unibz

Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Fakultät für Informatik
Facoltà di Scienze e Tecnologie informatiche
Faculty of Computer Science

Bachelor in Computer Science

Bachelor Thesis

Extending Esfinge Query Builder framework to support Apache Cassandra

Candidate: Samuel Dalvai

Supervisor: Professor Eduardo Martins Guerra

September 2022

Abstract

NoSQL database technologies are becoming increasingly popular and are sometimes preferred over classical SQL-based relational databases, especially in Web applications. The lack of a unified model among the various storage systems, however, introduces new challenges to the configuration and use of such technologies. Creating an application that is able to migrate from one database technology to another is a non-trivial task, especially if the transition is from a relational to a NoSQL database. To this end, it is necessary to create a mechanism to abstract communication with the database and, more importantly, to manipulate the data regardless of how it is modeled. In this thesis we propose a solution for creating a persistence layer which can be easily configured to migrate from a relational database to Apache Cassandra, a widely used NoSQL database technology. The proposed solution extends the Esfinge QueryBuilder framework with a module for Cassandra, enabling queries capable of expressing relational constructs that are not usually available to Cassandra. The solution was evaluated by integrating it into an application and measuring the effort of migrating from a relational database to Cassandra. In conclusion, this thesis demonstrates that with the proposed framework approach, it is possible to create a layer for polyglot persistence and make the implementation details of a NoSQL database invisible to the application.

Riassunto

Le tecnologie di database NoSQL stanno diventando sempre più popolari e talvolta sono preferite ai classici database relazionali basati su SQL, soprattutto nelle applicazioni Web. La mancanza di un modello unificato tra i vari sistemi di archiviazione, tuttavia, introduce nuove sfide alla configurazione e all'utilizzo di tali tecnologie. Creare un'applicazione in grado di migrare da una tecnologia di database a un'altra è un compito non banale, soprattutto se la transizione avviene da un database relazionale a uno NoSQL. A tal fine, è necessario creare un meccanismo per astrarre la comunicazione con il database e, soprattutto, per manipolare i dati indipendentemente da come sono modellati. In questa tesi proponiamo una soluzione per creare un livello di persistenza che può essere facilmente configurato per migrare da un database relazionale ad Apache Cassandra, una tecnologia di database NoSQL ampiamente utilizzata. La soluzione proposta estende il framework Esfinge QueryBuilder con un modulo per Cassandra, consentendo di effettuare query in grado di esprimere costrutti relazionali che solitamente non sono disponibili per Cassandra. La soluzione è stata valutata integrandola in un'applicazione e misurando lo sforzo di migrazione da un database relazionale a Cassandra. In conclusione, questa tesi dimostra che con l'approccio a framework proposto è possibile creare un strato di persistenza poliglotta e rendere i dettagli di implementazione di un database NoSQL invisibili all'applicazione.

Zusammenfassung

NoSQL-Datenbanktechnologien erfreuen sich zunehmender Beliebtheit und werden manchmal den klassischen SQL-basierten relationalen Datenbanken vorgezogen, insbesondere bei Webanwendungen. Das Fehlen eines einheitlichen Modells für die verschiedenen Speichersysteme führt jedoch zu neuen Herausforderungen bei der Konfiguration und Nutzung solcher Technologien. Die Erstellung einer Anwendung, die in der Lage ist, von einer Datenbanktechnologie zu einer anderen zu migrieren, ist keine leichte Aufgabe, insbesondere wenn der Übergang von einer relationalen zu einer NoSQL-Datenbank erfolgt. Zu diesem Zweck ist es notwendig, einen Mechanismus zu schaffen, der die Kommunikation mit der Datenbank abstrahiert und, was noch wichtiger ist, die Daten unabhängig von ihrer Modellierung manipulieren kann. In dieser Arbeit schlagen wir eine Lösung zur Erstellung einer Persistenzschicht vor, die einfach konfiguriert werden kann, um von einer relationalen Datenbank zu Apache Cassandra, einer weit verbreiteten NoSQL-Datenbanktechnologie, zu migrieren. Die vorgeschlagene Lösung erweitert das Esfinge QueryBuilder Framework um ein Modul für Cassandra, das Abfragen ermöglicht, die in der Lage sind, relationale Konstrukte auszudrücken, die für Cassandra normalerweise nicht verfügbar sind. Die Lösung wurde evaluiert, indem sie in eine Anwendung integriert und der Aufwand für die Migration von einer relationalen Datenbank zu Cassandra gemessen wurde. Abschließend zeigt diese Arbeit, dass es mit dem vorgeschlagenen Framework-Ansatz möglich ist, eine Schicht für polyglotte Persistenz zu schaffen und die Implementierungsdetails einer NoSQL-Datenbank für die Anwendung unsichtbar zu machen.

Acknowledgements

I would like to thank Professor Eduardo Martins Guerra for his advice and continued support during the writing of this thesis. My heartfelt thanks also goes to Giulia, who has been by my side supporting me every moment of this course of study, and to my family who has been able to understand the sacrifices I had to make to pursue my goals. Finally, I would like to thank all the professors at this university, who have been able to inspire me, and all the classmates who have helped me face the most difficult challenges.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Motivation

Relational databases are the most widely used type of storage system in modern applications, but recent years have seen a growing popularity of alternative storage technologies, commonly referred to as NoSQL databases. NoSQL databases have functionalities and features that in some cases are more in line with the application domain requirements than those offered by classic SQL databases[1]. A single application might even use different types of databases to take advantage of the specific benefits of various storage technologies, but the development of such a system would require considerable effort in terms of configuration and maintenance[2]. To develop such a flexible application, having tools and technologies that enable seamless switching from one storage technology to another can be considered a valuable resource. However, creating such tools can be difficult, because each type of database technology has special characteristics that must be taken into account. These differences are even more apparent when comparing the data models of relational database systems with those of NoSQL systems. For these reasons, an abstraction is needed that allows our application to communicate with a variety of database technologies and at the same time can handle the different types of data models as if they were unified with common characteristics[3].

1.2 Objective

The goal of this thesis is to reduce the effort of migrating an application's storage technology from a SQL-based relational database to Apache Cassandra[4]. Performing this type of migration means having to reconfigure the system to support a substantially different type of storage technology. In addition to configuration issues, it is necessary to take into account the fact that Cassandra does not support all the relational constructs and query capabilities available in SQL-based storage systems. For example, in classic relational databases it is possible to perform a join between two different tables, whereas in Cassandra this is not possible and the only way to achieve this is to create additional tables representing the join or to add client-side logic. Also, in Cassandra the order of query results is fixed and can be specified only for some special types of columns, whereas in relational databases the results can be ordered more freely. With this thesis, we not only want to develop a solution that reduces the configuration problems of such a migration, but also one that can express database queries that may not be supported by the Cassandra data model. For example, we

want to be able to perform queries with sorting on arbitrary columns of our tables, instead of being limited to specific ones.

1.3 Approach

To achieve the objective described in section 1.2, we started with an existing Java framework for creating a persistence layer and extended it to support Cassandra. The framework used for this thesis, Esfinge QueryBuilder[5], has a feature that allows it to interpret method names and translate them into queries. This functionality helped us develop an extension which is able to express relational constructs familiar to an SQL based system. After developing the extension, we tested it in an application and evaluated the ease of migration from an SQL-based relational database to Cassandra in quantitative and qualitative terms.

1.4 Structure of the thesis

This thesis is divided into six chapters. Chapter 1, which is the introductory chapter, describes the motivation and goal of this thesis, as well as describing the approach used. Chapter 2 provides an overview of the field of study and the theory behind this thesis, which can help the reader better understand the domain to which it relates. Chapter 3 describes the functionality and internal structure of the Esfinge QueryBuilder framework, which is used as the basis for the development of the solution that satisfies the objective of this thesis. Chapter 4 describes the particular challenges of the problem this thesis seeks to solve, as well as providing some details on the development approach and implementation results. Chapter 5 presents the experiment used to evaluate the results of the implementation and discusses the degree to which the goal was achieved. Finally, chapter 6 summarizes the work done and proposes a number of possibilities for future work on this specific problem.

Chapter 2

Background

Before describing the internal structure and functionality of the Esfinge Querybuilder[5] framework, it is necessary to provide an overview of the main concepts to better understand the macro area in which the software can operate. This chapter therefore aims to introduce these concepts.

2.1 Layered Architecture

Software architecture is important in the implementation of non-functional requirements like reliability, scalability and robustness. In addition, following a particular architectural pattern helps to structure an application in a way that enables the creation of software that is more maintainable. There are many different patterns that are common in the field of software development. For example, the Micro-services pattern creates units of software that are isolated between each other and can be deployed as a standalone modules. However, one of the architectural patterns that can be find more often in applications is the layered architecture. As the name implies, this particular software structure relies on the concept of layering, in which the software modules are divided based on specific roles.

Depending on the size of the application, usually between three and four different types of levels can be found[6].

