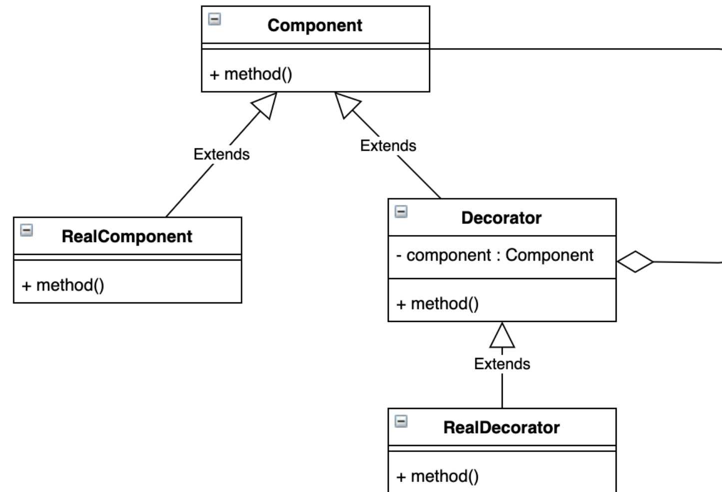

Figure 2.2: UML representation of Decorator pattern

an object of the same class of component, it contains an instance to use to methods delegation in order not to change the behavior of real class.

Data Access Object: also called DAO, is a pattern used to split low level data accessing API and high level services. DAO is designed according to `Data Access Object Interface` that designs the operations to perform over an object model, `Data Access Object concrete class` that implements the previous interface, writing the methods to effectively interact with databases, and `Model Object or Value Object` that will be the simplest POJO where storing fetched data.

In Figure 2.3 we can see that implemented interface allows to fetch target object independently from its sublayer. In fact, we can implement a different DAO for different purpose.

2.3 Multitier Architecture

In modern software engineering, a client-server architecture for distributed system is called multitier architecture when application processing and data processing are physically split.[6] All resulting layers are in communication.

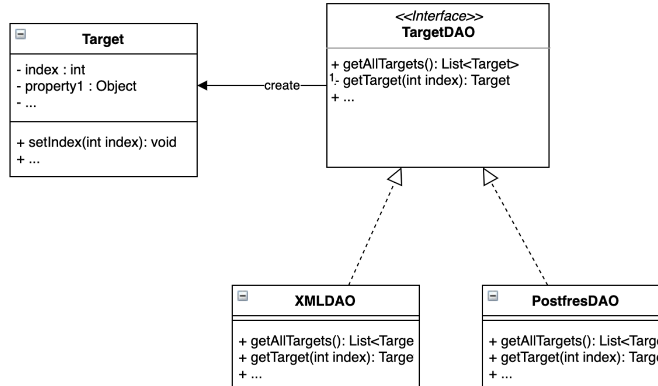

Figure 2.3: UML representation of DAO pattern

The most famous multitier architecture is the three-tier architecture, where presentation, control and data are decoupled. Developing in a n-tier architecture is a big advantage for a developer because it can exploit more flexibility and scalability.

2.4 Queueing system

Queueing systems have a very important role in the assessment of software and network performance. They allow modeling a theoretical system over its possible calculate metrics to study the real system. A queueing system is composed by three parts: an *arrival process* that describe how the jobs come into the system, a *waiting room* that describes the area where jobs wait for to be served, and a service room where jobs are served. The time spent into service room is called *service time*.

A system adopts a scheduling discipline, a rule that aims to define how to serve a job that is in waiting room. The discipline can be split in two categories: *preemptive*, a job can be removed from the service room and reput in the waiting room because of some events, or *not-preemptive*, when a job cannot be removed from the service room before it has finished.

The most famous are:

- FCFS: First Came First Served is a not-preemptive policy. A system

set with this rule will serve before the job, according to their arrival order.

- LCFS: Last Came First Server is a preemptive policy. A system set with this discipline will serve job on inverse arrival order.
- RR: Round Robin is a preemptive discipline. The system spends a certain amount of time for each job.
- SJF: Short Job First discipline is a preemptive rule that aims to maximize throughput serving first shorter jobs.
- SRTP: Shortest Remaining Processing Time is a preemptive discipline with resume that serves always the job that has the shortest completation time. It tries to minimize the response time.
- RAND: policy not-preemptive that chooses randomly a job whenever the server is free.

Describing a queueing system can be very verbose. It is possible to avoid this issue using Kendall's notation. A system can be described using a string as $A/B/m/K/P/D$ where:

- A and B describe the inter-arrival times and the distribution of the service times of jobs. A and B can be replaced with the following letters to describe the distribution type:
  - M denotes the exponential distribution. If M replaces A the represents the Poisson distribution.
  - D describes the deterministic distribution.
  - G denotes the general distribution. It is in most of the general case.
  - many others
- m is the number of identical servers in the system.
- K represents the capacity of the queue.
- P describes the population size.
- D is the scheduling discipline.

For example, M/M/1 is the Kendall's notation to describe a system with Poisson distribution at arrival process, exponential distribution at departure process and one server, with infinite capacity and infinity population, and the scheduling discipline is FCFS.

To study a queueing system it is necessary to introduce some *performance indices* that describe some particular behavior of the case of study.

- $N(t)$ is the number of jobs in the system in a certain epoch t.
- W is the random variable that describes the waiting time of a job into the waiting room.
- S is the service time of a job.
- μ is the service rate. $E[s] = \mu^{-1}$.
- R is the amount of time that a job spends into the system. $R = W + S$.
- λ is the arrival rate of a job into the system.
- U is the utilization of a single system queue defined as $U = \frac{\lambda}{\mu}$

2.4.1 Little's Law

Little's Law is one of the most important result in queueing system theory. It requires very few constraints to be applied. In fact, it is independent from arrival/departure distribution used to describe the system, but it allows however to calculate the number of jobs in a system in a defined time t.

Little's law: A queueing system without internal loss or generation of jobs is given, then the following relation holds for any finite time t:

$$\overline{N}(t) = \overline{R}(t)X(t)$$

From the previous law we can deduce the next theorem setting $t \to \infty$.

Little's Theorem: A queueing system without internal loss or generation of jobs is given, and the following limits exist:

- $\lambda = \lim_{t\to\infty} \frac{A(t)}{t}$
- $\overline{N} = \lim_{t\to\infty} N(t)$
- $\overline{R} = \lim_{t\to\infty} \sum_{i=1}^{C(t)} \frac{r_i}{C(t)}$

with $C(t)$ is the number of jobs served in $[0, t]$ and r_i is the response time of the $i - th$ service. Then

$$\overline{N} = \overline{R}(t)\lambda$$

If the previous limits are satisfied we will have a stable system for $t \to \infty$ and the system throughput will be balanced with the arrival rate into the system. A system is *stable* when $t \to \infty$ the expected number of jobs into the system is finite.

2.4.2 PASTA Property

Poisson Arrivals See Time Averages property, also known as *PASTA* is a property used to describe a queueing system.

Definition: A queueing system with a Poisson arrival process is given. A job immediately before its arrival into the system will see the same distribution as the random observer's one.

PASTA property is used to derive important results as the *residual life of a job*, the amount of remaining service time in service from the point of view of a random observer.

2.5 Queueing networks

In this chapter we talk about queueing network, a system where each single queue is connected to the other by a routing network. This system is necessary to describe and study complex systems.

Queueing networks are classified in two categories that depend from the customers' behavior, *open networks* and *closed networks*.

2.5.1 Open Network

An open network is characterized by one or more input customers stream and one or more output customers stream. We study steady-state behavior of queueing network. To achieve this goal is necessary to understand and test when a queueinq network is unstable.

Definition: An open queueing network is defined unstable when the number of jobs into the system go to infinity for $t \to \infty$ with higher probability. Let $\overline{N}$ the expected number of user into the system. If exists the limit

$$\overline{N} = lim_{n \to \infty} \frac{N(t)}{t}$$

the the system will be stable. If a open queueing network is stable, then all its stations will be stable.

In a stable open queueing network, the total input flow into a station will be equal to its throughput. The trough, then, is independent from the service rates of the stations.

2.5.2 Closed Network

A network can be defined closed if the number of user that interact with the system is fixed. Then, there are not arrival and departure to the system. The closed loop networks are classified in two categories: *interactive systems* and *batch systems*. In the interactive system, a customer can pass from a thinking state to a submitted state, and from a submitted state to a thinking state. The time spent in thinking state is called *thinking time* (Z). Here the customer consumes the obtained result. The time spent in submitted state is called *response time* (R). The fixed number of users in the system is also called *level of multiprogramming* of the system. The combination of thinking time and response time allow defining the next definition of *system time*, so the time spent from a customer for the entire processing into the system. To perform it, it is necessary use the expected values.

$$\overline{T} = \overline{R} + \overline{Z}$$

We define the response time for an interactive system, emphasizing over the number of customer into the system, with the following result

$$\overline{R}(N) = \frac{N}{\overline{X}(N)} - \overline{Z}$$

We can observe that increasing N the response time will increase. This because there will be more competition into the system. From that relation we remove the thinking time Z because it does not depend from N but occurs in parallel. The response time depends on both level of multiprogramming and service rates.

We assume that for each station in the network the job service time is independent from the number of visits for each of them. Then, we can introduce the *service demand*. *Service demand* is a index that measures the total amount of service that a customer needs to each station for each visit done to a reference station

$$\overline{D_i} = \overline{V_i}\frac{1}{\mu}$$

And, since $\frac{1}{\mu}$ is the expected service time for each visit, we can write the next relation that represents the *bottleneck law*

$$p_i = \frac{X_i}{\mu} = \frac{X_1 V_i}{\mu} = X1\overline{D_i}$$

That relates the service demand to the system throughput and the queues load factors.

The system speed is bounded by the *bottleneck*. A bottleneck is the slowest component in the system and the higher system bound are limited from it. Finding the slowest component is possible thanks to the relations

$$X \leq min(\frac{\overline{N}}{\overline{D} + \overline{Z}}, \frac{1}{\overline{D_b}})$$

and

$$\overline{R} \geq max(\overline{D}, N\overline{D_b} - \overline{Z})$$

then

$$\overline{D} = \sum_{j=2}^{K} \overline{D_j}$$

where D_b is the bottleneck of the system that is the max D_i. So,

$$(if\ p_b \to 1 \text{ when } N \to \infty) \to X_b \to \mu_b \text{ and } \forall_b\ Q_b,\ U_b \to 1$$

The last metric that we see in this paragraph is the optimal number of concurrent user that the system can accept without degrading its response time.

$$N_{opt} = \frac{\overline{D} + \overline{Z}}{\overline{D_b}}$$

2.6 Performance testing

In this section we talk about performance testing, an important aspect in the life cycle of real hardware and software architecture. It allows to analyze and study the system, describing it. Performance test are classified by goals and architecture:

1. *Performance regression testing*: aims to check if system performance has been degraded after some changes in the code.

2. *Performance optimization testing*: aims to find the best software configuration that improves the performance.

3. *Performance benchmarking testing*: aims to give a performance description to end users.

4. *Scalability testing*: aims to find the maximum number of simultaneous users into the system before performance degradation.

To measure automatically the performance of a system it is necessary that the employed software simulates the user behavior. Here we distinct the two main components: *System under test* (SUT) and the software that aims to study its performance, the *benchmark*. We classify the benchmarks in two categories:

- *Competitive benchmark* is a standardized test that aims to assess the software and hardware performance. In this way we can perform consistent tests to compare machines or applications.

- *Research benchmark* is a tool developed to measure the performance of a certain system. Using this kind of application aims to improve the software performance after and during development time.

We can also split the performance test in four categories:

- *Synthetic*: aims to produce fictitious workload to simulate real customer behavior.
- *Micro*: aims to test only one aspect of SUT, without considering the rest of application.
- *Kernel*: aims to test performance from the most important part of the entire application.
- *Application*: aims to test each single feature of the under analysis application.

2.7 Statistical inference

In this section we talk about the background necessary to understand how and why we have used statistical inference methods to assess the performance [3]. Statistical inference is a process that aims to deduce properties using data analysis of a probability distribution. Inferential statistical analysis infers properties of a population, using a large sample from the population, for example by *testing hypotheses*. In this document we decided to use statistical inference methods to estimate the expected value of `AdaORM` execution and simple JDBC execution, assessing the performance improvements in terms of execution time. First of all, we introduce now for the next paragraph two important contents, *estimator* and *hypothesis testing*.

Estimator: is an approximation $\hat{\Theta}$ of a distribution parameter Θ performed using a sample of the total population. An estimator tries to approximate the real value of a population parameter. It's value is called *estimate*.

2.7.1 Hypothesis Testing

Hypothesis testing is statistical inference method used to verify statements, claims, conjecture or in general, hypothesis. Hypothesis testing is wide diffused in computer science to verify the efficiency of a new algorithm or a hardware upgrade. First of all, we must define what we want to test. We call them *null hypothesis* H_0 and *alternative hypothesis* H_A. H_0 and H_A are mutually exclusive. The rejection of H_0 means that we must accept H_A. The

not rejection of H_0 means that we cannot accept H_A. To reject H_0 in favor of H_A is possible only with significant evidence provided by data.

We can describe three different cases for alternative hypothesis H_A: *two-side alternative*, *left-side alternative* and *right-side alternative* hypothesis.

two-side alternative	$H_0 : \Theta = \hat{\Theta}$	$H_A : \Theta \neq \hat{\Theta}$
left-side alternative	$H_0 : \Theta < \hat{\Theta}$	$H_A : \Theta \geq \hat{\Theta}$
right-side alternative	$H_0 : \Theta > \hat{\Theta}$	$H_A : \Theta \leq \hat{\Theta}$

Table 2.2: Recap of different type of alternative hypothesis

Performing an hypothesis test could make some mistakes. *Type I* error occurs when we reject the true null hypothesis, *Type II* error occurs when we accept a false null hypothesis. The type I error is the most dangerous and we want to avoid it absolutely. We assign a probability to commit this error, called *significance level* α of a test.

$$\alpha = P(reject\ H_0 \mid H_0\ is\ true)$$

Hypothesis test is based on *test statistic* T. Then, to test our estimator we must first of all normalize the value

$$T_{\hat{\Theta}} = \frac{\hat{\Theta} - \Theta_0}{SE(\hat{\Theta})}$$

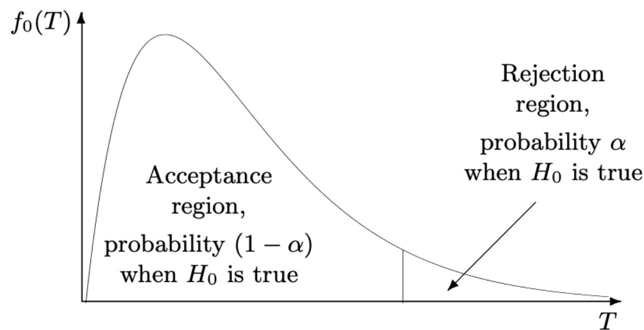

Figure 2.4: Acceptance and rejection regions

Using T statistic we are assessing that we are using *null distribution*. Then, we split the distribution in two areas *acceptance region* with probability $(1-\alpha)$ when H_0 is true and *rejection region* α when H_0 is true. Rejection regions are the tail (or the tails in two-side alternative) of the null distribution (in Figure 2.4[1] we can see a right tail of the null distribution).

Supposing that the estimator $\hat{\Theta}$ is unbiased and normally distributed, the null hypothesis can be tested using*Z-statistic*

$$Z = \frac{\hat{\Theta} - \Theta_0}{SE(\hat{\Theta})}$$

that has a *normal null distribution*. When $\hat{\Theta}$ is consistent and asymptotically normal, if the sample size is large enough Z has approximate standard normal distribution under the null.