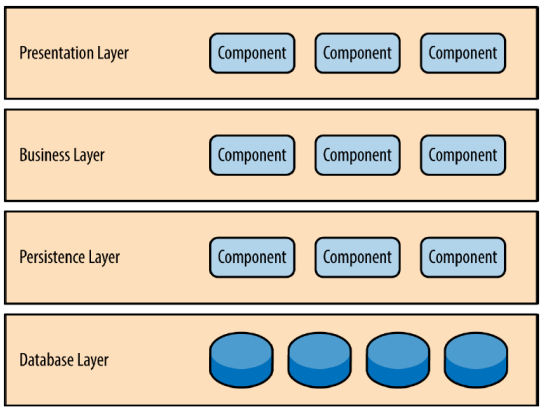

Figure 2.1: Layers in layered architecture

Figure 2.1 shows the four typical layers in a layered architecture, namely presentation,

business, persistence and database layer[6]. In the 3-layered alternative, the persistence and the business layer are usually merged together. As mentioned above, each layer has a specific role, decoupled from the others, for example, the presentation layer manages the user interface and presents data. The business layer on the other hand has the role of encapsulating the business logic of the application. The persistence layer acts as a link between the business layer and the database layer, in an object-oriented application it can for example translate the results of an SQL query into an object and pass it to the business layer. Finally the database layer is the part of the application in which the data is actually stored. Each layer in this architecture is defined as closed[6], which means that every communication has to go through each level in order to reach the not adjacent ones.

One of the advantages of this architecture is that by relegating specific implementation responsibilities to a particular level, the principle of separation of concerns can be applied, whereby each level can view the others as abstract and should not be interested in knowing its internal structure. By applying this model, the software also becomes more maintainable, as each change in one layer occurs in isolation and does not directly affect the other layers. Another advantage is that this type of architecture has become an industry standard for business applications, for this reason many companies have adopted it and there are many frameworks that support its development. Finally, one benefit of the layered architecture is that it lends itself to testing, because its structure favors the possibility to test each layer in isolation while emulating the others[6].

2.2 Persistence Layer

As described in section 2.1, the persistence layer in a layered architecture is that part of the application which acts as an intermediary between the database and the business layer. There are a variety of ways to create a persistence layer, in particular several types of patterns can be used; the next sections briefly describe the characteristics of the main ones.

2.2.1 Data Access Object pattern (DAO)

A solution that provides a high degree of abstraction for a persistence layer is the Data Access Object pattern, or in short "DAO". This particular pattern separates the application functionality that accesses the database from the one that manages the data. The DAO is responsible for reading and writing from and to the database and provides data encapsulation for the business layer[7]. By applying this pattern, the application domain model can be decoupled from the underlying database technology, thus making the application database-agnostic.

```
1 public interface DAO<E> {
2     void save(E e);
3
4     E get(Long id);
5
6     void update(E e);
7
8     void delete(E e);
9 }
```

Listing 2.1: DAO interface

Code fragment 2.1 shows an example of how the DAO pattern can be implemented in Java[8], using an interface to abstract the operations of an arbitrary object and defining the four main CRUD operations, namely create, read, update and delete.

```
public class PersonDAO implements DAO<Person> {
    @Override
    public void save(Person p) {
        // Implementation omitted ...
    };

    @Override
    public Person get(Long id) {
        // Implementation omitted ...
    };

    // Other methods omitted ...
}
```

Listing 2.2: DAO concrete implementation

Code fragment 2.2 shows an example of how the `DAO` interface shown in 2.1 can be implemented for a `Person` class. The implementation of the methods can be done in various ways, for example by using Object relational mapping, which is described in subsection 2.2.4. The DAO pattern hides from the business layer the actual implementation of operations that are performed on the database. As a result, changes can be applied to the business layer without affecting the persistence layer and vice versa.

2.2.2 Repository pattern and Domain-Driven Design

A somewhat similar concept to the Data access object is the Repository, which is a pattern typically used in domain-driven design. It is an abstraction that mimics the concept of a collection, in which clients perform requests based on some criteria and the Repository retrieves the data and responds with the object(s).

What primarily differentiates a DAO from a Repository is that the latter abstracts a collection of objects rather than just persistence. In addition, a Repository has a higher level of abstraction and, compared to a DAO, is closer to the business level.

```
public interface Repository<E> {
    E get(Long id);

    void add(E e);

    void update(E e);

    void delete(E e);
}
```

Listing 2.3: Repository interface

Code fragment 2.3 shows an example definition of a Repository interface. Like DAOs, Repositories can handle more complex requests than simple CRUD operations, such as performing calculations or returning the result of a count[9]. The benefits of Repositories are

similar to those offered by DAOs: both provide a mechanism for decoupling the business layer from the persistence layer.

2.2.3 Query-based persistence layer

Maybe the most straightforward way of implementing a persistence layer can be the one based on queries. In this pattern, the interface between the business layer and the database is created, providing a set of methods that execute queries by accepting certain parameters.

```
1  public void deletePerson(int id) throws SQLException {
2      String sql = "DELETE FROM person where id = ?";
3
4      connect();
5      PreparedStatement statement = conn.prepareStatement(sql);
6      statement.setInt(1, id);
7
8      statement.executeUpdate();
9      statement.close();
10     disconnect();
11 }
```

Listing 2.4: Example of query based method

The implementation of a persistence layer by using a query based approach is simple, but the level of coupling between the business layer and the database is too high. For example, code fragment 2.4 shows a method that specifically handles Person entities, and both the query language and connection handling refer to a specific type of relational database.

The problem with the query-based approach is that if the storage system has to be changed or if the application has to manage different entities, the persistence layer has to be changed along with the business layer. In particular, each query has to be modified according to the database query dialect and the name and characteristics of the entities, which can result in a large overhead in maintenance, depending on the number and complexity of the implemented methods.

2.2.4 Object relational mapping (ORM)

Sections 2.2.1 and 2.2.2 showed how to abstract the retrieval and storage of objects between the business layer and the persistence layer, however, a mechanism is still needed to map objects to database entries and vice versa. Object-oriented programming languages (OOP) and the languages used by relational databases are two distinct paradigms, which are based on different constructs. For example, in relational databases it is possible to find the concept of primary key, which is not found in OOP. At the same time, there are abstractions that can be found in OOP that are not found in relational databases, for example, inheritance. The activity of trying to bridge the gap between these two paradigms is called Object-Relational Mapping (ORM), and is typically done by using two patterns, namely the Active Record and the Data Mapper pattern. The difference between the two philosophies is that with the Data Mapper pattern the domain of the application is less tied to the Database, whereas the Active Record pattern makes the persistence layer less visible and the link between the business layer and the database stronger. One example of an ORM for the Java programming language following the Data Mapper pattern is the Java Persistence API (JPA), of which Hibernate[10] is one of the most widespread implementation.

```java
@Entity
@Table(name="PERSON")
public class Person {
    @Id
    @Column(name="PERSON_ID")
    private long id;
    private String name;

    // Rest of the class omitted ...
}
```

Listing 2.5: Example of Java annotations for JPA

Code fragment 2.5 shows an example of usage of JPA with Java annotations[11]. By using the Java annotations `@Table` and `@Column`, it is possible to map the object to the table `PERSON` and the `id` field to the column `PERSON_ID`.

```java
public class PersonDAO implements DAO<Person> {
    protected EntityManager entityManager;

    @Override
    public void save(Person p) {
        entityManager.persist(p);
    };

    @Override
    public Person get(Long id) {
        entityManager.find(Person.class, id);
    };

    @Override
    public void update(Person p); {
        entityManager.merge(p);
    };

    // Other methods omitted ...
}
```

Listing 2.6: DAO concrete implementation with JPA

Code fragment 2.6 shows an example of implementation of a DAO interface that uses the `EntityManager` class, which is part of the Hibernate framework and performs the mapping of objects for the persistence layer.

2.2.5 Internal domain-specific languages

Internal domain-specific languages are another important technique for creating a persistence layer. First of all, it is necessary to distinguish between general-purpose languages and domain-specific languages, or DSL for short. The former are Turing complete programming languages, which means that they can be used to solve any computational problem, the latter are programming languages that are specific to a domain, and for this reason can usually solve a restricted set of problems[12]. One example of a general purpose language is Java, while SQL is a typical example of domain-specific language. In addition it is possible to

use a general purpose language to create what is called an internal DSL, which is sometimes referred to as an API. For the creation of an internal DSL, the constructs of a general-purpose language are used to define a framework to help solve a domain-specific problem.

```
1  public class Query {
2      private StringBuilder query;
3
4      public Query() {
5          query = new StringBuilder();
6      }
7
8      public Query select(String entity) {
9          query.append("SELECT from " + entity);
10         return this;
11     }
12
13     public Query where(String clause) {
14         if (!query.toString().contains("WHERE"))
15             query.append(" WHERE " + clause);
16         else
17             query.append(" " + clause);
18
19         return this;
20     }
21
22     public Query and() {
23         query.append(" AND");
24         return this;
25     }
26
27     public String sql() {
28         return query.toString() + ";";
29     }
30 }
```

Listing 2.7: Method chaining implementation

Code fragment 2.7 shows a simple example of an API that uses the Java programming language to query a database by exploiting the concept of method chaining. Note the use of the "`this`" keyword at the end of the methods, which allows several method calls to be concatenated one after the other. In addition, the `StringBuilder` class is used to add the new elements to the query; in the end, the `sql()` method returns the resulting `String`.

```
1  public class Main {
2      public static void main(String[] args) {
3          String query = new Query().select("Person")
4                  .where(" name = 'Paul'")
5                  .and()
6                  .where(" age = 30")
7                  .sql();
8      }
9  }
```

Listing 2.8: Method chaining usage

Code fragment 2.8 shows how the methods of class `Query` can be concatenated one after the other, after first calling the constructor. The string resulting from the method concatenation is `"SELECT from Person WHERE name = 'Paul' AND age = 30;"`. What was shown in code fragment 2.7 is the creation of a new language on top of Java. This new language is what is called an internal domain-specific language and in this case it helps solve a specific problem, which is the creation of SQL queries. Note that the implementation shown in the code fragment 2.7 lacks a mechanism for forcing the methods to be called in a specific order, but for simplicity it has been omitted.