We interpret Z values in the following manner

$$\begin{cases} \text{close to zero} \rightarrow \text{insufficient evidence against } H_0 \\ \text{far to zero} \rightarrow \text{evidence against } H_0 \end{cases}$$

According to three alternative seen before, now we explain how interpreting $z - statistics$

Right-tail alternative

$$H_0 : \Theta = \theta_0 \text{ vs } H_A : \Theta > \Theta_0$$

The rejection region R consists of 'large values' of Z:

$$R = [z_\alpha, +\infty) \text{ and } A = (-\infty, z_\alpha)$$

Significance level:

$$\begin{aligned} \text{Pr(Type we error)} &= Pr(Z \in R | H_0) \\ &= Pr(Z > z_\alpha) \\ &= \alpha \end{aligned} \tag{2.1}$$

[1] Figure from [3]

Left-tail alternative

$$H_0 : \Theta = \theta_0 \text{ vs } H_A : \Theta < \Theta_0$$

The rejection region R consists of 'small values' of Z:

$$R = (-\infty, -z_\alpha) \text{ and } A = [-z_\alpha, +\infty)$$

Significance level:

$$\begin{aligned} \Pr(\text{Type we error}) &= Pr(Z \in R|H_0) \\ &= Pr(Z < -z_\alpha) \\ &= \alpha \end{aligned} \tag{2.2}$$

Two-side alternative

$$H_0 : \Theta = \theta_0 \text{ vs } H_A : \Theta \neq \Theta_0$$

The rejection region R consists of 'large' and 'small' values of Z:

$$R = (-\infty, -z_{\alpha/2}) \cup [z_{\alpha/2}, +\infty) \text{ and } A = (-z_\infty/2, z_{\alpha/2})$$

Significance level:

$$\begin{aligned} \Pr(\text{Type we error}) &= Pr(Z \in R|H_0) \\ &= Pr(Z < -z_{\alpha/2}Pr(Z > z_{\alpha/2}) \\ &= \frac{\alpha}{2} + \frac{\alpha}{2} \\ &= \alpha \end{aligned} \tag{2.3}$$

In this chapter we have covered the main topics to provide a sufficient background for the complete understanding of the following chapters. We explained what a database is and what a DBMS is. We have seen the differences between relational and non-relational DBMS. So, we have seen some of the main patterns that we have implemented describing the problem they aim and how they solve it. Then, we have given an introduction to the multi-tier architecture of computer systems. We provided the knowledge necessary to understand the theory of queues and we motivated the need to apply performance tests. Finally, we have described the statistical method used to affirm the improvement in response times. Now, we are ready to move on to the next chapters.

Chapter 3

Implementation of AdaORM

In this chapter, we explain what technologies we used to develop `AdaORM` and why we chose them. We talk about the programming languages, frameworks, libraries, patterns and their integration in the project. Then, we explain the most interesting parts and we give an explanation of the three main cores of the system.

3.1 Software

3.1.1 Programming language

`AdaORM` has been implemented using Java 8 programming language. We chose this language with this version because it is enough evolved and diffused, that it has given us the opportunity to have access to many libraries that can improve my work. Also, some features as `lambda functions` aren't available in previous versions.

3.1.2 Database

Choosing the correct DBMS according to the case can improve the system performance. In our case of study the choice of DBMS is fundamental. To develop `AdaORM` we have chosen as host DBMS, the DBMS that stores information that we want to fetch, `IBM DB2`[10] because, as we can see in next chapter, it implements a *mandatory* feature that we exploited to achieve our goals.

However, AdaORM uses two DBMS, the first is the host database, IBM DB2, the second is the system database, SQLitethat contains the collected execution statistics. We chose to use SQLitebecause the database is not shared externally with any other process and using a database on the network would have led to problems due to network latency. Furthermore, it is preferable that the statistics are calculated in the application server in order not to load the database server.

3.1.3 Frameworks

A *framework* is a platform to develop software application. A framework provides an essential behavior that developers can exploit to build specific software. We have used Spring-boot to create an efficient and convenient web server to expose API. API is used from a client to start routines that load data from database, according to specific behavior. In this way, we can use Tsung tool to benchmark AdaORM .

3.1.4 Libraries

Using libraries to develop an application is the best solution. Using the libraries allows to decrease the development time, it makes the system modular, debuggable and safe. Libraries can also provide essential functions for the integration of some components such as a DBMS. These are the libraries used in the project

1. sqlite-jdbc version 3.30.1 allows us to interact with SQLitedatabase, our system database.
2. sqlparser version 3.1 implements methods to parse a SQL statement. [20].
3. db2jcc version 4.0 allows connection among AdaORM and IBM DB2 database, our host database.
4. All mandatory libraries to allow Spring-Boot to work.

3.1.5 Patterns

A pattern is a standard solution to a recurrent problem. Pattern helps improving application architecture. The application of these working methods

avoids needless bugs and problems. To design a good application we decide to use the following patterns.

Singleton: we made many singleton class. This pattern is very useful to store data as personal user queries, or to develop some tool class that stores status that must be shared uniquely. Also, using singleton instead of static classes allow to implement interfaces. Implementing interfaces are a key feature to improve code reusability and to decrease coupling.

DAO: Submitting database queries is possible thanks to the use of Data Access Object (DAO). We made this choice to centralize requests for each database. Also, we used a customized version of this pattern. In fact, we exploits the fact that each class that wants to use the prediction features must implement an interface. In this way, we are able to generalize object creation, asking to the user only to override two particular methods:

- makeBean(ResultSet rs) $\rightarrow T$
- getQuery() $\rightarrow String$

Implementing these two functions the user teaches to the system how to get the correct kind of object with the correct query. The DAO that we implement exploits two main functionality: *generalization* and *dynamic casting*. Each POJO must be extended by a decorator that implements interface `DBPredicted`. In this way, DAO is able to work with this type of object, calling `getParameters()` $\rightarrow$ `SmartMap` a methods that return a Map that has as key a string that represents the name of POJO property, and as value a wrapper that contains all information about the loading and storing methods to interact with the POJO property.

Strategy with lambdas: is a pattern used to split algorithm from data. In this way, we can reach the aim to develop classes with single responsibility and improve code reusability. However, for our purpose, it is not enough. We decided to use this pattern combined with lambda functions to model the getting procedures of a property in a class. When we load objects, it could happen that to improve performance, the application doesn't load all properties values. Then, to get the next requested values but not loaded, is necessary to develop a procedure that allows us to do it. This procedure

follows always the same procedure, but changes the applied functions that get and set object properties (in fact depend from which properties we want to fetch). To solve this problem, we have a main algorithm inside the class, and as parameters we pass getting and setting object methods. So, writing only a general class we can define the properties fetching method for all parameters that respect our constraints.

Factory: Factory pattern centralizes object creation. We exploit features of factory and DAO patterns to generalize object instance. The factory centralize methods to allow creation of collections of items. A method wraps a generic abstract DAO that requires the implementation of two methods:

- `makeBean()` → DBPredicted
- `getQuery()` → QuerySmart

`makeBean()` requires to be implemented so that it returns the item type that must be stored into the required collection. `getQuery()` requires to be implemented so that it returns a `QuerySmart`, a wrapper that contains `SQL` information to fetch collection from DB.

Decorator: Decorator plays a very important role in `AdaORM` design, because it allows to reduce inheritance explosion and adapt the functionality and data of an existing POJO to a new object that contains methods to talk with the system, without changing too many lines of the code in a (possible) existing project.

3.1.6 Main functionalities

We can split `AdaORM` into three main cores:

1. Object mapping and recording
2. Statistic computation
3. Query prediction

Now we explain all the three parts.

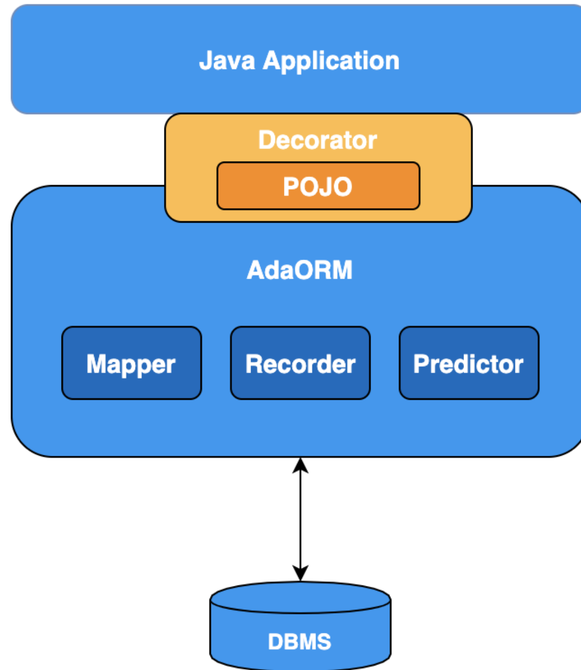

Figure 3.1: `AdaORM` architecture

Object mapping and recording

Object mapping and recording phase aims to favor mapping between data layer and POJO objects to load dynamically and *transparency* values from database. This phase isn't necessary meant only to get data, but also to trace client behavior and requests over the data. Making the mapping isn't a quick procedure, but it allows us to exploit the features that this mapping produce. First of all, the developer must know the schema of the database and the queries needed to interact with it. During application startup, a setup method is launched, which starts `AdaORM` engine so that the application can take advantage of the features available. The setup phase consists in two steps:

1. Host DB init: Setting host database into SmartEngine
2. Host queries init: Setting host queries to allow interaction

The first step accepts only a `DBManager` object, that implements the method to get the host database connection. The second step is also divided in two phases: phase one requires the definition of the main and complete query to load a target object the phase, storing class type and `SQL` statement. Phase two requires the definition of the objects that contain the information to fetch a single column. This object, called `QueryColumn` is a generics that require the column type and require as parameters reference table, column name, generic interrogation to fetch column of a generic row and if the column is a PK.

```
queries.add(
    new QueryColumn<String>(
        BookPlus.class.toString(),
        "Book_Title",
        "SELECT␣Books.title␣FROM␣Books␣WHERE␣Books.book_id␣=␣?"
));
```

Listing 3.1: An example how to store information to map links among POJO and table

Then, the developer can decor a standard POJO object with a special object that implements three methods from a mandatory interface `DBPredicted`:

- getParameters() → SmartMap;
- setIDCall(int index) → void;
- getIDCall() → int;

all three methods are necessary to interact with the ORM engine.

```
public class BookPlus extends Book implements DBPredicted {
...
}
```

Listing 3.2: An example of how to decor correctly a POJO

`getParameters()` returns a Map that contains as key the unique identifier for the column, and as content, an object called `DBColumn` that contains the previous key, the value type and lambda functions necessaries to get object identifiers and handles the properties value. Using these information we can generalize the execution flow, avoiding to ask developer to implements methods to interact directly with the engine.

The developer maps the methods to communicate with the object in a wrapper that will be used by the engine.

```
parameters.put(
    new ParameterRec<>(
        engine.getQueryColumn(this.getClass().toString(), "
           Book_Title"),
        this::getIDCall,
        item::getIndex,
        item::getBookTitle,
        item::setBookTitle,
        String.class
    ));
```

Listing 3.3: An example of how to map object properties to table columns

We decided to avoid inclusion of *Host queries init* information into decorated object to reduce coupling between classes and reduce the amount of code for class.

Statistics recording

Statistics recording is the part that allows predictions in `AdaORM` . In the previous section, we explained how the class property of a POJO are mapped with RDBMS columns. In this section, we explain when and how the statistics are recorded exploiting the previous mapping procedure. At the end, we talk about a possible bottleneck in the system.

The phases are the following:

- Object fetch
- Object column use
- Object column not used
- Query prediction

Object fetch: the client asks to fetch a result set from the database. Then, the `SmartEngine` gets the interrogation key and saves into the system db a new call, creating a new index. Also, `SmartEngine` saves into system db the columns that the predictor decides to load, linking them to the current index

query call. At the end, the query call index ia set into all fetched items. In this way we can memorize the purpose of the query and then increment the utilization of the right column.

Object column use: Recording is not applied only during loading phase. To model future predictions, it is necessary to understand and memorize how data is used. To achieve this results we exploited the previous mapping system. Each time that we request a property value, a driver is called: it checks if the property is set, if it is set returns the value, otherwise starts a routine to load the value using mapped query to perform the right query with object index to identify the right row. At the end, increments the utilization of requested columns with the current call index.

Object column not used: when an object column isn't used, its utilization remains zero. This is a fundamental behavior because we must store information that the columns do not use and so not useful for our execution.

Query prediction

This is the most important part. This is the part that allows us to really improve our performance and decrease system load. In fact, without this prediction core we could have a convenient behavior thanks to columns mapping, getting transparent loading procedure, such as standard ORMs configured in lazy loading strategies. But the records of the utilization are a waste of time and space. Then, we go inside the prediction core.

Prediction of a query starts when a client performs a request of a certain Result Set. Client asks to the `SmartEngine` to fetch the Result Set from a

given goal query. The algorithm follows the next steps:

Algorithm 1: Prediction algorithm

```
   Result: Q_R
1  if iteration_number mod cached == 0 then
2  |  C = Get columns from query Q;
3  |  R = ∅;
4  |  T = ∅;
5  |  C_s = getUsages(C);
6  |  while c_s ∈ C_s do
7  |  |  if isPK(c_s) || c_s.prob ≥ System.load || c_s.table ∈ T then
8  |  |  |  R = R ∪ c.name;
9  |  |  |  T = T ∪ c.table;
10 |  |  end
11 |  end
12 |  Q_R = new query with R as column set to query Q;
13 end
14 return Q_R;
```

In line 1. algorithm checks if the prediction needs to be refreshed. In fact, to reduce computation time we can predict a new query after some requests to have consistent and valuable changes. In lines 2., 3. and 4. it creates, if the condition in previous line is true, the two sets that contain columns and the set that contains table: C columns in the original query, R columns predicted and to use in the query prediction and T that contains the necessary tables to be scanned. Line 5. sets for each column the respective frequency utilization, creating a new set, C_s. Then, in rows 6., 7., 8. and 9. The algorithm performs for each column $c \in C_s$ a check to decide if column must be or not included in query to perform. If the column will be deemed necessary, it is included in set R and its table is stored in a set to know what tables we must use. In row 12. the column set R is the new column set that our DBMS fetches. In the last row (14.), it returns the predicted query. Remember that this value must be stored statically to allow caching,

avoiding useless computation.

Algorithm 2: Algorithm to get frequencies

```
   Result: C_s
1  C_s = ∅;
2  while c ∈ C do
3  |  frequency = getFrequency(c);
4  |  C_s = C_s ∪ (c, frequency);
5  end
6  return C_s;
```

In line 1. We create a new data structure where we store our result. In line 2. the while loop where for each column we want to get its frequency starts. Line 3. performs a database call that fetches from *view* into the database the already performed frequency. In the next step we save the column and its frequency into the C_s map. In the last step returns the data structure filled. We have used a *view* to maintain always updated the columns frequencies. In this way, when we want to fetch the requested values we have not execute a complex and expensive query, but we retrieve information faster because they are already computed.

Complexity: the complexity of the prediction algorithm depends from the number of columns that the submitted query has. In fact, to get column frequencies the methods complexity is $\Theta(|C|)$ and to check which columns load has a complexity always of $\Theta(|C_S|) = \Theta(|C|)$. Then we can conclude that the asymptotic complexity of `AdaORM` prediction algorithm is

$$\Theta(|C| + |C|) = \Theta(2|C|) = \Theta(|C|)$$

3.1.7 Conclusion

In this chapter we have seen how `AdaORM` has been developed. We have talked about databases, frameworks, libraries choice. We explained because we used some pattern and how they are integrated in `AdaORM` . Then, we have seen highlights code parts, for example how to write a POJO compatible with `AdaORM` and how to perform communication among decorated POJOs and system engine. At the end, we have shown prediction algorithm and we have talked about its complexity.