A real-world example of a domain-specific internal language is found in the Spring Data JPA[13] framework. With this particular framework, database queries can be created based on method names. Spring Data has similar functionality to Esfinge QueryBuilder[5], which is the framework at the heart of the field of study of this thesis and will be described in the chapter 3.

```
1 interface PersonRepository extends Repository<Person, Long> {
2   List<Person> findByLastname(String lastname);
3 }
```

Listing 2.9: Spring Data JPA

Code fragment 2.9 shows an example usage of the Spring Data framework; the method `findByLastname()` is translated into a query by the framework, and the naming convention follows some rules that define what can be considered a domain-specific internal language.

2.3 NoSQL Databases

So far, only persistence layers that interface with relational databases through the SQL language have been considered, but relational databases are not the only way to store data in a persistent state. An alternative to the well established relational databases are NoSQL databases, which is a term that means "not only SQL". Relational database systems, in short RDBMS, offer many useful properties, for example consistency and reliability in the data, they follow the ACID properties (Atomicity, Consistency, Isolation, Durability) and are meant for software systems in which many concurrent users have to access a single database.

There are, however, cases in which RDBMS are not the best solution, for example when data has to be distributed or when the database has to scale out, which means adding more servers to increase the capacity of the system. In these cases, NoSQL databases can be preferred over conventional ones. The main characteristic of NoSQL databases is that they are usually schema-less, which means that they don't follow a rigid schema like RDBMSs, and are for this reason more flexible in terms of the constraints imposed to the data that is being stored[14]. NoSQL databases are usually categorized into four main categories:

- Key-Value databases
- Document-Oriented databases
- Column-Family databases
- Graph databases

Key-Value NoSQL databases store a collection of key-value pairs and offer the advantage of speed and scalability[3]. Examples of Key-Value databases are Redis[15] and Amazon Dynamo DB[16].

Document-Oriented databases are based on the Key-Value databases, but as a value they store a file, which could be for example a `JSON` or an `XML`. Examples of Document-Oriented databases are MongoDB[17] and Apache Couch DB[18].

In Column-Family databases data is stored by column instead of by row. A characteristic of these types of databases is that columns having the same name are grouped in clusters and are able to store data of different types, the advantage with this technique is that queries do not access data which is not needed, thus making operations on large amounts of data more efficient. An examples of a Column-Family database is Apache Cassandra[4].

Finally, Graph databases are storage systems in which elements are represented as nodes arranged in a graph structure, and each node has a pointer to the next element. These characteristics make this type of databases ACID compliant and are useful when dealing with data which has a high number of interconnections. An example of Graph databases is Neo4j[19].

2.4 Cassandra

Since the focus of this thesis is the extension of the Esfinge QueryBuilder framework to support Apache Cassandra, this section introduces this storage technology with some additional details. As described in the previous section, Cassandra is a Column-Family NoSQL database, open source, and developed and maintained by the Apache Foundation. It is a distributed database, which means that data is not usually stored in a single node or machine, but rather in a relatively high number of nodes. These features of Cassandra make the storage system faster and more resistant to failure. Another important characteristic of Cassandra is that it does not fully support the relational model of classic RDBMS systems. In fact, the query language used by Cassandra, called CQL, is mostly similar to SQL, but lacks some of its most common features, such as joins[1]. Data in Cassandra is modeled by using the following core elements[4]:

- **Cassandra Column**: a column is the most basic element in the Cassandra data moder, and represents a name/value pair.

- **Cassandra Row**: a row is a collection of columns with different names, similar to the concept of data records in relational databases.

- **Keyspaces**: keyspaces are the most important element of Cassandra and can be compared to the concept of database schema of a relational database. They have two important attributes, the first of which is the replica placement strategy, which defines how replicated nodes are distributed in the system and can follow either a "simple strategy" or a "network topology strategy", the latter of which also takes into account how the network of nodes is structured. The second attribute is the replication factor, which defines how many times data has to be replicated among different nodes[4].

[1] Section 4.1 describes the main differences between Cassandra and relational databases in more detail

- **Column Families**: rows in Cassandra are organized into column families, which are somewhat similar to the concept of tables in RDBMSs, but unlike tables, columns in Cassandra can be added to individual rows without doing the same to all rows.

```
CREATE KEYSPACE test WITH replication = {'class': 'SimpleStrategy','
    replication_factor': 3};
```

Listing 2.10: Keyspace creation

Code fragment 2.10 shows the creation of a keyspace using the CQL syntax; in this example, a Keyspace is created by defining the replica placement strategy and the replication factor.

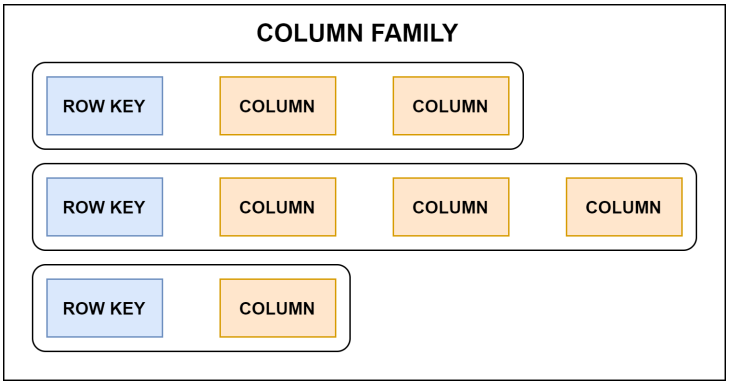

Figure 2.2: Column family with different number of columns

Figure 2.2 shows a column family with three rows, where each row has a different number of columns, as explained previously, column families in Cassandra can have rows with different number of columns.

The way in which Cassandra is structured allows the storage system to be highly scalable and provides a high degree of availability. Because of the replication factor between nodes, when a failure occurs the other nodes in the topology can intervene and take care of the request.

2.5 Polyglot Persistence

When a layered architecture is to be created, it is preferable to make each layer as abstract as possible, so that when changes need to be applied to it, the other layers can continue to function without the need to modify the code. In addition, when a persistence layer is built that integrates a business layer with a database, the ability to be able to change the technology of the storage system without having to rethink the entire layers offers a major advantage. This, however, can become particularly complicated when moving from a classical relational database to a NoSQL database, as the two technologies can differ substantially in the way they view data storage and retrieval.

In order to build an application that has the ability to switch "almost" seamlessly from one database technology to another, it is necessary to exploit the concept of polyglot persistence. The need for polyglot persistence comes from the fact that software applications for different domains might have different requirements in terms of the storage systems.

For example a financial-related software might need the properties of relational databases, because transactions should always be reliable. The shopping cart of an E-commerce on the other hand, might prefer using a key/value database because of the heterogeneity of the products that it needs to store.

In addition, by comparing SQL databases with NoSQL databases, it is also possible to find substantial differences in the way data is structured and queried, e.g., in some storage systems the concept of joins does not exist, also two systems might use a completely different query language. The three main approaches used to implement polyglot persistence are the following[20]:

- **Domain-Oriented**: this approach, which is currently the most popular, seeks to define the characteristics of the system and choose the most appropriate data storage technology accordingly[20].
- **Querylanguage-Based**: this approach is based on translating one query language into another.
- **Framework/Middleware**: in the Framework/Middleware approach an intermediate layer between the persistence layer and the database serves as an additional abstraction. Examples of Framework/Middleware are Apache Drill[2] and the Polyglot Persistence Mediator[2], the latter is a REST-based system that acts as a mediator between the application and the back end. The Esfinge QueryBuilder framework, which is the subject matter of study of this thesis is a further example of Framework/Middleware for polyglot persistence.

The concept of polyglot persistence can also be useful when dealing with large-scale applications, which can have a variety of requirements based on their functionality. Depending on the use case, a different type of database, relational or NoSQL, might be more appropriate; therefore, using different types of storage systems in the same application might offer some advantages[21].

Whether an application needs to integrate a different storage technology or a larger application needs to manage different types of databases simultaneously, the presence of an abstract persistence layer that takes advantage of the concept of polyglot persistence will make the configuration and migration process easier. However, research on polyglot persistence still needs to solve some issues before it is effectively implemented, for example, the problem of authentication in NoSQL storage systems[20].