Chapter 4

Experiments and Analysis of AdaORM

Performing there test is a common way to describe a system, hardware or software. Exist four types of benchmark test: *performance regression testing*, *performance optimization testing*, *performance benchmarking testing* and *scalability testing*. Each of them has a different goal. We perform a performance benchmarking testing over different cases of study, implemented with `AdaORM` and with a static solution. The benchmarked solutions are described in terms of response time, free memory, CPU load and concurrent users at the system. In this chapter, we show how query execution time drastically decreases in DBMS that implement `join elimination` optimization when the statement has been written by pruning not useful tables. We assess through statistical methods if there are performance improvements. We analyze the queuing network system adopted using a theoretical model, finding the slowest station, the bottleneck, and the optimal number of users into the system. In the end, we give some prototype weaknesses.

4.1 Network architecture

In the following section, we describe the network architecture adopted and the hardware component that we used to perform the development and the test. This explaination allows the reader to better understand the results obtained in the chapter.

In modern software engineering, a client-server architecture is called mul-

titier architecture when the application processing layer and data processing are physically split.

The developing and testing of `AdaORM` is based on this architecture that is commonly adopted in a real systems. By this assumption, we are able to perform some interestings test with Tsung benchmark tool that show us how the system performance changes in a real architecture.

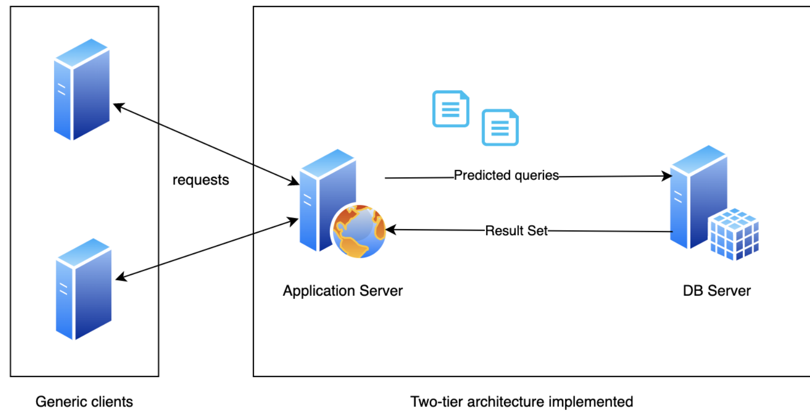

Figure 4.1: Two tier architecture implemented

In figure 4.1 we see how the system architecture has been developed. The left-hand panel contains some clients that perform requests to the application server to get results through the call of APIs. The right-hand side panel contains the developed system: the web server that contains an application that uses `AdaORM` and sends predicted queries (and column queries to fetch missing values) to the DB server.

Using this kind of architecture we can study the performance of our system. In fact, each application/db servers can be replicate many times for reliability or performance reason. In this way, if a node fails, our system can continue to work. However, in this thesis we did not focus over this aspect, letting the implementation of the system to developer preference and cases.

4.1.1 Application Server

In the previous section, we saw how the system architecture has been developed. Below, we explain the hardware components of machine that host the application.

Components	Characteristic
Device	Apple MacBook Pro 13"
CPU	2 GHz Dual-Core Intel Core i5 6th generation
Cache L1	32k/32k x2
Cache L2/L3	256k x2, 4 MB*
RAM	8 GB 1867 MHz LPDDR3
VRAM	1.5 GB
GPU	-
Secondary memory	250 GB SSD
Operative System	macOS Catalina 10.15.5 (19F101)

Table 4.1: Developing machine skills

4.1.2 Database Server

Components	Characteristic
Device	HP 630"
CPU	2,1 GHZ Dual-Core i3-2310M 2nd generation
Cache L1	-
Cache L3	3 MB*
RAM	DDR3 SDRAM (1066 MHz)
VRAM	-
GPU	-
Secondary Memory	250 GB 7200 rpm
Operative System	Ubuntu 19.10

Table 4.2: Test machine skills

Database server has been chosen to simulate a real system. In fact, often the secondary memory is implemented using a hard drive that allows for big storage capacity and high fault tolerance. However, in real systems, the storage can be implemented with a RAID system to improve the throughput of data fetch, but this consideration is out of our scope.

4.2 Databases Benchmark

In this thesis database benchmark is not a way to assess wich is the fastest DMBS, but a way to study how the presence of `join elimination` , the feature that prunes unnecessary joins, can drastically improve performance. First of all, we specify the three kinds of queries that we have performed.

1. Query type #1 (or Heavy query): is an optimized statement that calls at least one parameter for each table that has been requested. This query is the most expensive that we have decided to test.

2. Query type #2 (or Light query): is an optimized which requires to scan only a table. This is the cheapest query that we have decided to test.

3. Query type #3 (or Critical query): is a statement that calls only columns from the principal table inserted in `FROM` clause. This is a critical interrogation because, although it calls only columns from a single table without the needs to join or filter other tables, if the `join elimination` optimization is not implemented in DBMS, the execution time is close to that of the heavy query. Instead, if the DBMS implements this optimization, the execution time of the statement is close to the light query.

All queries introduce the `DISTINCT` keyword because if we do not use it the DBMS in any case scans and join all the tables specified in the query. In the case of `many-to-many` relationships, to provide a consistent and correct result, it will have to calculate the multiplicity relations of each row. Then, we assess that if a DBMS implements the `join elimination` optimization, the execution time for a critical query is closed to the execution time required by a light query. Otherwise, the execution time is close to that of the heavy query if the DBMS does not implement the optimization. The comparison has been done among different queries over the same data set. We never compared the execution time among databases because we are not interested in what database is the fastest, but we want to test how execution times change using the three types of queries described above, with or without the `join elimination` optimization. Also, for the moment, we do not talk about database size and the number of fetched queries because the experiment is consistent also without these assumptions.

In next sections, we analyze the DBMS benchmarks and `EXPLAINED PLANs`, a sequence of operations that the DBMS performs to run a SELECT, INSERT, UPDATE or DELETE statement, about three DBMS: `MySQL`, `Postgres` and `DB2`. We want to study how the system responds with different kind of queries, to prove that our handled query does not degrade the system performance.

4.2.1 Join Elimination optimization

The implementation of `AdaORM` largely exploits the `join elimination` optimization. `join elimination` is an optimization implemented in some DBMS that removes unnecessary `JOINs` to load the right required result set. Avoiding operations on not mandatory tables can considerably decrease the query execution time. `AdaORM` takes advantage of `join elimination` to delegate the removal of unnecessary tables in the query that it sends to the DBMS.

4.2.2 MySQL

MySQL is a relational DBMS developed by Oracle. It is a free software, one of the most popular RDBMS. We choose this software as a case study following some of its main features[11]:

- Client/Server architecture: one of the environments where it is necessary to manage a high number of requests and therefore it is necessary to optimize.
- Diffusion: given the strong diffusion, the experiment becomes interesting for a large number of users.

Experiment explanation

We have used the following frameworks to perform the experiment:

- RDBMS: MySQL version 8.0.19
- Benchmark tool: `mysqlslap` [16]
- Dataset: Sample Employees database [17]

`mysqlslap:` is a diagnostic command line application to simulate client load to a MySQL server. It allows us to get and report the execution time for each statement or stage. Using this technology we are able to get the execution time of our queries to understand how the system manages the critical query. `mysqlslap` has been customized with the following options:

- `-concurrency=1` calculate service time of a query with one user.
- `-iterations=50` set the number of experiment iteration.

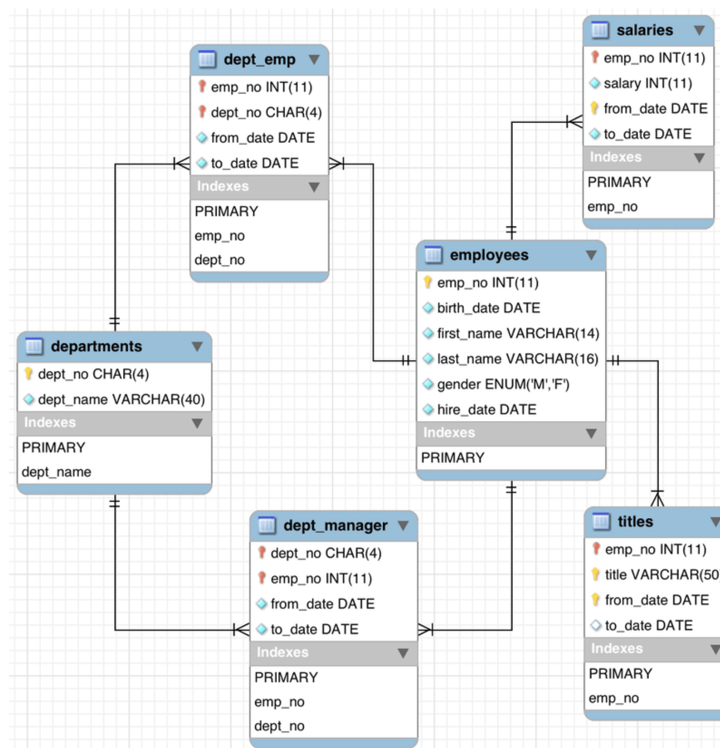

Figure 4.2: Data set employees schema

Then, now we start with benchmark execution and explanation.

Heavy query: all tables and almost one column for table.

```
SELECT DISTINCT
    employees.first_name,
```

```
    employees.last_name,
    employees.birth_date,
    titles.title,
    salaries.from_date,
    salaries.to_date,
    salaries.salary,
    departments.dept_name
    FROM employees
    LEFT JOIN titles on employees.emp_no = titles.emp_no
    LEFT JOIN salaries on employees.emp_no = salaries.emp_no
    LEFT JOIN dept_emp on employees.emp_no = dept_emp.emp_no
    LEFT JOIN dept_manager on employees.emp_no = dept_manager.
       emp_no
    LEFT JOIN departments on dept_emp.dept_no = departments.
       dept_no;
```

Listing 4.1: Example of heavy SQL query for MySQL. The format is the same for each heavy query for each other DBMS into the test

Light query: only one table and request column over this table.

```
SELECT DISTINCT
    employees.first_name,
    employees.last_name,
    employees.birth_date
    FROM employees;
```

Listing 4.2: Example of light SQL query for MySQL. The format is the same for each light query for each other DBMS into the test

Critical query: All tables are linked together but only the columns from the main table, `Employees`, are required. The execution time of this query is crucial to assess or reject the presence and the efficiency of `join elimination` optimization. If the execution time is close to the lower bound, we can think that the DBMS applies the optimization, if it is near to the upper bound, probably not.

```
SELECT DISTINCT
```

```
employees.first_name,
employees.last_name,
employees.birth_date
FROM employees
LEFT JOIN titles on employees.emp_no = titles.emp_no
LEFT JOIN salaries on employees.emp_no = salaries.emp_no
LEFT JOIN dept_emp on employees.emp_no = dept_emp.emp_no
LEFT JOIN dept_manager on employees.emp_no = dept_manager.
   emp_no
LEFT JOIN departments on dept_emp.dept_no = departments.
   dept_no;
```

Listing 4.3: Example of critical SQL query for MySQL. The format is the same for each critical query for each other DBMS into the test

Results

In 4.3 we can see the benchmark results recap expressed in seconds.

Statistic	Heavy	Light	Critical
Max	28.997 ms	0.528 ms	21.478 ms
Min	22.433 ms	0.169 ms	18.551 ms
AVG	23.754 ms	0.182 ms	19.068 ms

Table 4.3: MySQL benchmark with mysqlslap results

The table 4.3 tells us that probably `join elimination` feature is not present. In fact, the critical query execution time is close to the heavy query than the light. To assess this last statement we ask at MySQL to shows the `execution plan`. Execution plan is a set of physical operations that the DBMS must perform to fetch the correct request data set. We can get the execution plan from MySQL preceding the statement with the reserved keyword `EXPLAIN`. After this procedure, we assess that MySQL does not implement `join elimination` feature because the execution plan does not change between heavy query and critical.

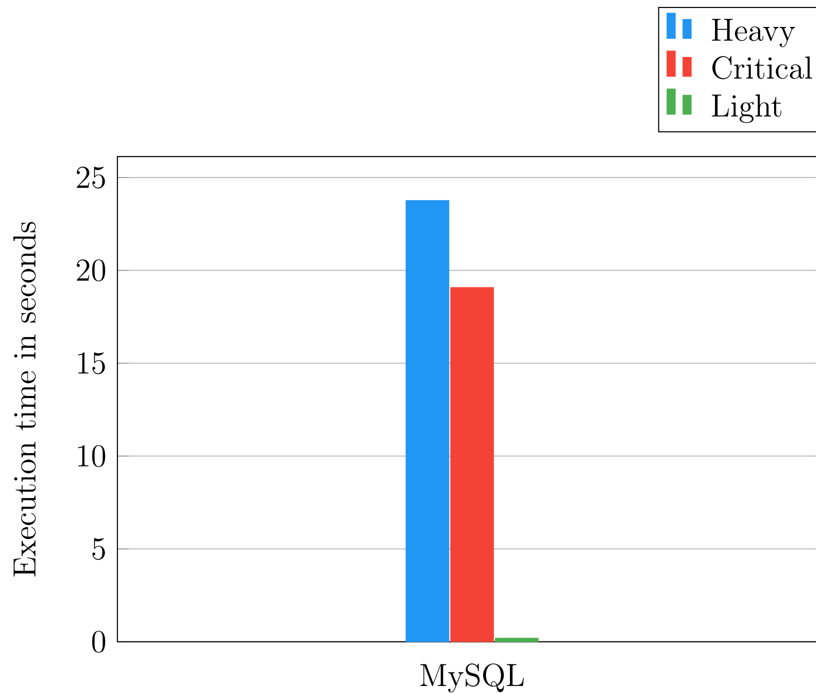

Figure 4.3: MySQL execution time between different queries

4.2.3 PostgreSQL

`PostgreSQL`, also called Postgres, is an open source object-relation oriented DBMS. Postgres uses SQL dialect to specify queries. We decide to perform an experiment with it because it is another widespread DBMS.

Experiment explanation

We want to assess if the current DBMS offers the `join elimination` feature and how its presence improves performance. To test the presence of this optimization, we use a demo database with demo queries such as in the previous experiment. The three queries are structured as described above in List 4.2.

Results

Also in this experiment we do not see any significant improvement from `join elimination` optimization. In fact, the time for a type 3 query is close to that of a type 1 query. In the tables below there is a recap of the avarage execution time. The results are expressed in milliseconds.

Statistics	Heavy	Light	Critical
AVG	56.893 ms	3.219 ms	38.404 ms

Table 4.4: Postgres benchmark results

Also in this case, we can see how the execution time of a critical query is close to the heavier. However, to assess our hypothesis, we can see the execution plan. To see the execution plan in Postgres we must precede our query with the reserved keyword `EXPLAIN ANALYZE`. As expected, Postgres does not implement `join elimination` optimization because continue to perform operation with critical query[1].

4.2.4 DB2

IBM DB2 is a DBMS developed by IBM. DB2 offers a tool suite that uses Artificial Intelligence technology to improve data management, structured and unstructured. IBM DB2 was developed to meet the needs of data warehouse. In this product, we find the functionality we want to exploit, the `join elimination` optimization. With the following experiment, we assess how `join elimination` improves the execution time getting the time of each execution and watching the `EXECUTION PLAN`.