[2]Apache Foundation, Apache Drill, `https://drill.apache.org/`

Chapter 3

Esfinge QueryBuilder

3.1 General View

Esfinge QueryBuilder[5] is an open source Java framework that allows the creation of a persistence layer in a simple way. The framework translates method names at runtime and generates queries that can be executed on the database. The main module of the framework is `querybuilder-core`, which contains the logic for the method parsing, in addition there is a total of four modules for the integration to four different storage system technologies, namely:

- QueryBuilder-JPA1
- QueryBuilder-JDBC
- QueryBuilder-MongoDB
- QueryBuilder-Neo4J

These modules use the results of the method parsing made by the `querybuilder-core` module and perform the actual translation to the specific query language of the database of interest. In order to use the framework, the application needs to extend the `Repository` interface for the specific entity.

```
1 public interface PersonDAO extends Repository<Person>{
2     public List<Person> getPersonByName(String name);
3     public List<Person> getPersonByAgeGreater(Integer age);
4 }
```

Listing 3.1: Repository example

Code fragment 3.1 shows an example of how the `Repository` interface can be extended for a class `Person`. In addition, the extended `Repository` interface must be instantiated by the framework in order to be used.

```
1 PersonDAO dao = QueryBuilder.create(PersonDAO.class);
2
3 List<Person> list = dao.getPersonByName("Paul");
```

Listing 3.2: Repository instantiation and utilization

Code fragment 3.2 shows an example of instantiation and usage of the `Repository` interface extension. The framework will translate the method at line 3 into a query, which will be executed by automatically handling the database connection and the results will be returned as a `List`.

```
1    select person.id, person.name where name = 'Paul';
```

Listing 3.3: Translated method

Code fragment 3.3 shows an example of a query generated for the method shown in example 3.2 by using the `QueryBuilder-JDBC` extension. The example assumes that the `Person` class only has the attributes "id" and "name".

3.2 Features

3.2.1 Method interpretation

As described in the previous section, the main functionality of the framework is the interpretation of methods for the run-time generation of queries. In order for the translation to be successful, the following rules have to be observed:

- Methods must start with the keyword `get` and be followed by the name of the entity, for example `getPerson()`.
- In order to add parameters to the query, the keyword `By` followed by the parameter name must be used. Also, the parameter of the correct type must be added to the method, for example: `getPersonByName(String name)`.
- The return type of the method can be either a `List` of entities or a single entity.
- If an entity has an attribute which is a custom type, the framework will also manage joining the different entities of the database. For example, if the `Person` entity has an `Address` attribute with fields `id` and city, the method `getPersonByAddressCity()` can be used to retrieve a person based on the city attribute.
- In order to query the entities by more than one parameter it is necessary to add the keyword `And` or `Or` between the parameter names. An example method with two parameters might be for example `getPersonByNameAndAge(String name, Integer age)`.

```
1 // Invalid method definitions
2 public interface PersonDAO extends Repository<Person>{
3     List<Person> getPersonByNameOrLastName(String name)
4     List<Person> getPersonByNameAge(String name, Integer Age)
5     List<Person> setPersonById(Integer id)
6 }
```

Listing 3.4: Examples of invalid method names

Code fragment 3.4 depicts an example of a Repository with invalid method names. The first method is invalid because it lacks `lastName` as a parameter, the second method lacks the connector between `Name` and `Age`, and finally the third method does not start with the keyword `get` as it should.

3.2.2 Basic CRUD operations

In addition, the Repository interface also supports the following CRUD operations by default:

- `E save(E obj)`: the object is saved or updated if it already exists.
- `void delete(Object id)`: the object with the corresponding id is deleted from the database.
- `List list()`: returns a List of all the entities in the database.
- `E getById(Object id)`: returns the object with the corresponding id.
- `List queryByExample(E obj)`: returns the object based on the attributes of the object passed as parameter. The attributes that are not instantiated are ignored by the query.

3.2.3 Types of comparisons

The framework supports all the standard comparison types for queries, namely: `Lesser`, `Greater`, `LesserOrEquals`, `GreaterOrEquals`, `NotEquals`, `Contains`, `Starts` and `Ends`. In order to produce a query with a particular type of comparison, the framework has two options. The first is to add the name of the comparison after the parameter name in the body of the method, and the second is to use the annotations provided by the framework.

```
1 List<Person> getPersonByAgeGreater(Integer Age)
2 List<Person> getPersonByAge(@Greater Integer Age)
```

Listing 3.5: Method name with comparison

Code fragment 3.5 shows two methods that from the perspective of the framework are equivalent and produce the same query.

3.2.4 Sorting

The framework also supports sorting with the keyword `OrderBy` followed by one or more parameters for the sorting criteria. If more than one parameter is provided for the sorting criteria, the `And` connector must be used; in addition, the sorting direction can be defined as descending using the `Desc` keyword or ascending using the `Asc` keyword or by omitting it.

```
1 List<Person> getPersonByAgeOrderByName(Integer Age)
2 List<Person> getPersonByAgeOrderByNameAndAge(Integer Age)
3 List<Person> getPersonByAgeOrderByNameDesc(Integer Age)
```

Listing 3.6: Method name with sorting

Code fragment 3.6 shows three examples of method declaration with ordering.

3.2.5 Query Objects

Another feature supported by the framework is the use of query objects as method parameters. We can use query objects to shorten the method declaration when there are too many parameters. To use a `QueryObject` as a parameter, it is necessary to declare it by following some naming conventions for attributes. First, the attributes of the query object have to be named after the attributes of the entity to be queried. Also, the name of the comparison must be added after the attribute names or, alternatively, the specific annotation can be placed above the attribute, similar to what is shown in 3.2.3.

```
1  public class PersonQueryObject {
2      private Integer ageLesser;
3
4      @Starts
5      private String name;
6
7      // ... getters and setters omitted
8  }
9
10 public interface PersonDAO extends Repository<Person>{
11     // Method with QueryObject
12     public List<Person> getPerson(@QueryObject PersonQueryObject obj);
13
14     // Method without standard declaration
15     public List<Person> getPersonByAgeLesserAndNameStarts(Integer age, String
           name);
16 }
```

Listing 3.7: QueryObject

Code fragment 3.7 shows an example of a `QueryObject` and a method that uses it as parameter. The methods at lines 12 and 15 are equivalent and will produce the same query. Please note that the `@QueryObject` annotation is required in front of the `getPerson()` method parameter.

3.2.6 Domain terms

Domain terms are a special feature of the framework that allows the creation of unique keywords that can be associated with complex rules; their definition is done by annotating the class that extends the `Repository` interface.

```
1 @DomainTerms({
2 @DomainTerm(term="young", conditions=@Condition(property="age",comparison=
      ComparisonType.LESSER,value="20")),
3 @DomainTerm(term="italian citizen", conditions=@Condition(property="address.
      country",value="IT"))
4 })
5 public interface PersonDAO extends Repository<Person>{
6     public List<Person> getPersonYoung();
7     public List<Person> getPersonYoungItalianCitizen();
8 }
```

Listing 3.8: Domain terms

Code fragment 3.8 shows an example of a `Respository` interface with the declaration of domain terms. The rule is to create the main `@DomainTerms` annotation, within which one or more `@DomainTerm` annotations can be inserted. Each `@DomainTerm` annotation can in turn contain one or more `@Condition` annotations that define the conditions associated with the defined keywords. After a domain term is defined, the keyword can be used in the method name, and several domain terms can also be combined together. For example, calling the `getPersonJoung()` method of code fragment 3.8 automatically applies the condition that the `age` attribute must be less than 20.

3.2.7 Queries with null parameters

Finally, the framework provides the ability to specify query behavior when dealing with null parameters; the following two annotations can be used to annotate method parameters for this purpose:

- `@IgnoreWhenNull`: the parameter of the method will be ignored if it is null.
- `@CompareToNull`: if the parameter of the method is null it will be compared with null values found in the database for that specific parameter.

3.3 Internal Structure

The following is an overview of internal structure of the `querybuilder-core` module of the framework. All the packages that are described next are contained in the core package of the framework `net.sf.esfinge.querybuilder`:

- package `annotation`: contains all the annotations of the framework, for example the ones related to comparison types 3.2.3 or the ones related to domain terms 3.2.6. Examples of these are the `@GreaterOrEquals` and the `@DomainTerm` annotations.
- package `exception`: contains the custom exceptions of the framework.
- package `executor`: contains the `QueryExecutor` interface, whose concrete implementation is called by the framework to execute the interpreted query on the database and return the results.
- package `methodparser`: this package contains classes and interfaces that enable the parsing of methods names. The `MethodParser` interface, which is implemented by the abstract class `BasicMethodParser`, is one of the most important elements in this package. `DSLMethodParser` and `QueryObjectMethodParser` are the two classes that provide concrete implementations of the `parse()` method.

  Figure 3.1 depicts the inheritance hierarchy for the `MethodParser` interface. The `parse()` method is particularly important because, as the name suggests, it is the one that actually performs the parsing and creates a `QueryInfo` object. The `QueryInfo` class is a special object that encapsulates query information, such as the entity name, query conditions, and other information that can describe the query in an abstract way.

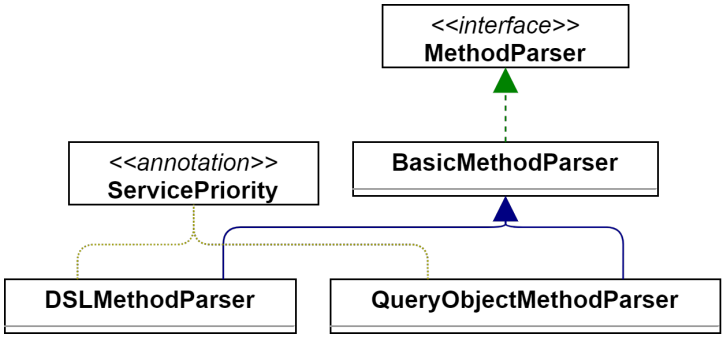

Figure 3.1: MethodParser interface and implementations

- `package utils`: this package contains several utility classes useful to the parsing process, for example the class `ReflectionUtils` contains useful methods that use reflection to infer method information, for example the existence of a given annotation.
- `package querybuilder (core package)`: this package contains two important elements, the first one is the `Repository` interface, which has to be extended when using the framework in order to add our custom methods, as explained in section 3.1. The second important element is the `QueryBuilder` class. This class is responsible for configuring the concrete implementations of the main interfaces of the framework, for example `MethodParser` and `QueryExecutor`, and invoking the `executeQuery()` method of the `QueryExecutor` interface.