Experiment explanation

This is the most interesting experiment. In fact, here we can observe the `join elimination` at works. To benchmark DB2 we used a sample Book database [4]. Also in this case we write three queries, one for each previously explained type.

[1] An article in jOOP blog [14], assesss that Postgres does not implement the searched features only for OUTER JOIN. In fact, form normal JOIN the feature might be present.

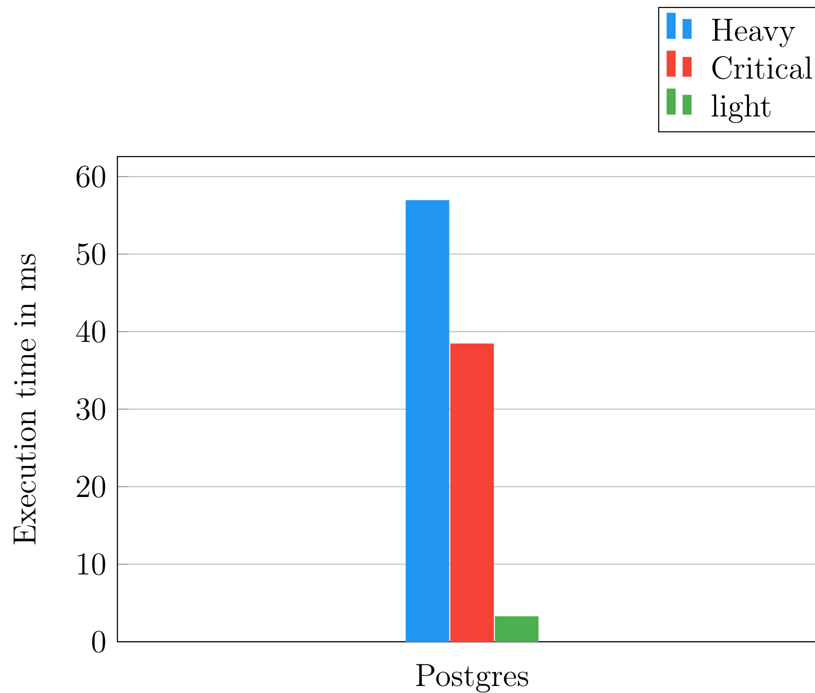

Figure 4.4: PostgreSQL execution time between different queries

IBM provides a command line tool to benchmark the database, `db2batch`. db2batch can be ran from command line environment setting the follow options (to see the full list, you see the official documentation[8]):

- `-d` set the database over run the test
- `-o` set an options
  - `o` set optimization level
  - `e` set
- `-q` set the query visibility
- `-f` set the file that contains SQL code to execute

The experiment is performed with the statement `db2batch -d demo -o o 9 -q del -o e yes -f query.sql`

The tool that the DB2 production company provides to see the `EXECUTION PLAN` is `db2expln`. Also in this case we have some options to set before ran the explanation (too read the full documentation see [9]):

- `-d` set the database over run the test
- `-statement` set the statement to explain
- `-terminal` set the terminal as standard output

The explanation is performed with the statement `db2expln -d demo -statement "<query>" -terminal`

Results

As expected, we can safely concclude that DB2 offers the `join elimination` optimization and that, removing not necessary queries, it provides a high improvement to the execution time.

Statistic	Heavy	Light	Critical
AVG	352 ms	19 ms	19 ms

Table 4.5: DB2 benchmark results

From Table 4.5 we can see that the blue bar that represents the heavy query execution takes about 18 times more than the execution time that light and critical queries take.

4.2.5 Conclusion

From the above experiments, we can assess that from the previous three DBMS, MySQL, Postgres and DB2, only *IBM DB2* offers `join elimination` . Also, we can prove that by introducing this feature a DBMS can avoid to perform not useful and expensive computation. As a consequence, we can assure that the join elimination is a very powerful optimization but that unfortunately is currently implemented only by the most important business-oriented solutions. However, this is not a problem. Our solution is designed for cases where the amount of data to be managed and the complexity of the system is particularly large. So, the adoption of a more performing database is already in itself a necessary condition.

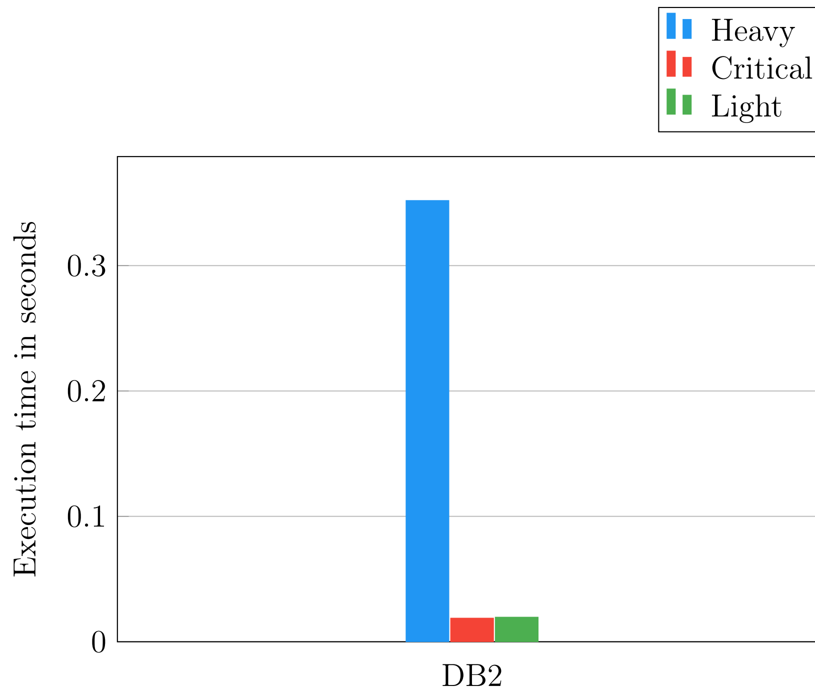

Figure 4.5: DB2 execution time between different queries

4.3 Application Benchmarks

In this section, we explain how we have performed the benchmarks over `AdaORM` configured with DB2 database. In the first part, we talk about the test configurations, then we describe the custom benchmark where we have compared `AdaORM` with static JDBC solution. In the third part, we show the plots and the results obtained through the use of Tsung [15] benchmark tool, comparing a web server that uses `AdaORM` with a web server, with the same network configuration, that implements a static solution with JDBC. To perform the previous benchmark, we used Spring-Boot to build quickly an efficient and reliable web server, where we write API to interact with DBMS. In the last benchmark that we perform, we compare `AdaORM` vs Hibernate ORM tool, showing some results and giving some observations. To assess the convenience of use `AdaORM` against a static solution with JDBC or Hibernate we have used statistic methods *hypothesis testing* explained in Chapter 2. In the last two sections, we explain the queueing network model adopted using

the metrics and model introduced in Chapter 2, to conclude giving the limits of proposed prototype.

4.3.1 Configuration

We include a test package where we implement some demo cases. In the demo, we write the class to link the host database, we design the POJO that represents the main database item (in this case the Book class). In the end, we write all the queries necessary to fetch the data from database. After we perform this configuration phase, we develop a main class where we simulate four different cases:

1. (`AHQ`) Adaptive heavy query
2. (`ALQ`) Adaptive light query
3. (`JHQ`) JDBC heavy query
4. (`JLQ`) JDBC light query

Each case takes its name from the type of query that is performed *to load the same goal result set* and from the technology used. Then, in each case, we want to fetch the same result set but using different strategies. `AHQ` case applies `AdaORM` to exploit its features. The case uses a heavier and more expensive query, but `AdaORM` understands, using collected statistics, that not all the columns are useful. So, it predicts a new and lighter query, that improves the performance. `ALQ` applies, also in this case, `AdaORM` features. Now, it starts using a light query, that loads less data than required. So, it must to fetch in lazy loading missing columns until `AdaORM` learns, using collected statistics, what is the correct dataset to fetch. `AHQ` and `ALQ` after the wrong prediction, they learn how to predict a faster query and, since they want the same result, execution times converge to optimal.

`JHQ` applies a simple static JDBC connection to fetch a big result set from database. Unfortunately, not all fetched columns are useful, but the system it is not prepared to manage this case. Then, at each call we have a waste of time and a higher system load. The last case is `JLQ` that applies a lighter query, but after the execution, at run-time, it needs to fetch other information, increasing the database requests, execution time and system load. `JHQ` and `JLQ` never converge in analyzed case to optimal solution. They are statically set to load always the same result sets, that are too big

or too small. Also, `JLQ` has a much longer execution time, with the same data set to be loaded. In fact, it will have to continuously connect to the database to load a missing data.

4.3.2 Database

The IBM DB2 database used is hosted into a the Linux machine previously described. The installation has been performed using a docker container, whose image has been pulled from the official repository in the docker hub[5]. The database accepts connections over port 50000. We see the loaded schema under exam is a book database [4] that contains the following tables with their cardinality in table 4.6.

Name	Cardinality
Books	1009
Publishers	193
Authors	1378
Genres	69
Books_Authors	1717
Book_Genres	3064

Table 4.6: Tables with their cardinality in Books database for IBM DB2

The figure 4.6 show us the UML[2] schema of the under test Book database.

Application

The application has been hosted in a machine with better performance than the machine that hosts database. The application has been executed from its `jar` wrapper. Using a wrapper configuration such as jar is a convenient way to move the application in different contexts, because contains all dependencies that the application needs.

Connection between DB and Application server

The connection between the two previous components, IBM DB2 database and `AdaORM` is performed through JDBC driver. The application sets the url that contains all information required to perform a well connection. The

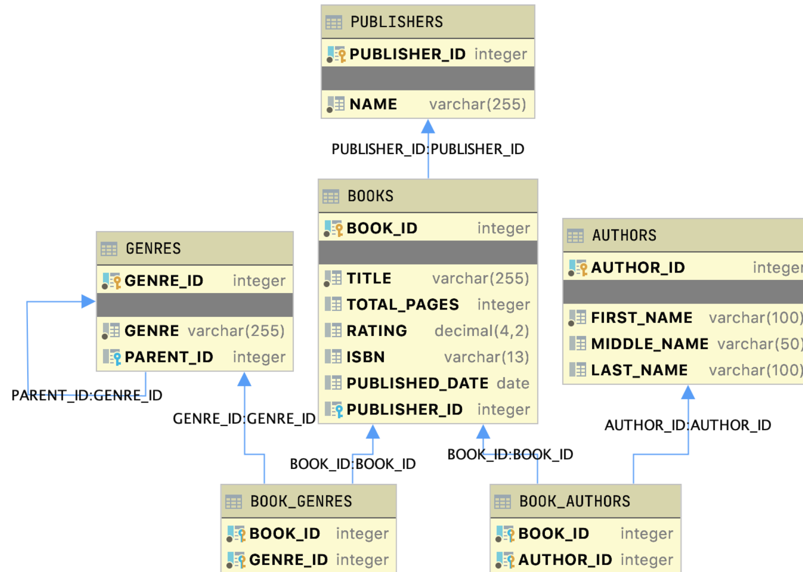

Figure 4.6: Book database UML schema

database and the application are hosted in two different machines in the same LAN. In this way we tried to simulate a real system configuration where we can find a DB server and a Web application server.

4.3.3 Custom benchmark

The first benchmark that we do is performed through a custom tests. In fact, we test different cases and measure the execution time of each case. The cases have been compared due to execution time, according to the start query to submit. For example, we match two configurations that want to reach the same result starting with the same query (i.e. a heavier query), one case uses `AdaORM` and the second static JDBC. Benchmark is started in a command line environment. The database is always in a different device and it is reached through the network. Timing has been recorded by the self class that performs the benchmark, through an object called `Chrono`. `Chrono` starts before the result set request (that will be returned as a list of object), and it stops after result set use. The test has been performed about 30 times.

We decide to split the benchmark in two parts. In first part we want

to understand how the execution time improves when AdaORM guesses the correct query to execute for the current context. In second part, we want to show how the excution time changs when the static choice is the right choice and AdaORM does not guess the right result set. AdaORM might not guess the correct query as soon as the behavior of the application changes. However, AdaORM will learn how to predict the new query from the collected statistics.

NB: In custom benchmark we decide to perform tests using less data from the result set (100 rows). In the light cases, the response times increase too much both for the solution managed with AdaORM and for the version implemented statically. In fact, the number of connections to be made subsequently to load the missing data is a high number. The inclusion of the test results on the whole result set would have made difficult to compare the various cases.

First part - Right prediction

In Table 4.7 we can observe the expected response time from the custom benchmarks when AdaORM guesses the right query, versus a static implementation, to retrive the same result set.

Type	AdaORM	JDBC
Light	80 ms	813 ms
Heavy	81 ms	280 ms

Table 4.7: Execution time when AdaORM guesses the query to execute

The response times in cases that use AdaORM are (about) equal because they fetch the same dataset after that AdaORM understands what is the best query to submit (we remember that the goal result set is the same for each case). We can observe that when AdaORM guesses the right query the system is 10 times faster in light mode and 3.5 times faster in heavy mode. We can observe better the previous results in Figure 4.7. AdaORM provides better expected response time. In the heavy case, AdaORM improves the starting query by eliminating the inconvenient columns and exploiting the `join elimination` to prune useless tables. In this way, the cost of the query will be lower, and consequently also the size of the result set and the response times of the system. Therefore, AdaORM can significantly lower the expected

response time predicting the right query in the light case, which starts appling the lazy strategy. In fact, it will not need to make further connections to the database.

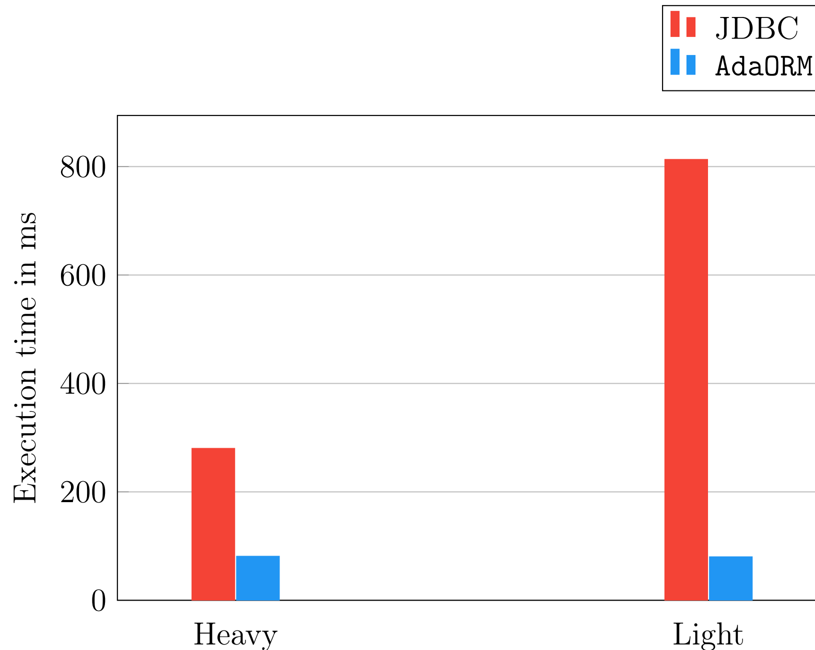

Figure 4.7: Home made benchmark comparison - guessed

It is interesting to see how the system works during the first executions. So, in next two-line plot (Figure 4.8 and Figure 4.9), we can see the evolution of the executions time before `AdaORM` predicts the right query and after its prediction.