3.4 Extension Points

As described in section 3.1, the Esfinge QueryBuilder framework has currently a total of four modules that allow the integration with different database technologies, for example JPA[11] and MongoDB[17]. Each one of these modules adds specific behaviour to the querybuilder-core module by providing concrete implementations of the five main extension points of the framework, which are listed and described here:

1. `EntityClassProvider`: this interface is used to map the entities used by the framework into the class. Depending on the storage system, the entities classes may be mapped in different ways, for example through the definition in an XML file or by adding a particular annotation to the class. The method of this interface unifies these methods in a general one.
2. `QueryVisitor`: this interface has various methods that are invoked by the framework when the method name is parsed. The purpose of this interface is to build the query step by step according to the rules of the specific storage system.
3. `QueryRepresentation`: this interface is created starting from the QueryVisitor interface with the method `QueryRepresentation getQueryRepresentation()` and adds an additional layer of information to the query, for example by applying parameters dynamically.
4. `QueryExecutor`: this interface is the one that actually executes the query on the specific database and returns the results. The concrete implementation has to handle the

database connection and call the `visit()` method of the `QueryInfo` class to produce a query.

```
1 QueryVisitor visitor = JPAVisitorFactory.createQueryVisitor();
2 info.visit(visitor);
3 qr = visitor.getQueryRepresentation();
```

Listing 3.9: JPA executeQuery method implementation

Code fragment 3.9 shows some of the implementation details of the `executeQuery()` method for QueryBuilder-JPA.

5. `Repository`: this interface needs to be implemented with the basic CRUD operations for the specific storage system technology. Also in this case the connection to the database needs to be managed autonomously.

In addition, the class of the concrete implementation of the interfaces `QueryExecutor`, `EntityClassProvider` and `Repository` must be explicitly declared as services. In order to do this, the `src/main/resources/META-INF/services` folder needs to contain the following three files:

- net.sf.esfinge.querybuilder.executor.QueryExecutor
- net.sf.esfinge.querybuilder.methodparser.EntityClassProvider
- net.sf.esfinge.querybuilder.Repository

Each one of these files has to contain the path to its concrete implementation. Figure 3.2 shows the default folder structure of an extension for the querybuilder-core module.

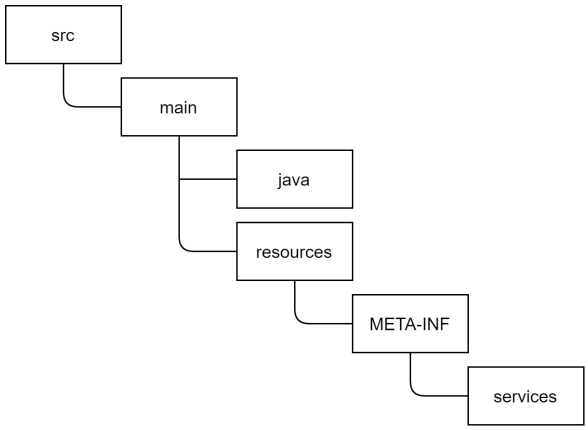

Figure 3.2: Querybuilder extension folder structure

When the service files are declared, the concrete implementation of the three interfaces become available to the framework.

3.4.1 Framework test suite

Each of the framework extensions[22] also includes a set of unit and integration tests that help verify whether the implementation of the extension points are correct. In addition,

these tests are equivalent across extensions because they test the same behavior, only the expected results are different as they change depending on the database technology. The test suite consists in a little more than 30 unit tests and an average of 45 integration tests, depending on the complexity of the framework extension. In addition, the test classes use a naming convention that depends on the name of the storage technology implemented by the framework extension, for example, the test class for the `QueryVisitor` implementation of QueryBuilder-JDBC is called `TestJdbcQueryVistor`. Some features of the framework extensions are tested with unit tests, while others must be verified by running integration tests with a real database instance.

Unit tests

The categories of functionalities that are verified by the unit tests are QueryVisitor, DynamicQueries and EntityClassProvider. Dynamic queries are queries that are allowed to have null parameters. The tests in the DynamicQueries category verify the functionality related to the `QueryRepresentation` interface, which handles this type of query.

Integration tests

The functionalities verified by the integration tests are Repository, QueryBuilder, NullValues, QueryObject, DomainTerms, Custom Methods and Pagination. The last two categories are related to some advanced functionalities of the framework, which are not of interest to this thesis. Custom methods are a mechanism which enable methods declared in the `Repository` interface to be passed to a specific implementation, while pagination offers the possibility to specify the page number for the results of a query. Both categories are not present in every framework extension; for example, pagination is tested only in the querybuilder module for JPA.

```
1  // Test for single condition in QueryBuilder JDBC
2  @Test
3  public void oneCondition() {
4    visitor.visitEntity("Person");
5    visitor.visitCondition("Personname", ComparisonType.EQUALS);
6    visitor.visitEnd();
7
8    String query = visitor.getQuery();
9
10   assertEquals(
11       query,
12       "select person.id, person.name, person.lastname, person.age, address.id
             , address.city, address.state from person, address where person.
             personname = 1? and person.address_id = address.id");
13 }
14
15 // Test for single condition in QueryBuilder JPA
16 @Test
17 public void oneCondition(){
18   visitor.visitEntity("Person");
19   visitor.visitCondition("name", ComparisonType.EQUALS);
20   visitor.visitEnd();
21
```

```
22    QueryRepresentation qr = visitor.getQueryRepresentation();
23    String query = qr.getQuery().toString();
24
25    assertEquals(query,"SELECT o FROM Person o WHERE o.name = :nameEquals");
26  }
```

Listing 3.10: Unit tests for single condition in JDBC and JPA

Code fragment 3.10 shows an example of the unit test for a query with one condition for JDBC and JPA, these tests are related to the implementation of the `QueryVistor` interface. Notice how the two tests are verifying the same behavior, they only differ in the specific implementation of the underlying database technology.

Chapter 4

QueryBuilder Extension for Cassandra

4.1 Cassandra Particularities

As described in section 2.4, Cassandra is a NoSQL distributed database technology in which data is replicated among different nodes to provide resistance to failures. Due to its nature, Cassandra does not fully support queries based on the relational model constructs of classic SQL databases. Therefore, the implementation of a QueryBuilder extension for this storage technology must take these differences into account. The particularities that can be identified for Cassandra and that need a different implementation approach compared to the other extensions are described next.

4.1.1 Join queries

Cassandra does not support the concept of joins as in normal relational databases. To perform such an operation, the two possibilities are: either to store an additional table with the results of the join or to perform the operation at the application level[23]. In addition, Cassandra does not support the concept of referential integrity constraints from one table to another.

4.1.2 Order-by queries

The concept of ordering in Cassandra CQL queries is somewhat different and limited compare to SQL databases. Order-by queries in Cassandra have two limitations: they can only be run on clustering keys and they cannot be run on every row in the table. Clustering keys are special columns that define the order of the table partition and are declared along with the primary key when the table is created.

```
1 CREATE TABLE Person(
2     id int,
3     name text,
4     lastname text,
5     PRIMARY KEY (id,name)
6 );
```

Listing 4.1: Primary key and clustering key in Cassandra

Code fragment 4.1 describes an example of table declaration for Cassandra, where both the primary key and the clustering key are declared. The `PRIMARY KEY` statement on line 5

declares `id` as the partition key and `name` as the clustering key[23]. In Cassandra, order-by queries are allowed only for clustering keys, and, in addition, the query must have at least one equivalence or `IN` statement.

```
-- Valid order by queries
SELECT * FROM person WHERE id IN (1,2,3) ORDER BY name
SELECT * FROM person WHERE name = 'Paul' ORDER BY name DESC

-- Invalid order by queries
SELECT * FROM person ORDER BY name
SELECT * FROM person WHERE name IN ('Paul','Max') ORDER BY lastname
```

Listing 4.2: Valid and invalid order by queries in Cassandra

The last two queries shown in code fragment 4.2 are not valid, the first one is missing an equivalence or IN statement, the second one is not using a clustering key for the order-by clause.

4.1.3 OR operator

Cassandra does not support the `OR` operator in queries[24], only the `AND` operator is valid. One possible solution is to use the operator `IN` followed by the values searched for the attribute.

```
SELECT * from Person WHERE id IN (1,2,3)
```

Listing 4.3: OR operator alternative in Cassandra

Code fragment 4.3 shows a possible alternative to a query with `OR` statements in Cassandra, where the `IN` clause is used instead. Although this is a possible solution, it is limited only to primary-key columns, so in the shown example `id` must be a primary key, furthermore, each query can contain at most one `IN` statement.