From Figure 4.8 and Figure 4.9 is evident that the first execution is more expensive because the data set to be loaded is the largest, as in the static choice. Furthermore, with `AdaORM` , it is also necessary to carry out operations on the loaded data, which have a cost. However, after few iterations (if the prediction calculation is made at each iteration) `AdaORM` understands what is the best query to execute to fetch the right dataset according to the client behavior. In the end, at the first execution it is better to run the heavier query because, in case we need all the data, we should not make many connections to the database. In fact, we can assess from ligh cases that too many database

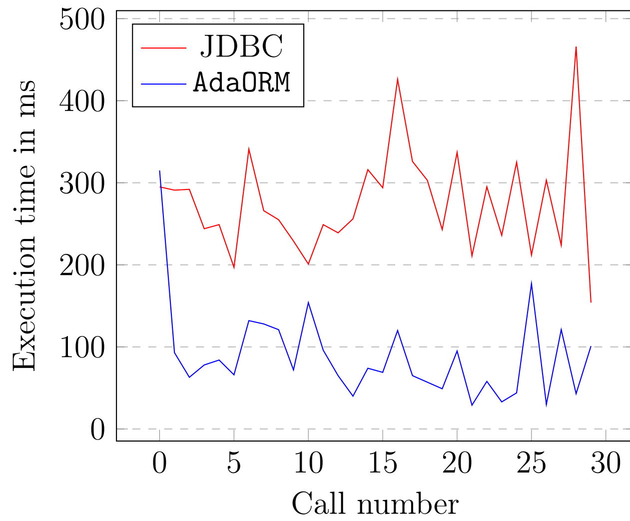

Figure 4.8: Execution time of queries in Heavy mode from custom benchmark

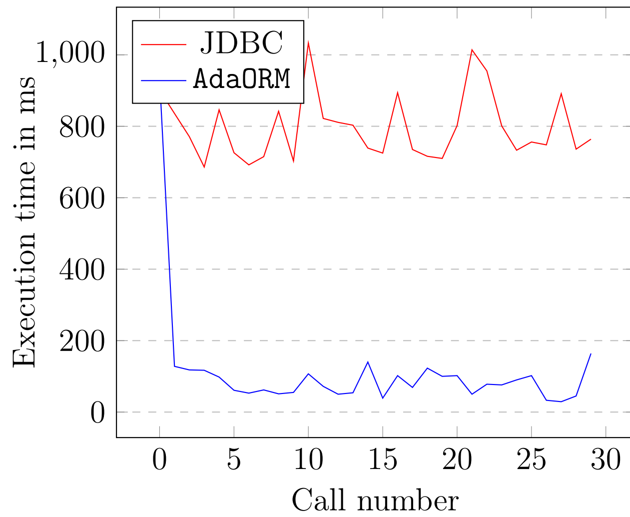

Figure 4.9: Execution time of queries in Light mode from custom benchmark

connections worsen performance.

Second part - Wrong prediction

Now, we analyze the expected response time before `AdaORM` guesses the right query. The guess is wrong because the predictor compute a query with less parameters and then `AdaORM` must load missing data. Notice that, at a certain point, the predictor will understand the new application behavior and will compute the right query.

Type	`AdaORM`	JDBC
Light	933 ms	813 ms
Heavy	302 ms	280 ms

Table 4.8: Execution time when `AdaORM` does not guess the query to execute

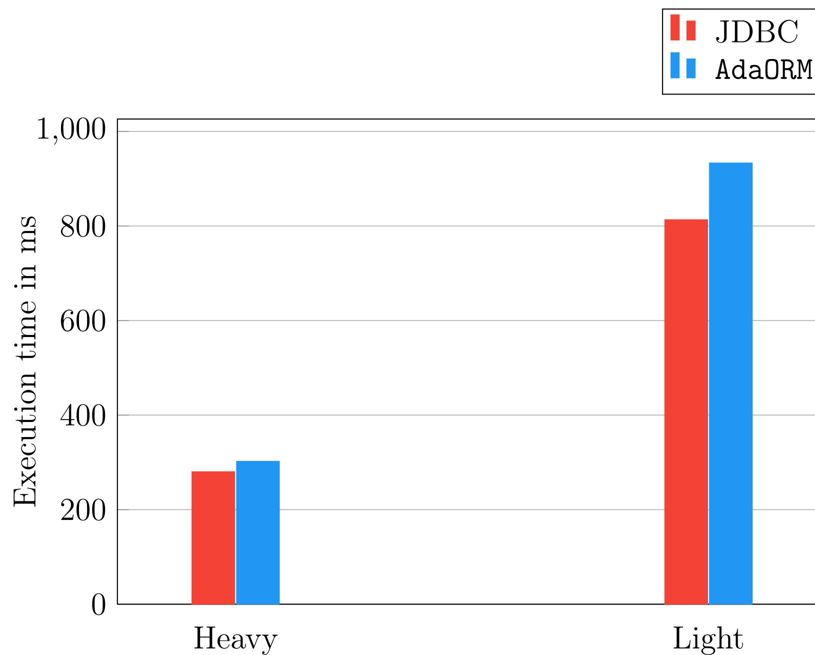

Figure 4.10: Home made benchmark comparison - not guessed

The static cases do not change because, by definition, the query that we want to perform is always the same. The most interesting values in the Table 4.8 are those obtained by `AdaORM` experiments. This is a very

costly procedure, but if the system does not change too quickly its behavior, `AdaORM` gives a good prediction. Also, we are sure that after some wrong prediction, `AdaORM` understands which columns are needed and predict the correct query.

4.3.4 Web Server benchmark with Tsung

Using a famous benchmark tool as Tsung, we want to assess that `AdaORM` improves the web server performance with respect to static JDBC case. In the following section, we will see some results obtained with the four configurations seen before: `AHQ`, `ALQ`, `JHQ` and `JLQ`. We will show the plots that describe *service time*, *CPU load*, *free main memory* and *concurrent users*, obtained stressing the system. Tests `AHQ`, `ALQ` and `JHQ` have been done with the following configuration:

- Queue system: Closed loop network
- Thinking time: 2 seconds
- Users: 300 u
- Duration: 600 seconds

while, test `JLQ` has been done with the next configuration:

- Queue system: Closed loop network
- Thinking time: 10 seconds
- Users: 60 u
- Duration: 600 seconds

the differentiation among the two tests is necessary to avoid system overloads in the JLQ case. In fact, it has a very long service time compared to the other cases and it needs to be managed differently.

The development of an efficient web server has been possible thanks a famous Java framework, `Spring-boot`[19], a convenient platform to develop stand alone *maven* or *gradle* [2] application with embedded web server such

[2] Maven and Gradle are two powerful project management tools, that can be used for building and managing any Java-based project

as *Tomcat*, *Jetty* or *Undertow*. With `Spring-boot` we develop a controller that, mapping *get* requests with methods, exposes callable API to activate back end routines. Back-end routines are our standardized work that `AdaORM` or the static solution perform.

Cases are slightly different from custom benchmark. We always request a result set that is be returned as a list of objects, and always performs operations over the result set. But now, methods start after a network request and the used result set has increased to its real size. This last configuration degrades the response time because we handle more data.

From Figure 4.11 and Table 4.9, we can see the *service time* performed by Tsung benchmark tool for each case analyzed. From this plot is evident that `AdaORM` responds fastest to requests than a static solution when guess the right query. So, a web server configured with `AdaORM` improves its service rate.

Cases	Heavy	Light
`AdaORM`	0.12 s	0.12 s
JDBC	0.38 s	7.02 s

Table 4.9: Service time recap in seconds

Looking the Table 4.9, it is clear that the static test of the light query on the entire dataset becomes unmanageable given its enormous response times. For this reason, we decided to not scale the graph in Figure 4.11 to improve plot readability.

For this experiment we only show the case when `AdaORM` guess the right query because, from the previous section we have seen how the system behaves in case the query is not guessed correctly.

Now we can see system behavior and response time of system under stress with and without `AdaORM` .

In Figure 4.12 and in Figure 4.13 we can see how the response time is lower in the case managed with `AdaORM` . Furthermore, it is interesting to see time peaks in the responses, which may coincide with the moments when `AdaORM` recalculates the statistics for the prediction. Even, in the static case we find peaks in response times. Peaks in static implementation are probably due to the presence of multiple customers within the system. Further, we will see the graphs corresponding to the competing customers in the system.

Also in the lighter case the response time in Figure 4.14 for `AdaORM` con-

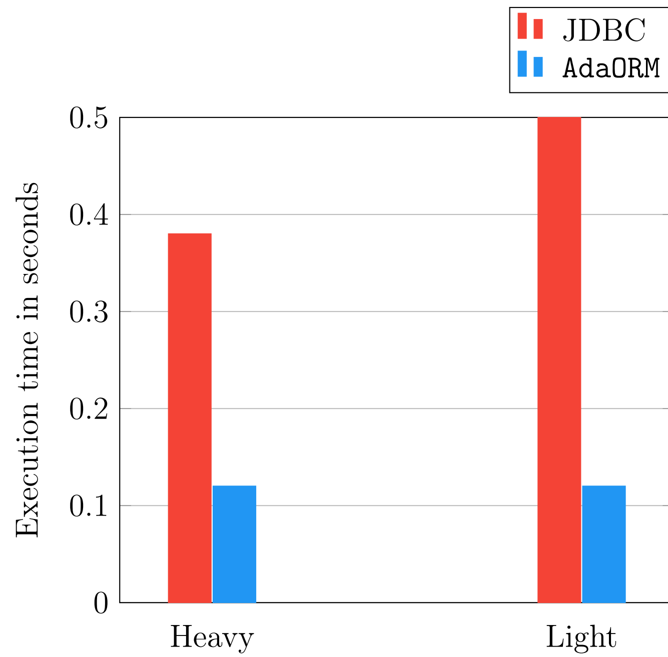

Figure 4.11: Tsung benchmark service rate comparison

Figure 4.12: System response time in AHQ

tinues to be lower than response time in the light and not managed case. In

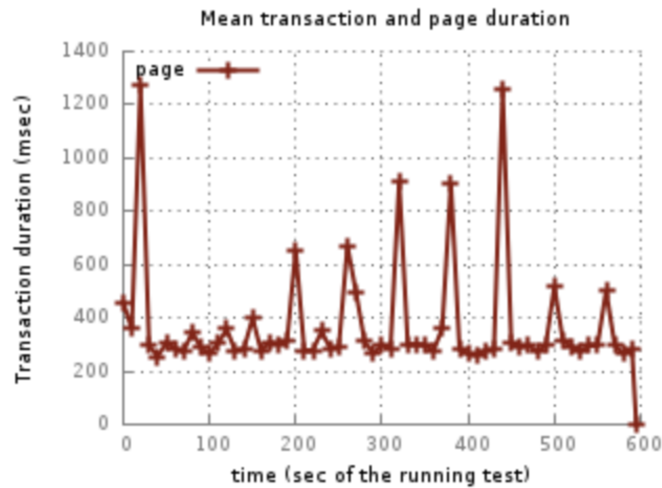

Figure 4.13: System response time in JHQ

fact, predicting the right result set `AdaORM` avoids to perform other database connections. Also in this case, peaks in response times are evident, probably due to the time spent in recalculating the statistics. In Figure 4.15 the situation is slightly different. The peaks in this graph represent a distortion due to the great expectation in response times.

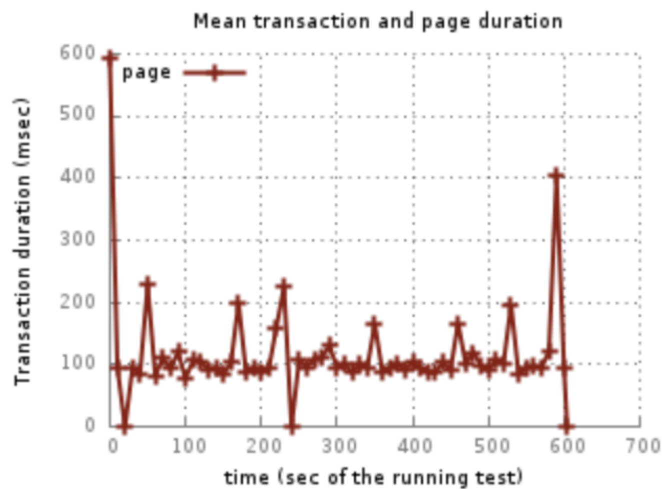

Figure 4.14: System response time in ALQ

Figure 4.15: System response time in JLQ

Now, to understand if we really managed to lighten the load on the system, we check the CPU load. We can see that in the two heavy cases (Figure 4.16 and Figure 4.16), the load for the processor is similar. Probably, for the moment, the statistics storage and computation cause a load that compensate the reduction of the loaded data set.

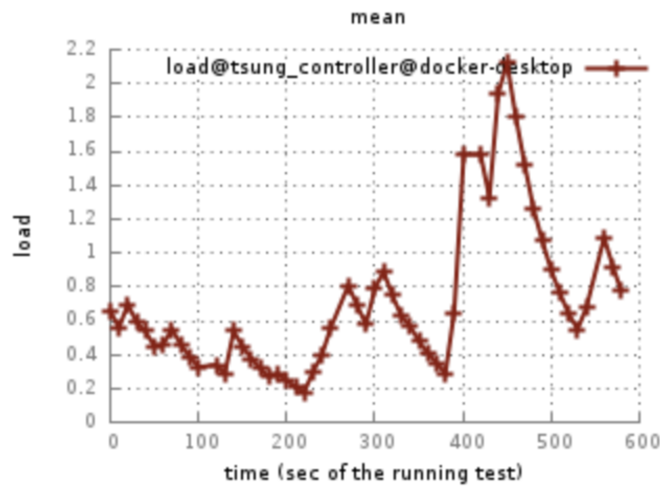

Figure 4.16: AHQ CPU load

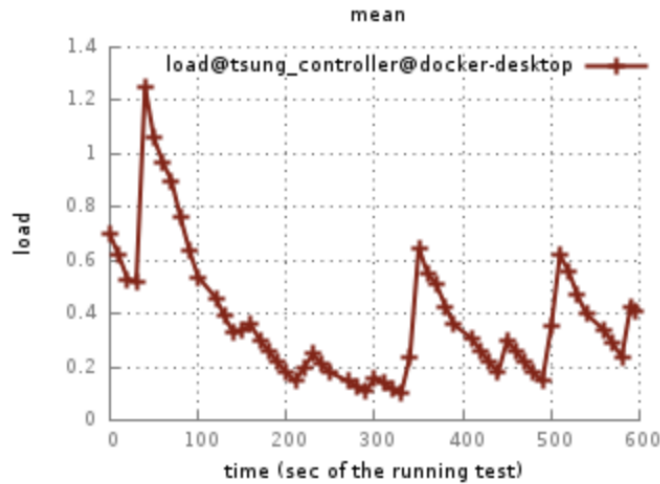

Figure 4.17: JHQ CPU load

In the light case for `AdaORM` (Figure 4.18) we have moment in which the system works intensively and moments when system is unloaded. Also, in this case, the peaks may be due to the calculation of the new predictions. The prediction of the right query allows us to avoid evident and not convenient requests both to the application server and to the database server, avoiding to make too many requests and thus obtaining high improvement in the response time. In the static case (Figure 4.19) session starts with a high load, then decrease, but subsequently it goes up again probably because of the accumulated work.

After the CPU usage, we analyze how the memory management behaves. In the heavy case for `AdaORM` (Figure 4.20), we see a slightly greater consumption of memory than the static solution (Figure 4.21). The cost of maintaining the managed solution has not yet been totally reduced by the gain provided the advantage of maintaining a smaller data set.

Instead, in the light case, it is clear that predicting the correct result (Figure 4.22) not only brings enormous advantages in terms of response times, but also in the amount of memory saved with respect to the static solution (Figure 4.23).

Finally, we analyze the graphs of competing customers within our system. Its evident that with `AdaORM` (Figure 4.24) the number of customers into the system is lower. The system serves customers faster, so they will have to

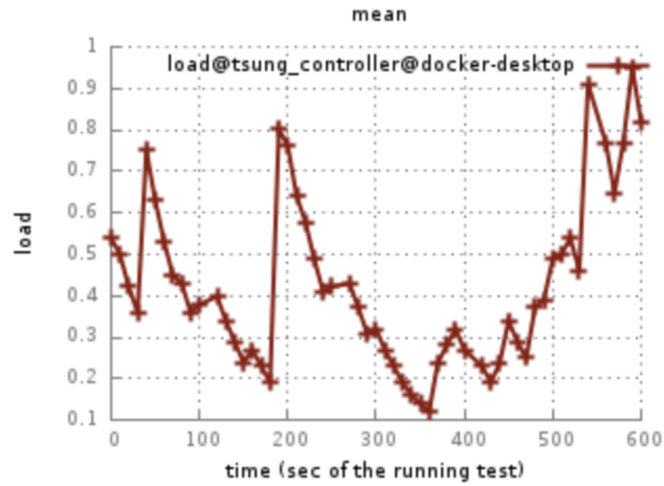

Figure 4.18: ALQ CPU load

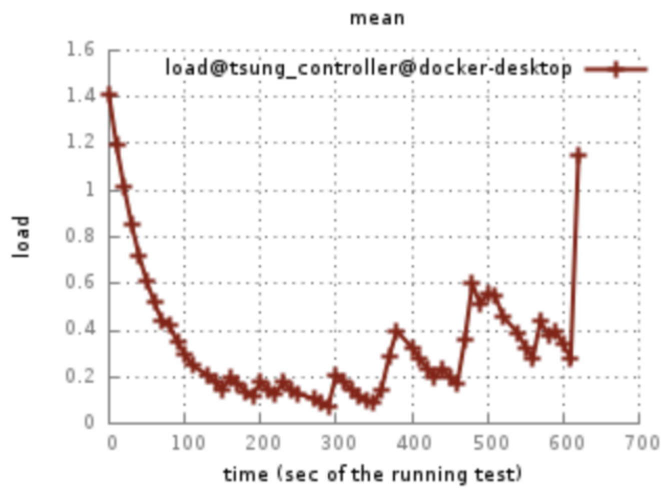

Figure 4.19: JLQ CPU load

spend less time within the system by *Little's law*.

To conclude the plot explanation, we analyze the last test. In the light cases, a system that uses `AdaORM` (Figure 4.26, guessing the right query, reduces response times. Then the number of concurrent users into the system decreases with respect to the static solution (Figure 4.27).

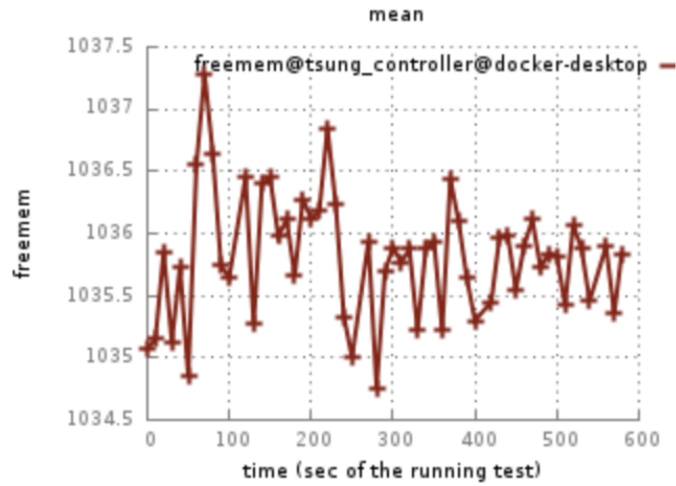

Figure 4.20: AHQ memory free

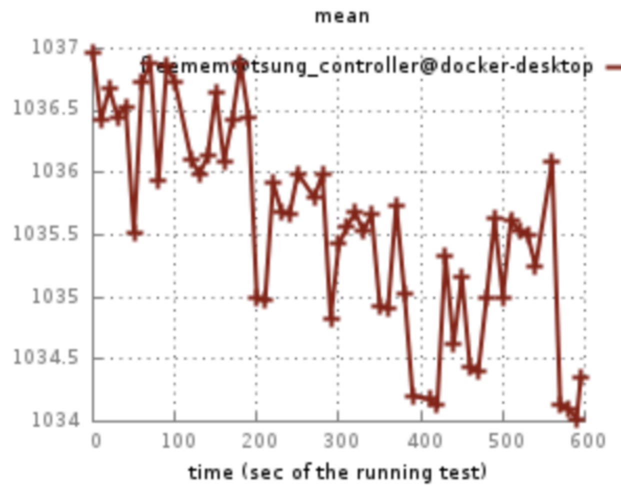

Figure 4.21: JHQ memory free

Assessing results

To conclude this benchmark test we decide to apply a hypothesis test to assess statistically that using `AdaORM` can improve the performance with respect to the static solution that may be wrongly configured. We test the null hypothesis $H_0 : T_{\texttt{AdaORM}} \geq T_{JDBC}$ against a *one-sided left-tail alternative*

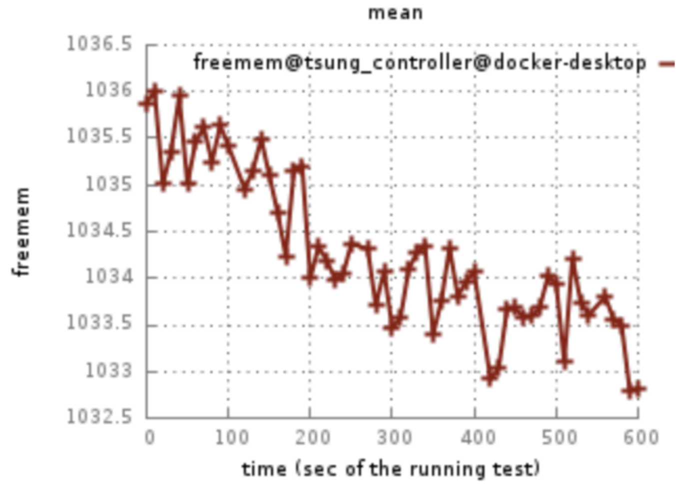

Figure 4.22: ALQ memory free

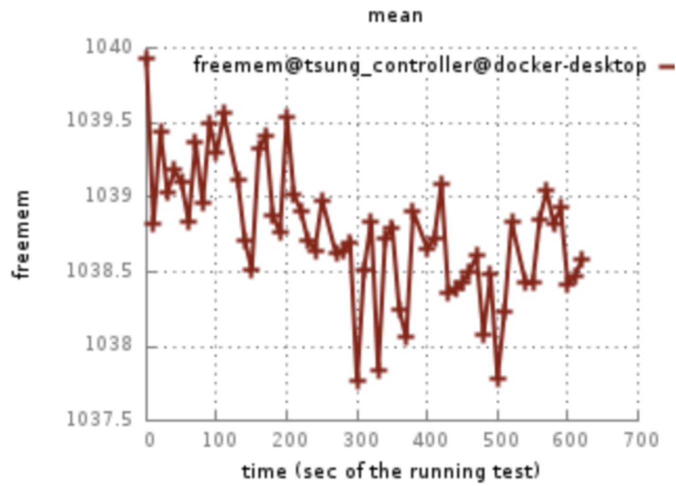

Figure 4.23: JLQ memory free

$H_A : T_{\texttt{AdaORM}} < T_{JDBC}$, because we are only interested to know if the *response time* has decreased.

$$H_0 : T_{\texttt{AdaORM}} \geq T_{JDBC},\ H_A : T_{\texttt{AdaORM}} < T_{JDBC}$$

We have that $T_{\texttt{AdaORM}} = 0.12$ *seconds*, $SE(T_{\texttt{AdaORM}}) = 0.042$ *seconds*, $T_{JDBC} = 0.38$ *seconds* and a significance level $\alpha = 0.01$.

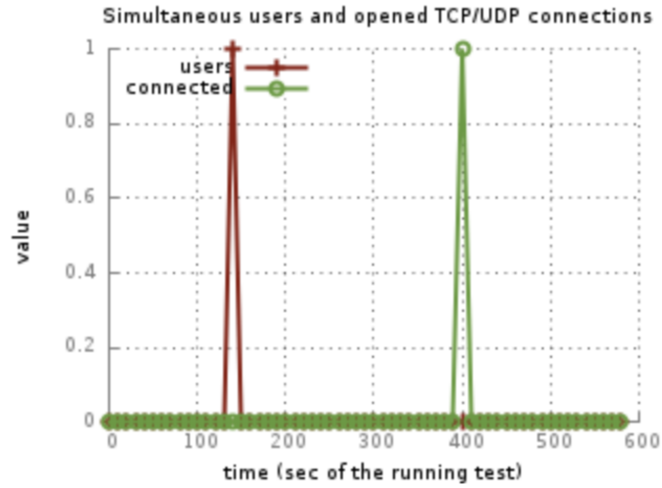

Figure 4.24: AHQ concurrent users

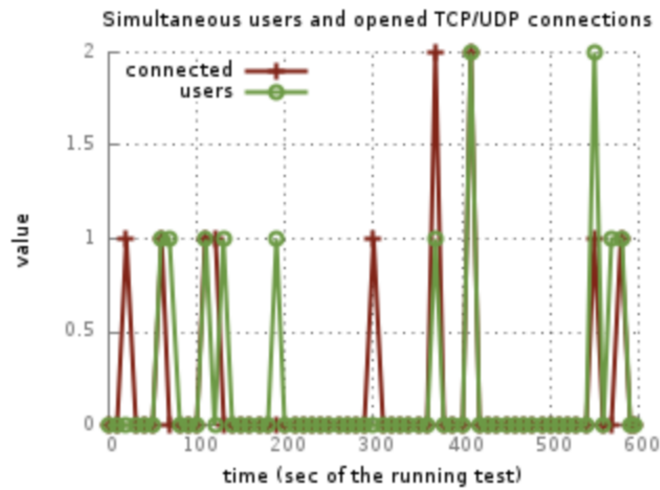

Figure 4.25: JHQ concurrent users

$$H_0 : T_{\texttt{AdaORM}} \geq 0.38\ s,\ \ H_A : T_{\texttt{AdaORM}} < 0.38\ s$$

Then we compute the test statistic

$$Z = \frac{T_{\texttt{AdaORM}} - T_{JDBC}}{SE(T_{\texttt{AdaORM}})} = \frac{0.12 - 0.38}{0.042} = -6.19$$

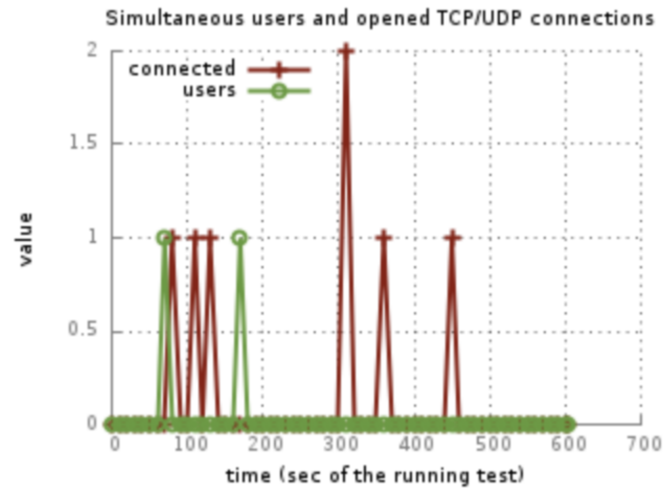

Figure 4.26: ALQ concurrent users

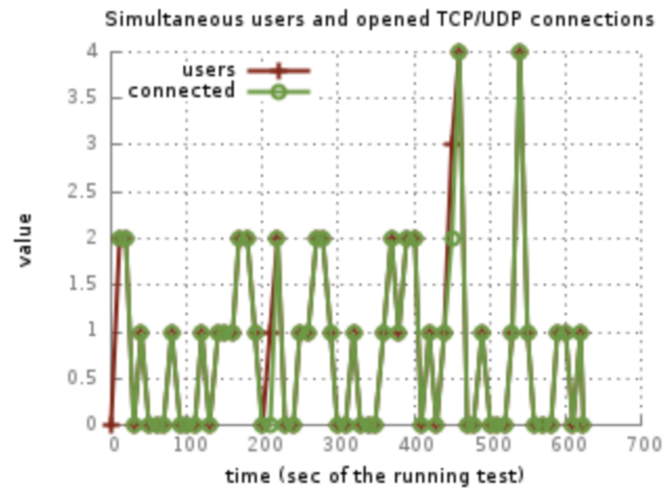

Figure 4.27: JLQ concurrent users

Well, now we calculate the acceptance regions

$$z_\alpha = z_{0.01} = 2.33$$

With a left tail alternative we

$$\begin{cases} reject\ H_0\ if\ |Z| \geq 2.33 \\ cannot\ reject\ H_0\ if |Z| < 2.33 \end{cases}$$

Then we assess that we can reject H_0 because Z falls into a rejection area. In fact

$$|Z| \geq z_\alpha = 6.19 \ \geq 2.33$$

Conclusion

In this section, we have seen a set of plots obtained with benchmarking experiments that shows the system behavior when it implements a predictive strategy with `AdaORM` or a static strategy with a simple JDBC connection. We have seen how service time decreases with `AdaORM` although not all the metrics in the system have improvements. To conclude our analysis between a Java application implemented with `AdaORM` or with an static JDBC solution we compared the obtained service time with hypothesis testing, that confirmed the improvements.

At the end, in Table 4.10 we can see a recap of all expected response time obtained from previous benchmark. We also include the expected response times obtained when `AdaORM` misses the prediction. The values are expressed in seconds.

Expected response time		
Cases	Heavy	Light
`AdaORM` (Guesses)	0.12 s	0.12 s
`AdaORM` (Misses)	0.40 s	7.58 s
JDBC	0.38 s	7.02 s

Table 4.10: Expected response time recap

4.3.5 Comparison of `AdaORM` and Hibernate

In this section, we show how `AdaORM` obtains better results than Hibernate in performance testing. To assess this statement, we get the service time from the same tests in Hibernate and we compare them with those previous obtained with `AdaORM` in the previous section.

To improve readability of experiments and analysis, we introduce new cases of study for Hibernate:

1. (PR) Prefetching
2. (LL) Light Lazy
3. (LLR) Light Lazy Request
4. (LE) Light Eager
5. (LER) Light Eager Request

Cases of study *require the same result set* but they apply different strategies. Also, we decide to test the cases when we apply `request()` methods. `request()` allows the program to load all properties of an object but, how we will see, it affects performances.

`PR:` it applies prefetching submitting a heavy query to retrieve all values from a database. It is equivalent to the static case JHQ seen previously, but implemented with Hibernate.

`LL:` it does not apply prefetching and its loading strategy is lazy.

`LLR:` it does not apply prefetching, its loading strategy is lazy and it applies the `request()` method.

`LE:` it does not apply prefetching and it uses eager loading strategy. This strategy provides a waste in terms of time and resources if the extra data fetched are not used.

`LER:` it does not apply prefetching and it uses eager loading strategy and `request()` method to load all data. This strategy provides a large waste of time and resource if the extra data fetched are not used.

In Table 4.11 we can see at the left-side panel the obtained results from Hibernate benchmarking, at the right-side panel the previously obtained results with `AdaORM` benchmarking.

From the Figure 4.28 is easy to see that cases in which we ask Hibernate to load all information about a result set, the response times grow exponentially.