4.1.4 Unsupported comparison types

Cassandra only supports the following conditional operators in the `WHERE` clause: IN, =, >, >=, <, or <=, but not all in certain situations. The other comparison types that can usually be found in RDBMSs are not supported, those are: `<> (NOT EQUALS)`, `STARTS`, `ENDS` and `CONTAINS`[24]. In addition also comparing to null values is not supported by Cassandra.

4.2 Development Approach

The QueryBuilder extension for Cassandra has been developed by following an iterative and test-driven approach. The TDD development strategy involves writing the tests before the actual implementation of the functionality, and is one of the common practices of extreme programming[25]. By using this approach, virtually every method and class in the software has an associated test. This is especially useful when code needs to be refactored, because if a change introduces a new bug it can be detected immediately.

Typically, the TDD development strategy involves writing tests from scratch, but since Esfinge QueryBuilder already has a set of unit and integration tests to verify the features

implemented by the framework extensions, as described in subsection 3.4.1, the new extension can be developed by taking existing tests and adapting them to the features expected by Cassandra. Thus, in each iteration of our test-driven development process we performed the following steps:

1. We selected one of the core test classes from the test suites of the other Query-Builder extensions, for example the unit tests for the `QueryVistor` interface.

2. We copied the test class into our test suite and applied the naming convention of adding the word "Cassandra" to the class name.

3. For each test case in the test class, we adapted it to the specific implementation and expectation for Cassandra. An example of an adaptation is the translation of an SQL query into a Cassandra CQL query.

4. For each adapted test, we developed the functionality until all the tests where successful.

5. At the end of the process, if a special feature of Cassandra was identified that required additional functionality, we first developed the new tests and then implemented the new functionality.

```
1  @Test
2  public void oneConditionTest() {
3      visitor.visitEntity("Person");
4      visitor.visitCondition("name", ComparisonType.EQUALS);
5      visitor.visitEnd();
6
7      QueryRepresentation qr = visitor.getQueryRepresentation();
8      String query = qr.getQuery().toString();
9
10     assertEquals(
11             "SELECT * FROM <#keyspace-name#>.Person WHERE name = 0? ALLOW \
                   FILTERING",
12             query);
13 }
```

Listing 4.4: Unit tests for one condition in Cassandra

Code fragment 4.4 shows an example of the unit test for a query with one condition for Cassandra, developed by following the same principles used in 3.10. We followed this iterative development process until all unit tests were successful and did the same for integration tests with a real instance of a Cassandra database. The test suite provided by the framework's other extensions, as explained in the 3.4.1 section, was used to ensure that the main features were implemented correctly, including the ability to express queries that are not normally supported by Cassandra, such as those with sorting, joins, or null parameters.

4.3 Implementation

4.3.1 Implementing the extension points

As described in section 3.4, the querybuilder-core module has 5 interfaces that need to be implemented. The implementation approach for the different extension points is described below:

1. `EntityClassProvider`: the purpose of this interface is to identify the entities for the framework. We analyzed the approach adopted by the other extensions of the the framework, for example the JDBC extension uses an XML file to map the entity classes and make them available. The strategy that we decided to adopt was to use the marker interface pattern[26], which consists in an empty interface whose only purpose is to signal a special behavior of a class. The `ServiceLoader` class can be used to load implementations of the classes signaled by the marker interface.

2. `QueryVisitor`: for the concrete implementation of this interface we added multiple attributes that keep track of the statements that are interpreted from the method name, for example, a `List` of `ConditionStatement` objects keeps track of all the conditions encountered during the translation process.

```
public class ConditionStatement extends Clause {

    private ComparisonType comparisonType;
    private NullOption nullOption = NullOption.NONE;
    private String nextConnector = null;

    // Constructor, getters and setters omitted
}

public class Clause {

    protected String propertyName;
    protected Object value = null;
    protected int argPosition;

    // Constructor, getters and setters omitted
}
```

Listing 4.5: The ConditionStatement class

In particular, the `ConditionStatement` class depicted in code fragment 4.5 stores the information passed to the `visitCondition()` method of the `QueryVisitor` class. For example if a method in the `Repository` class has the naming `getPersonByNameEquals()`, the `QueryVisitor` will store a `ConditionStatement` object with the property name equal to "Name" and the comparison type corresponding to `EQUALS`. The class `comparisonType` is an enum class of the QueryBuilder core module that is used to express the type of comparison.

```
private String getConditionRepresentation() {
    StringBuilder sb = new StringBuilder();

```

```
4       if (nullOption == NullOption.NONE) {
5           sb.append(propertyName + " " + comparisonType.getOperator() + "↘
                " + getValueRepresentation());
6       } else if (nullOption == NullOption.IGNORE_WHEN_NULL) {
7           if (value != null)
8               sb.append(propertyName + " " + comparisonType.getOperator()↘
                    + " " + getValueRepresentation());
9       }
10
11      return sb.toString();
12  }
```

Listing 4.6: getConditionRepresentation method of ConditionStatement class

Code fragment 4.6 describes the implementation of the method developed for the `ConditionStatement` class, which can translate the object values into a `String` that can be added to the query created in the `QueryVisitor`. The method `getValueRepresentation()`, found in the same code fragment 4.6 maps the value attribute into its query representation based on type, e.g. if the value is a `String`, it is enclosed in two apostrophes. In addition, during the implementation of this interface we added additional fields that handle the Cassandra particularities.

3. `QueryRepresentation`: This interface is instantiated by the `QueryVisitor` interface, which is why we decided to pass additional fields to the constructor of this class for Cassandra's special cases. This made it possible to handle the special cases for the `getQuery()` method of this interface.

4. `QueryExecutor`: this interface is responsible for executing the query on the database and returning the results as an Object, for this reason it needs to manage the connection and the mapping of objects to and from the Cassandra database. In order to cover these two aspects we decided to use the Datastax[24] driver. This library offers utility methods for the management of the connection and provides the `MappingManager` class, which has the capability of performing the mapping of the objects to and from the database. In addition, we made the database connection rely on an interface with only one method to return the Session object; thus, the application using the framework will have to provide its own way of connecting to the database.

```
1  public interface CassandraSessionProvider {
2
3      Session getSession();
4  }
```

Listing 4.7: Cassandra Session provider

Code fragment 4.7 describes the interface that the framework uses to provide the connection to the specific database.

5. `Repository`: the purpose of this interface is to provide the basic CRUD operations for the database, like save and delete, and can later be extended with the custom methods that are interpreted by the framework, as described in subsection 3.2.1. For the implementation of this interface it is sufficient to write the functionality for the basic

CRUD methods, and for this purpose we used the `MappingManager` class as for the implementation of the `QueryExecutor` interface.

4.3.2 Implementing the Cassandra particularities

To account for the particularities of Cassandra described in 4.1, additional application logic on top of the framework is required. In the framework's JDBC or JPA extensions, sorting can be done with a simple `ORDER BY` statement, but with Cassandra this is not possible. The same concept can be extended to the other special features of Cassandra that cannot be solved with a query.

In order to solve these problems we decided to provide a mechanism that performed a post processing on the results, this was achieved by using the "Chain of Responsibility" pattern[27]. This pattern involves the creation of a chain of processor objects, each holding a reference to the next processor. Each processor applies an operation to a list of objects and forwards it to the next processor, which does the same, until it reaches the last processor in the chain.

To implement this pattern, it is necessary to create an abstract class for the processor, which is inherited by the concrete processors. The concrete processors will have their own implementation of the method for processing the List and will also contain a reference to the next processor.

```
1  public abstract class BasicResultsProcessor implements ResultsProcessor {
2      private ResultsProcessor nextResultsProcessor;
3
4      public BasicResultsProcessor() {
5          super();
6      }
7
8      public BasicResultsProcessor(ResultsProcessor nextPostprocessor) {
9          super();
10         this.nextResultsProcessor = nextPostprocessor;
11     }
12
13     @Override
14     public <E> List<E> postProcess(List<E> list) {
15         list = resultsProcessing(list);
16
17         if (nextResultsProcessor != null)
18             list = nextResultsProcessor.postProcess(list);
19
20         return list;
21     }
22
23     public abstract <E> List<E> resultsProcessing(List<E> list);
24 }
```

Listing 4.8: Abstract processor implementation

Code fragment 4.8 describes the implementation of the abstract processor class that were developed for the Cassandra framework extension. Note how the implementation of the `resultsProcessing()` method is left to the concrete classes that extend the abstract class `BasicResultsProcessor`.

```
1 ResultsProcessor proc = new ConcreteProcessor1(new ConcreteProcessor2());
2
3 List<Object> processed = proc.postProcess(results);
```

Listing 4.9: A Chain of ResultsProcessor objects

Code fragment 4.9 shows how a chain of `ResultsProcessor` objects can be created by passing successive processors to the constructor, after which the call to the `postProcess()` method executes the `resultsProcessing()` method for each processor in the chain.

4.4 Implementation Details

The final version of the QueryBuilder module for Cassandra consists in the concrete implementation of the five extension points described at 3.4. The source code of the developed software is available as open source in the official repository of the Esfinge QueryBuilder framework[22] and also separately in another repository created for this thesis project[28]. All unit and integration tests derived by the other QueryBuilder framework plugins have been adapted and included in the software module. This chapter focuses in particular on the architecture and final results of the implementation of the additional application logic layer that handles the particularities described in 4.1. By following the implementation strategy explained in section 4.3.2 we developed a total of four results processors.