H8 Cases	Expected exec time
PR	1263 ms
LL	558 ms
LLR	16017 ms
LE	1899 ms
LER	15896 ms

AdaORM Cases	Expected exec time
AHQ	120 ms
ALQ	120 ms

Table 4.11: Hibernate and AdaORM expected response time

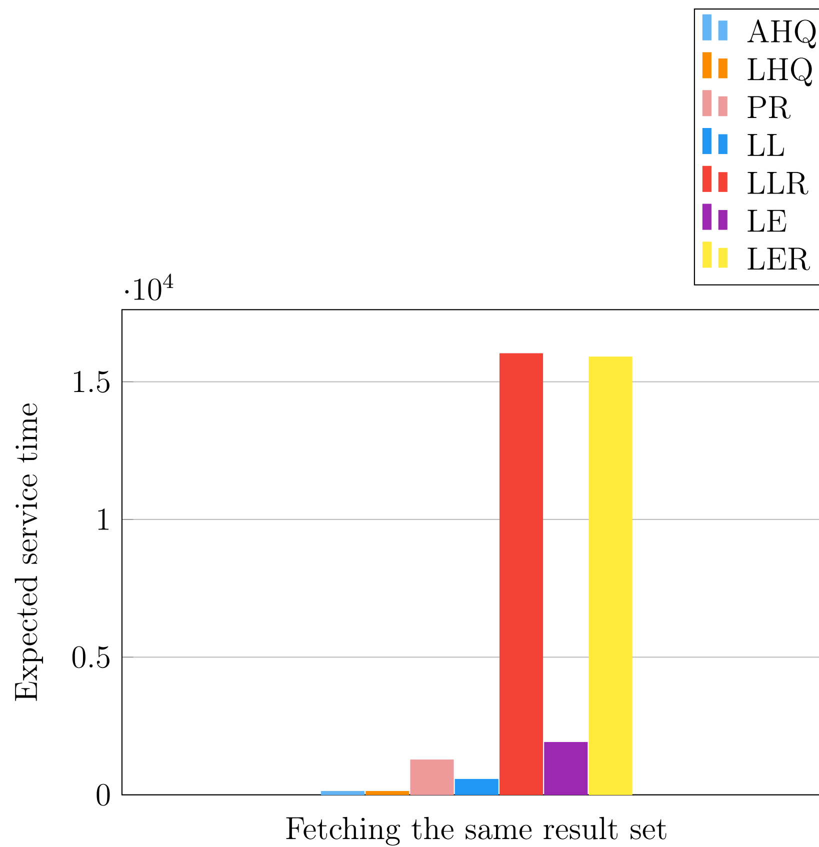

Figure 4.28: Hibernate and AdaORM benchmark comparison

Then is *mandatory* to apply a strategy to avoid this excessive waste of time[3]. We assess that the standard configuration of Hibernate (`LL`) is the one that comes closest to the results obtained by `AdaORM` for the cases tested.

Assessing the results

Finally, to conclude this last benchmark test we decide to perform a hypothesis test to assess that using `AdaORM` can improve performance. We test the null hypothesis $H_0 : T_{\texttt{AdaORM}} \geq T_{H8}$ against a *one-sided left-tail alternative* $H_A : T_{\texttt{AdaORM}} < T_{H8}$, because we are only interested to know if the *response time* has decreased. The two case that we analyze are `AHQ` for `AdaORM` and `LL` for Hibernate.

$$H_0 : T_{\texttt{AdaORM}} \geq T_{H8},\ H_A : T_{\texttt{AdaORM}} < T_{H8}$$

We have that $T_{\texttt{AdaORM}} = 0.12$ *seconds*, $SE(T_{\texttt{AdaORM}}) = 0.042$ *seconds*, $T_{H8} = 0.558$ *seconds* and a significance level $\alpha = 0.01$.

$$H_0 : T_{\texttt{AdaORM}} = 0.558\ s,\ H_A : T_{\texttt{AdaORM}} < 0.558\ s$$

Then we compute the test statistic

$$Z = \frac{T_{\texttt{AdaORM}} - T_{H8}}{SE(T_{\texttt{AdaORM}})} = \frac{0.12 - 0.558}{0.042} = -10.43$$

Well, now we calculate the acceptance regions

$$z_\alpha = z_{0.01} = 2.33$$

With a left tail alternative we

$$\begin{cases} reject\ H_0\ if\ |Z| \geq 2.33 \\ cannot\ reject\ H_0\ if |Z| < 2.33 \end{cases}$$

Then we assess that we reject H_0 because Z falls into a rejection area. In fact

$$|Z| \geq z_\alpha = 10.43\ \ \geq 2.33$$

[3] `request()` is a method that works only if the developer calls it. The default hibernate strategy is *lazy*

Conclusion

We can assess that `AdaORM` works better than Hibernate in the analyzed cases. However, we must say that these results should be considered carefully. In fact, Hibernate provides many features that we have not incorporated in `AdaORM` and then an absolute comparison is unfair. However, the conclusion that dynamic configuration of the fetching strategy can improve the performance drastically emerge from the experiments in a clear way. Indeed, we think that the inclusion of an adapting strategy can be a good improvement in object relational mapping tools as Hibernate.

4.4 Queueing system analysis

According to the definitions in Chapter 2, now we analyze the theoretical queueing model behind `AdaORM` [12]. In Figure 4.29 we can see the closed network designed to perform the previous tests and now defining metrics.

4.4.1 Adaptive Heavy Query

First of all, we define the analysis context. The parameters used in this analysis are obtained from Adaptive Heavy Query (AHQ) service rate described in the previous section. We assume that the service times are exponentially distributed with `FCFS` scheduling discipline. Under these assumptions, we can compute the expected response time and the throughput. Let the service time of AHQ be $T_{service} = 0.12$ seconds and thinking time $T_{thinking} = 2$ seconds. $T_{service}$ is the total time required by a customer to be served. Splitting this time in each station we obtain that the time spent over software layer (`AdaORM`) is $T_{sw} = 0.1$ seconds and time spent at the database is $T_{database} = 0.02$ seconds. According to those values we can start our analysis.

Variable	Seconds
$T_{thinking}$	$2.0\ s$
T_{sw}	$0.1\ s$
$T_{database}$	$0.02\ s$

Table 4.12: Expected service time at each station

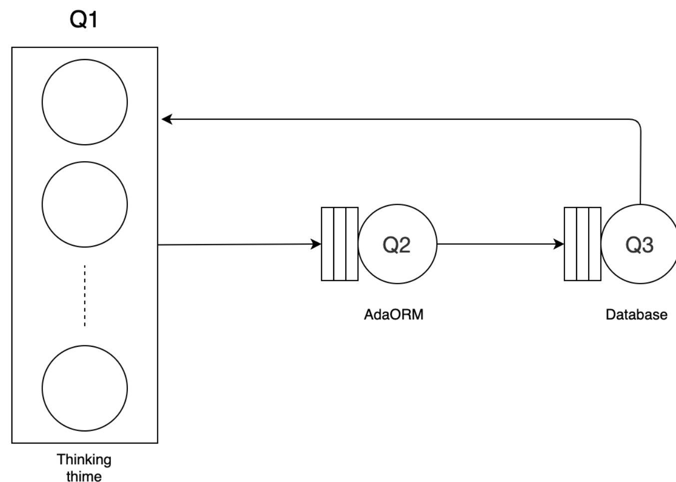

Figure 4.29: Queueing network architecture

We start our analysis defining our traffic equations

$$\begin{cases} e_1 = e_2 \\ e_2 = e_3 \\ e_3 = e_1 \end{cases}$$

Figure 4.29 represent a closed loop, then to calculate the performance indices we need to fix to 1 a station relative visit ratio. We choose the station Q_1, setting $e_1 = 1$. Then we have

$$\begin{cases} e_1 = 1 \\ e_2 = 1 \\ e_3 = 1 \end{cases}$$

According to the previous traffic equations, we calculate the service rate for each station

$$
\begin{aligned}
\mu_1 &= \frac{1}{T_{thinking}} = \frac{1}{2.0} = 0.5\ s^{-1} \\
\mu_2 &= \frac{1}{T_{sw}} = \frac{1}{0.1} = 10\ s^{-1} \\
\mu_3 &= \frac{1}{T_{database}} = \frac{1}{0.02} = 50\ s^{-1}
\end{aligned} \tag{4.1}
$$

and solving the next equation we can retrieve the service demand for each station to find the slowest component.

$$
\begin{aligned}
D_1 &= \frac{e_1}{\mu_1} = \frac{1}{0.5} = 2.0\ s \\
D_2 &= \frac{e_2}{\mu_2} = \frac{1}{10} = 0.1\ s \\
D_3 &= \frac{e_3}{\mu_3} = \frac{1}{50} = 0.02\ s
\end{aligned} \tag{4.2}
$$

from previous results we have not take into account the thinking time as possible bottleneck. Then the slowest component is the software layer, with service time is $D_b = D_2 = 0.1$.

Now, we want to discover what could be our maximum level of multiprogramming. Using the theoretical results deriving from operational analysis on the upper bounds of the throughput we can achieve our intent. We know these bounds are given by

$$X \leq min(\frac{1}{D_b}, \frac{N}{D+Z})$$

where D is the sum of each service demand, D_b the service demand of the bottleneck, N is the level of multiprogramming and Z is the thinking time.

Let $\bar{R} = 0.12$ seconds the service time. We obtain the *throughput* as

$$X = \frac{1}{\bar{R}} = \frac{1}{0.12} = 8.33 \ \frac{jobs}{s}$$

Let then T the *system time*

$$\bar{T} = \bar{R} + \bar{Z} = 0.12 + 2 = 2.12 \ s$$

Applying the bound $X \leq min(\frac{1}{D_b}, \frac{N}{D+Z})$ we find that the *optimal number of customers in the network* is

$$N_{opt} = \frac{\bar{D} + \bar{Z}}{\bar{D}_b} = \frac{0.12 + 2}{0.1} = 21.2 \approx 21 \text{ users}$$

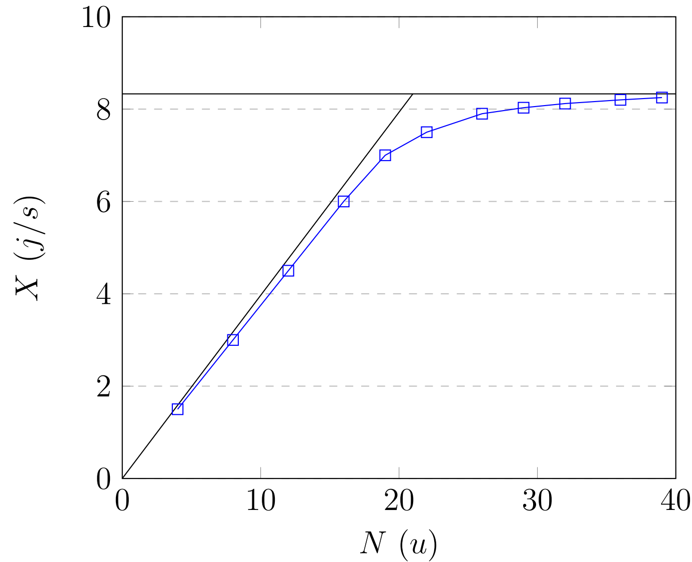

Figure 4.30: Throughput: plot of the bounds and the expected behavior for AHQ case

In Figure 4.30 and in Figure 4.31 we can see how we expect that the *throughput* and *response time* behave when the number of customers in the system increases. We see that the throughput is limited above by the maximum throughput $X = 8.33 \ j/s$ calculated with the previous analysis, the response time is limited by the response time $\bar{R} = 0.12 \ s$. By increasing

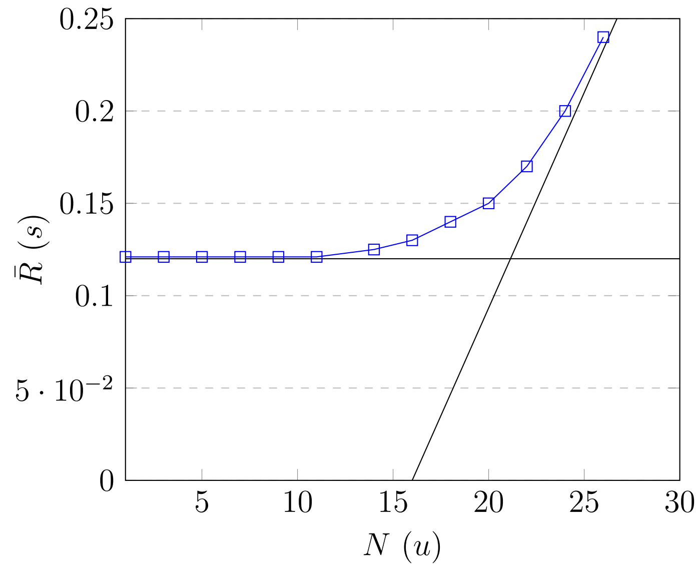

Figure 4.31: Expected response time: plot of the bounds and the expected behavior for AHQ case

the number of customers in the system, the throughput tends to the theoretical maximum in correspondence with the optimal number of customers $N_{opt} = 21\ users$, and then slowly grows up to the theoretical maximum limit. The response time remains stable close to the service time until N_{opt}, then begins to grow rapidly.

4.4.2 JDBC Heavy Query

Also, in JDBC Heavy Query we want perform a queueing network analysis. We use the following result to compare the two cases AHQ and JHQ.

$$
\begin{aligned}
T_{thinking} &= 2\ s \\
T_{service} &= 0.38\ s \\
T_{sw} &= 0.03\ s \\
T_{database} &= 0.35\ s
\end{aligned} \tag{4.3}
$$

service times exponentially distributed with FCFS scheduling discipline for JHQ cases. We define our traffic equations, that define a relation among the relative visit ratios in closed queueing networks.

$$
\begin{cases}
e_1 = e_2 \\
e_2 = e_3 \\
e_3 = e_1
\end{cases}
$$

And setting also in this case $e_1 = 1$ we have

$$
\begin{cases}
e_1 = 1 \\
e_2 = 1 \\
e_3 = 1
\end{cases}
$$

So, we obtain the next results

$$
\begin{aligned}
\mu_1 &= \frac{1}{T_{thinking}} = \frac{1}{2.0} = 0.5\ s^{-1} \\
\mu_2 &= \frac{1}{T_{sw}} = \frac{1}{0.03} = 33.3\ s^{-1} \\
\mu_3 &= \frac{1}{T_{database}} = \frac{1}{0.35} = 2.86\ s^{-1}
\end{aligned} \tag{4.4}
$$

and to get the bottleneck we must solve the follow equations, finding the slowest component

$$
\begin{aligned}
D_1 &= \frac{e_1}{\mu_1} = \frac{1}{0.5} = 2.0\ s \\
D_2 &= \frac{e_2}{\mu_2} = \frac{1}{33.3} = 0.03\ s \\
D_3 &= \frac{e_3}{\mu_3} = \frac{1}{2.86} = 0.35\ s
\end{aligned}
\tag{4.5}
$$

In this case the bottleneck is the database, then bottleneck service time is $D_b = Q_3 = 0.35$.

Let $\bar{R} = 0.38$ seconds the service time. We define the *throughput* as

$$X = \frac{1}{\bar{R}} = \frac{1}{0.38} = 2.63 \frac{jobs}{s}$$

Let then T the *system time*

$$\bar{T} = \bar{R} + \bar{Z} = 0.38 + 2 = 2.38\ s$$

In the end, the *optimal number of customers in the network is*

$$N_{opt} = \frac{\bar{D} + \bar{Z}}{\bar{D}_b} = \frac{0.38 + 2}{0.35} = 6.8 \approx 7 \text{ users}$$

In Figure 4.32 and in Figure 4.33 we can see how we expect that the *throughput* and *response time* behave when the number of customers in the system increases. Also in this case, we see that throughput is limited above by the maximum throughput $X = 2.63\ j/s$ calculated with the previous analysis, the response time is limited by the response time $\bar{R} = 0.38\ s$. By increasing the number of customers in the system, throughput tends to the theoretical maximum in correspondence with the optimal number of customers $N_{opt} = 7\ users$, and then slowly grows up to the theoretical maximum limit. The response time remains stable close to the service time until N_{opt}, then begins to grow rapidly.

To conclude, in the following table, we can see a recap of calculated values.

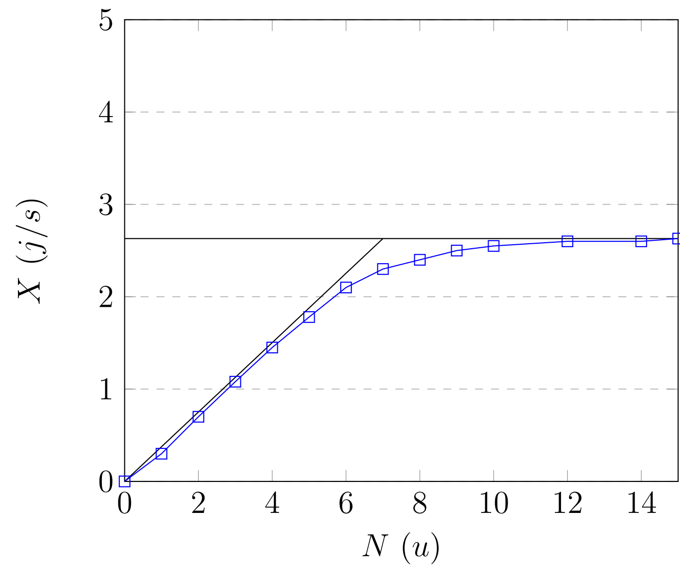

Figure 4.32: Throughput: plot of the bounds and the expected behavior for JHQ case

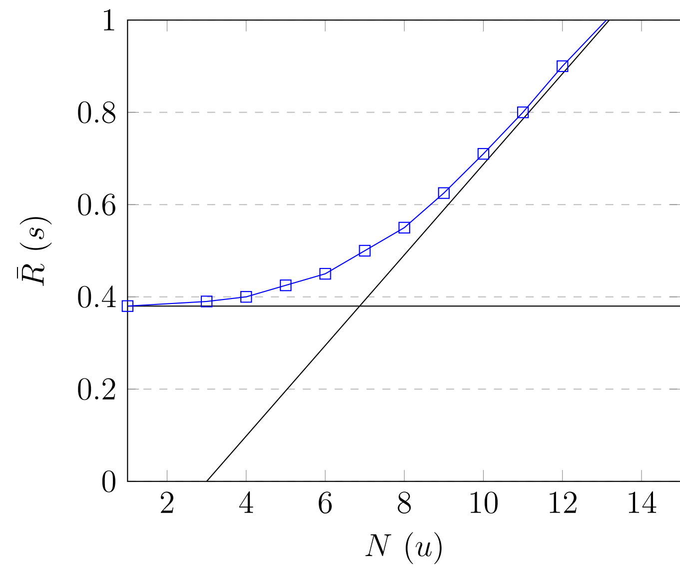

Figure 4.33: Expected response time: plot of the bounds and the expected behavior for JHQ case

Metrics	AdaORM	JDBC
$T_{service}$	$0.12\ s$	$0.38\ s$
$T_{thinking}$	$2\ s$	$2\ s$
T_{sw}	$0.1\ s$	$0.03\ s$
$T_{database}$	$0.02\ s$	$0.35\ s$
D_b	$0.1\ s$	$0.35\ s$
$\bar{R}$	$0.12\ s$	$0.38\ s$
X	$8.33\ j/s$	$2.63\ j/s$
T	$2.12\ s$	$2.38\ s$
N_{opt}	$21\ u$	$7\ u$

Table 4.13: Queue analysis recap

4.4.3 Conclusion

From the two previous analysis we obtain that the optimal number of users that the system can manages improves. So, we can assess that the proposed solution *improves the scalability* of the system and the response time. From the analysis it is clear that we have moved the bottleneck from the database to the application layer, by sending a lighter query. Furthermore, by having the bottleneck on the application layer, we have the possibility to improve response times by improving the AdaORM code. That means that a java application that exploits AdaORM increases the system scalability and reduce the response time with respect to the static solution.

4.5 Prototype limits

Before analyzing the limits of the presented tool, we say that AdaORM is a *proof of concept* and not a complete tool. Then, we are conscious that AdaORM it could be developed better. However, this current version is enough performing to increase the system response time. So, AdaORM fulfills our purposes.

In previous sections we saw how the system has been developed: the network architecture thought to simulate a real critical environment and the most important and critical parts of the framework. Then, we saw the behavior under stress with comparison among different configurations of the cases of study. Now we are ready to understand what are the limits of this

type of implementation.

4.5.1 Concurrency

AdaORM has been developed without considering big improvements given by introduction of concurrency. The only threads introduced were used to record statistics asynchronously into the system database. Introducing concurrency at beans creation or prediction computation can improve response time.

4.5.2 System database

SQLiteis not a client-server database engine. It is though to be included into a client program, but its application in this case can be enough performaning. However, if we would implement concurrency parts, we should change DBMS because SQLite does not suggest to manage high degree of parallelism[18].

4.5.3 Usability

Hibernate implements mapping using annotations. Also, we can find some tools that, given the database schema generate the respective class with annotations. AdaORM is a very young ORM project born to assess if is possible trying to guess the best result set through the collected statistics.

Given this observations is "normal" that there are some imperfection and its integration with an RDBMS and java application is slow. The mapping between POJO object and database tables can be improved, but this improvement would not changes the goals of this project.

In this chapter, we have seen some cases where is convenient use AdaORM and other where AdaORM perform worse than a static solution. We have seen how a good prediction can boost our execution time, and we have also seen how a wrong prediction can gives us a little or a big waste of time. However, the algorithm guarantees us that in case the program's "habits" change by making us make a mistake, AdaORM will learn how to adapt to the current behavior, offering us again high performance and avoiding long and complicated code maintenance.

Chapter 5

Conclusions and Future Works

In this thesis we have addressed the problem of the dynamic configuration of ORM tools. We have proposed `AdaORM`, a prototype tool that decides policy of data fetching at runtime, according to the user behavior. To assess the convenience of the tool we have performed many experiments, performance tests and we have compared the results with state of art solutions.

It is evident that, using `AdaORM` , the response time and the system performance improve when the predictor guesses the right query to submit. The great improvement is given in that by guessing the right result set, the system will not need to make new connections to the database, or to retrieve unnecessary data, reducing response times. Instead, when `AdaORM` misses the query to execute, especially in the case in which it loads a smaller dataset, it will have large delays, than the static solution, due to the multiple database connections necessary to collect usage statistics. If it loads a larger dataset, we will still have delays comparable to the static solution since the service times are dominant by the application needs.

However, we know that is always do better. This development is only a proof of concept to assess that is possible improve execution time and system load handling the statement before its execution and exploiting `join elimination` optimization.

Developing the project we realized that it is possible to introduce interesting and convenient optimizations to improve the impact of the ORM in the system and improve the data prediction.

In conclusion, the dynamic decision of the policy of fetching is very useful in those cases in which the programmer cannot know which will be the usage pattern of the application and hence the static decision may lead too poor

performance. The overhead introduced by predictor is tolerable where the application is data intensive while it may become heavy otherwise.

5.1 Data prediction

Proposed work 1: Collecting many information about execution and data utilization, we can apply different data mining algorithm. For example, we can find a pattern over the data using `PaNDa+` algorithm [13]. The pattern will be the set of columns called together most frequently. Balancing the thresholds using the system load, we can shrink our prediction: with higher system load we will allow less noyse, with lower system load we will allow more noyse, then more columns.

Proposed work 2: An other possible prediction implementation could be the use of a machine learning algorithm. We can use an algorithm that can learn when we need a particular set of columns.

Proposed work 3: Another interesting feature that could be implemented is the introduction of the clustering of statistics to make the predictor more flexible to change. In fact, calculating the predictions on many values, if the behavior of the system changes, the change in the predictions will not be as fast.

Proposed work 4: In the end, the last possible future work that we want to suggest changing the core of the prediction from columns to the tables. In fact, after the prediction of a column, we require the fetching of all requested columns that belongs to the same table because it is more convenient load now the information. Changing the observed elements, the granularity of the system changes but the predicted result set does not change. In this way, we can store less data and the predictor will do less computation.

5.2 Cost impact

Proposed work 1: By introducing asynchronous calls and promises, we can improve system performance. In fact, while `AdaORM` retrieves information

from the database, it is possible to use another thread (or an asynchronous call) to start composing an object.

Proposed work 2: In the current version, the prediction will be performed after its request, caching the results for 10 requests before re computation. However, computing the prediction after its request can decrease the speed of the system. To avoid this issue, it is possible an *offline prediction*. An *offline prediction* is a prediction performed not after its requests, but after the fetching of the result set using the cached predicted query. Also, we can plan the new query computation by keeping in mind the system load: when the system is unloading we can perform the planned execution after a less number of requests.

Proposed work 3: Applying the Data prediction proposed work 3 we can reduce the cost to perform prediction. In fact, $|tables| \leq |columns|$, then, the asymptotically cost of the prediction function change in terms of cardinality from columns to tables.

Chapter 6

Acknowledgements

This has been a long and demanding course of study, but I have come to its end. Every single step towards the achievement of this title has cost me effort and sweat. I must thank my determination, my commitment, my organization and my resilience for having supported me every single day of the past few years to get where I am here today.

I want to thank my supervisor Andrea Marin, who has patiently followed me remotely, with all the difficulties of the case, in my latest interesting and stimulating academic project. Also, I want to thank him since he accepted me to work with him. I also want to thank professor Alvise Spanò, whose deep knowledge in software architecture, has given me some excellent advice on how to model `AdaORM` . Thanks also to Stefano Calzavara, who made himself available in giving me tips about DBMS problems. Thanks to my boss, who gave me a lot of flexibility to follow my studies. Thanks to my family, who gave me the opportunity to attend this academic path and and they bear me even in my worst moments.

Last but not least, all my friends, who have always been there, who supported me in difficult times and with whom we celebrated in moments of joy. I want to thank all my fellow students with whom I spent entire days, first shoulder to shoulder, then by teleconference, to complete projects and to prepare exams. Teamwork has made us stronger.

This is not only the end of a course of study for me, but this is the finish of an era. From tomorrow my life will never be the same, again. But, I can't wait to face all the new challenges that life has to offer me.

Thank you again, Paolo.

List of Figures

1.1 Hibernate architecture 9

2.1 UML representation of Strategy pattern 18
2.2 UML representation of Decorator pattern 19
2.3 UML representation of DAO pattern 20
2.4 Acceptance and rejection regions 28

3.1 `AdaORM` architecture 35

4.1 Two tier architecture implemented 42
4.2 Data set employees schema 46
4.3 MySQL execution time between different queries 49
4.4 PostgreSQL execution time between different queries 51
4.5 DB2 execution time between different queries 53
4.6 Book database UML schema 56
4.7 Home made benchmark comparison - guessed 58
4.8 Execution time of queries in Heavy mode from custom benchmark 59
4.9 Execution time of queries in Light mode from custom benchmark 59
4.10 Home made benchmark comparison - not guessed 60
4.11 Tsung benchmark service rate comparison 63
4.12 System response time in AHQ 63
4.13 System response time in JHQ 64
4.14 System response time in ALQ 64
4.15 System response time in JLQ 65
4.16 AHQ CPU load 65
4.17 JHQ CPU load 66
4.18 ALQ CPU load 67
4.19 JLQ CPU load 67

4.20 AHQ memory free 68
4.21 JHQ memory free 68
4.22 ALQ memory free 69
4.23 JLQ memory free 69
4.24 AHQ concurrent users 70
4.25 JHQ concurrent users 70
4.26 ALQ concurrent users 71
4.27 JLQ concurrent users 71
4.28 Hibernate and AdaORM benchmark comparison 74
4.29 Queueing network architecture 77
4.30 Throughput: plot of the bounds and the expected behavior for AHQ case 79
4.31 Expected response time: plot of the bounds and the expected behavior for AHQ case 80
4.32 Throughput: plot of the bounds and the expected behavior for JHQ case 83
4.33 Expected response time: plot of the bounds and the expected behavior for JHQ case 83

List of Tables

1.1 Matching principal ORMs 11

2.1 Comparison between SQL vs NoSQL 17
2.2 Recap of different type of alternative hypothesis 28

4.1 Developing machine skills 43
4.2 Test machine skills 43
4.3 MySQL benchmark with mysqlslap results 48
4.4 Postgres benchmark results 50
4.5 DB2 benchmark results 52
4.6 Tables with their cardinality in Books database for IBM DB2 55
4.7 Execution time when `AdaORM` guesses the query to execute . . 57
4.8 Execution time when `AdaORM` does not guess the query to execute 60
4.9 Service time recap in seconds 62
4.10 Expected response time recap 72
4.11 Hibernate and `AdaORM` expected response time 74
4.12 Expected service time at each station 76
4.13 Queue analysis recap 84

Bibliography

[1] Antonio Albano, Dario Colazzo, Giorgio Ghelli, and Renzo Orsini. Relational dbms internals, 2015.

[2] Sinan Si Alhir. *Learning Uml*. " O'Reilly Media, Inc.", 2003.

[3] Michael Baron. *Probability and statistics for computer scientists*. CRC Press, 2014.

[4] Db2Tutorial. Db2tutorial - book database. https://cdn.db2tutorial.com/wp-content/uploads/2019/06/books.zip.

[5] Docker. Docker. https://www.docker.com/.

[6] J. Dollimore G. Coulouris and T. Kindberg. *Distributed Systems: concepts and design*. Addison Wesley Professional, 5 edition, 2011.

[7] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[8] IBM. db2batch - benchmark tool command. https://www.ibm.com/support/producthub/db2/docs/content/SSEPGG_11.5.0/com.ibm.db2.luw.admin.cmd.doc/doc/r0002043.html.

[9] IBM. db2expln - sql and xquery explain command. https://www.ibm.com/support/knowledgecenter/SSEPGG_11.1.0/com.ibm.db2.luw.admin.cmd.doc/doc/r0005736.html.

[10] IBM. Ibm db2 – data management software. https://www.ibm.com/analytics/db2.

[11] Javapoint. Mysql features. https://www.javatpoint.com/mysql-features.

[12] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[13] Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. A unifying framework for mining approximate top-k binary patterns. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):2900–2913, 2013.

[14] Lukaseder. Join elimination: An essential optimiser feature for advanced sql usage, September 2017. `https://blog.jooq.org/2017/09/01/join-elimination-an-essential-optimiser-feature-for-advanced-sql-usage/`.

[15] N. Niclausse. Tsung, 06 2017. `http://tsung.erlang-projects.org/`.

[16] Oracle. Mysql slap. `https://dev.mysql.com/doc/refman/8.0/en/mysqlslap.html`.

[17] Oracle. Sakila structure. `https://dev.mysql.com/doc/employee/en/sakila-structure.html`.

[18] Mike Owens. *The definitive guide to SQLite*. Apress, 2006.

[19] Spring. Spring-boot, June 2020. `https://spring.io/projects/spring-boot`.

[20] Wumpz. Jsqlparser. `https://github.com/JSQLParser/JSqlParser`.