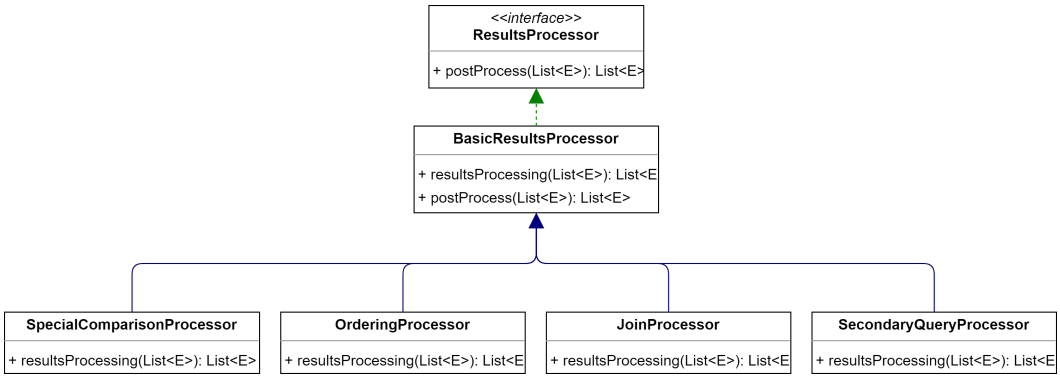

Figure 4.1: Results processors UML

Figure 4.1 shows the UML diagram of the inheritance tree of the implemented processors. The next subsections provide implementation details of the different processors and how they solve the integration problems with Cassandra.

4.4.1 SpecialComparisonProcessor

This processor solves the problem of unsupported comparison types in Cassandra, namely `<> (NOT EQUALS)`, `STARTS`, `ENDS`, `CONTAINS` and also comparing to `null`.

Figure 4.2 shows the classes that were implemented for this functionality. The idea for the implementation is to create a `List` of `SpecialComparisonClause` objects in the query visitor when a special condition statement is parsed. `SpecialComparisonClause` is similar to the `ConditionStatement` class described in code snipped 4.5, but uses the SpecialComparisonType enum, which encapsulates only the special types of comparisons, for example `STARTS` and `CONTAINS`. The `SpecialComparisonProcessor`, which also accepts the `List`

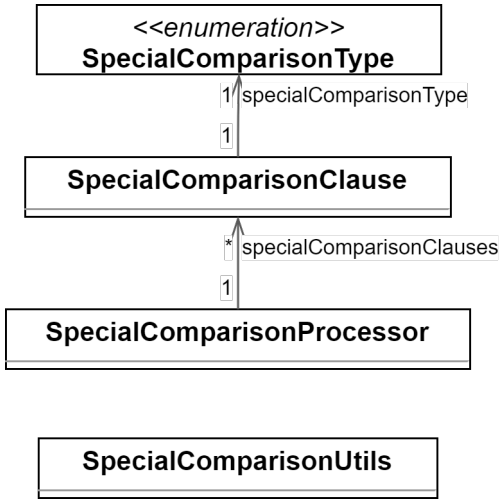

Figure 4.2: SpecialComparisonProcessor classes

of `SpecialComparisonClause` objects in the constructor, performs the processing by using a utility method of the `SpecialComparisonUtils` class, which uses reflection to filter the List. Thanks to the implementation of the `SpecialComparisonProcessor` it is possible to express queries in Cassandra which use comparison operators that would normally not be supported.

```
1 public interface PersonDAO extends Repository<Person> {
2     List<Person> getPersonByNameNotEquals(String name);
3     List<Person> getPersonByNameStarts(String name);
4 }
```

Listing 4.10: Methods with special comparison

Code fragment 4.10 provides two examples of methods with special comparison clauses that can be interpreted by the Cassandra framework extension.

4.4.2 JoinProcessor

Implementing joins for Cassandra is particularly cumbersome, for this reason the solution proposed to solve this problem has been limited to a joining depth of 1, this means that if an entity has a custom attribute, that attribute will have to contain only primitive types or primitive wrapper classes, for example `int`, `double`, `Integer` and `String`.

Figure 4.3 shows the classes that were implemented for this functionality. Similarly to the solution for the special comparison, we created a `List` of `JoinClause` objects in the `QueryVisitor`. The difference between `SpecialComparisonClause` and `JoinClause` is that the former supports only the special comparison types, while the latter supports also the standard ones, like `EQUALS`, `GREATER` and so on. The `JoinProcessor` class filters the results with a utility method of the JoinUtils class, which uses reflection to compare the value in the JoinClause with the one contained in the custom attribute of the object. In this case, values are compared with a method similar to that used by the processor for special comparison clauses, but the attributes of the nested class are accessed instead. As explained before, with the `JoinProcessor` the framework can express join queries for custom attributes only to a depth of 1.

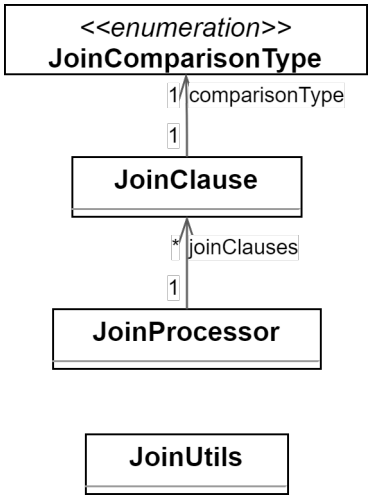

Figure 4.3: JoinProcessor classes

```
1 public interface PersonDAO extends Repository<Person> {
2
3     List<Person> getPersonByAddressCity(String city);
4     List<Person> getPersonByAddressCityAndAddressState(String city, String ↘
          state);
5
6 }
```

Listing 4.11: Methods with join

Code fragment 4.11 shows two examples of methods with joins that are supported by the Cassandra framework. The examples suppose that the class `Person` has a custom attribute `Address` and fields `city` and `state`, both of type `String`.

4.4.3 OrderingProcessor

This processor solves the problem of Cassandra not fully supporting ordering queries. As as explained in subsection 4.1.2, the standard usage of order-by queries is really limited, for this reason we implemented a processor that can reorder the results based on the attributes. The idea for this implementation is to first query the database by only considering the comparison statements, and at the end sort it based on the attributes specified in the order-by statement.

Figure 4.4 shows the classes that were implemented for achieving the same behavior as order-by queries. An important class is the `ChainComparator` class, which encapsulates a `List` of `Comparator` objects and then invokes the specific implementation of the `compare` method for each `Comparator` in the List.

```
1 public class ChainComparator implements Comparator<Object> {
2     private final List<Comparator> comparatorList;
3
4     public ChainComparator(List<Comparator> comparatorList) {
5         this.comparatorList = comparatorList;
6     }
7
```

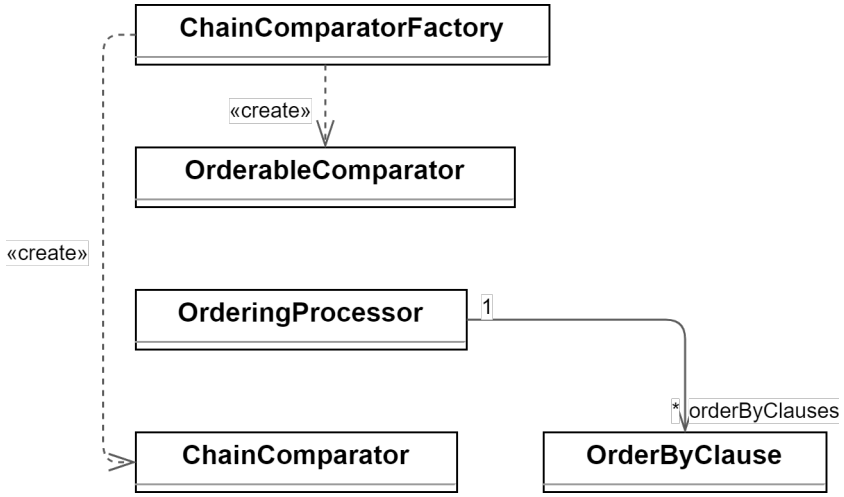

Figure 4.4: OrderingProcessor classes

```
8       @Override
9       public int compare(Object obj1, Object obj2) {
10          int result;
11          for (Comparator<Object> comparator : comparatorList) {
12              if ((result = comparator.compare(obj1, obj2)) != 0) {
13                  return result;
14              }
15          }
16          return 0;
17      }
18  }
```

Listing 4.12: ChainComparator class

We create a `ChainComparator` object with the `ChainComparatorFactory` class, which adds `OrderableComparator` objects to the `List` of `Comparator` objects. An orderable comparator is an implementation of a `Comparator` object, in which the direction of sorting can be defined as ascending or descending. This implementation allows the flexibility of performing one or multiple order-by queries on arbitrary attributes of the objects, instead of being limited to the ones defined by Cassandra.

```
1  public interface PersonDAO extends Repository<Person> {
2
3      List<Person> getPersonOrderByName();
4      List<Person> getPersonByAgeOrderByNameDesc(Integer age);
5
6  }
```

Listing 4.13: Methods with order-by

As an example, when the framework will parse the method `getPersonOrderByName()`, it will generate and execute a query that selects all the persons in the database, and after that it will use the `OrderingProcessor` to order them by the attribute `name`.

4.4.4 SecondaryQueryProcessor

This processor solves the lack of support of Cassandra for the `OR` operator. The idea is to create additional queries when an `OR` statement is encountered, and later merge the results into one single List. The implementation of the `resultsProcessing()` method for the `SecondaryQueryProcessor` class simply filters the results by removing duplicate elements.

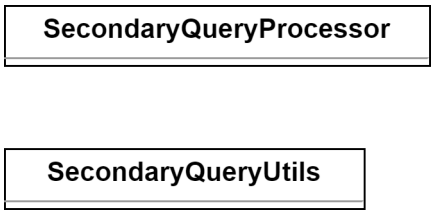

Figure 4.5: SecondaryQueryProcessor classes

Figure 4.5 shows the classes that perform the filtering or the results. The utility class defines the important method `reflectiveEquals()`, which is used to compare two objects, but works even if they do not have an implementation of the `equals()` method. The secondary queries are created in the `CassandraChainQueryVisitor` class, which is able to create a chain of `QueryVisitor` objects. If the `OR` instruction is encountered during the interpretation process, a new `QueryVisitor` object is initialized and subsequent instructions are sent to it.

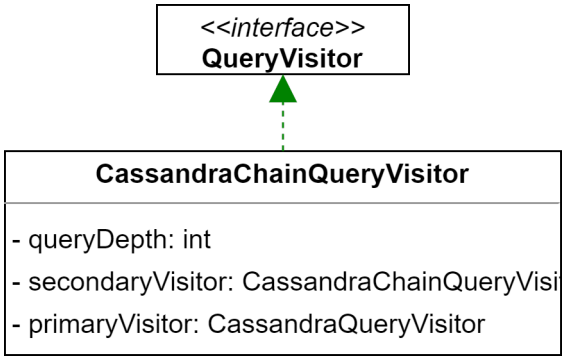

Figure 4.6: CassandraChainQueryVisitor class

Figure 4.6 shows the main attributes of the CassandraChainQueryVisitor class. The secondary visitor is also a `CassandraChainQueryVisitor` object, which means that it can create a chain of `QueryVisitor` objects when it encounter new `OR` statements.

```
1 public interface PersonDAO extends Repository<Person> {
2     List<Person> getPersonByNameOrLastName(String name, String lastname);
3     List<Person> getPersonByNameOrLastNameOrAge(String name, String lastname,\
          Integer age);
4 }
```

Listing 4.14: Methods with or statement

With this implementation, the framework is able to support `OR` statements in methods, as shown in code fragment 4.14.

```
-- Main query
SELECT * FROM Person WHERE name = 'Paul' ALLOW FILTERING;

-- Secondary query
SELECT * FROM Person WHERE lastName = 'Max' ALLOW FILTERING;
```

Listing 4.15: Interpretation of method with OR statement

A call of the method `getPersonByNameOrLastName("Paul","Max")` is parsed by the framework and translated into two separate queries, which are shown in code fragment 4.15. At this point the results returned by the two queries is merged and filtered by the `SecondaryQueryProcessor`.

Chapter 5

Evaluation

5.1 Objective

The objective of this chapter is to present the evaluation process and the results obtained by the usage of the QueryBuilder Cassandra extension on an application. In particular the goal is to asses the ease with which it is possible to migrate from a relational database to Cassandra by using the QueryBuilder framework.

5.2 Methodology

The methodology adopted for the evaluation involves five steps:

1. We chose a Java application that performs basic CRUD operations on a database and allows the user to interact with it through the browser.

2. We selected an Esfinge QueryBuilder extension for a relational database, for this evaluation: JDBC with MySQL.

3. We created two alternative versions of the application, one integrating the QueryBuilder extension for the relational database, the other one integrating the Cassandra extension.

4. We created an end-to-end test suite in order to test that the functionality of the application is equivalent for both version of the application.

5. We evaluated the changes needed to migrate from one extension to the other quantitatively and qualitatively.

The metrics used for the quantitative evaluation were the number of lines of code added, deleted, and modified for each file that was subject to a change after migration. For the qualitative evaluation, the role and impact of the files on the complexity of the migration were considered instead.

5.3 Application Implementation

5.3.1 The CRUD application

As the base for the evaluation we started from a tutorial Java Servlet application[1]. The starting application supports the four main CRUD operations and is integrated with a MySQL database, the user can interact with the browser and can visualize, create, delete or update books.

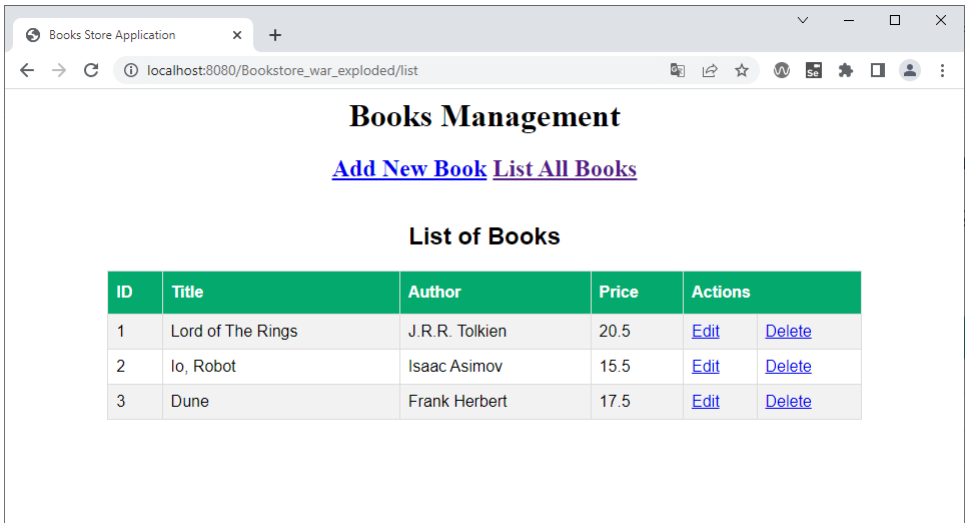

Figure 5.1: The bookstore application main page

Figure 5.1 shows the main page of the CRUD application used for the evaluation. When the user clicks on "Edit" or on "Add New Book", he is redirected to a page with a form where he can modify an existing book or create a new one. The original version of the application has a `BookDAO` class, in which the CRUD operations are specifically implemented for the MySQL database. The `BookDAO` class is the equivalent of the `Repository` class of the Esfinge QueryBuilder class, for this reason the application was modified to use the QueryBuilder class instead.

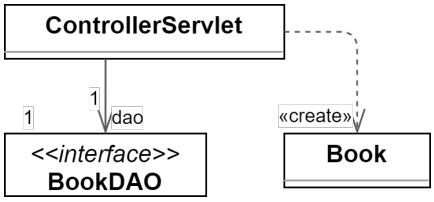

Figure 5.2: The bookstore application classes

Figure 5.2 describes the main classes of the application. The `ControllerServlet` class serves the correct JSP page based on the request and orchestrates the creation, deletion, or update of Book elements. The Book class is used to represent the book entities of the application. On top of the basic version of the application, two different versions were developed, each configured to use a different QueryBuilder extension:

[1] Nam Ha Minh, JSP Servlet CRUD application, `https://www.codejava.net/coding/jsp-servlet-jdbc-mysql-create-read-update-delete-crud-example`

- `JDBC version`: we added the maven dependencies for the querybuilder core module, querybuilder-jdbc and mysql-connector-java. In addition we added the implementation of the `DatabaseConnectionProvider` interface, which manages the database connection for the QueryBuilder framework for JDBC. We also added the annotations required by the framework to the Book class, and we configured the `Entities.xml` file with the name of the entities. Finally, we declared the implemented services in the META-INF.services folder, which is only necessary for the concrete implementation of the `DatabaseConnectionProvider` interface.

- `Cassandra version`: we added the Maven dependencies for the querybuilder core module and the Datastax drivers for Cassandra. The dependency for querybuilder-cassandra is a local jar file, since the extension is not released on Maven. Like for JDBC we added the interface implementation for the connection to the database, in this case the name of the interface is `CassandraSessionProvider`. In addition we added the proper annotations to the Book class and added the `CassandraEntity` marker interface, which is the technique used by the Cassandra extension to identify the entities. As for JDBC, we declared the services we implemented in the `services` folder, in this case we declared the `CassandraSessionProvider` concrete implementation and the `CassandraEntity`. Finally we also added the `config.json` file which allows setting the maximum number of entities that can be ordered with a query and the maximum number of secondary queries.

The source code for the two versions of the application is available as open source in the repository created for this thesis project[28].

5.3.2 Functional test suite

The test suite has been developed with Cypress[29], a JavaScript end to end testing framework built for web applications. With this technology, it is possible to create tests that rely only on the visual part of a Web application, selecting page elements based on the text they contain or CSS selectors. The testing framework simulates the user interaction with the browser, and in case some functionality is not working correctly, the test will fail. Four test cases covering the main functionality of the application were developed:

- Creation of a new book
- Editing of book
- Deletion of book
- Error message when the book price is invalid

Figure 5.3 shows the end to end test for the deletion of a book. This test suite was used on both versions of the application, the one integrated with JDBC and the one integrated with Cassandra, as a control mechanism to ensure that the two are equivalent in terms of functionality and that the databases are properly integrated.

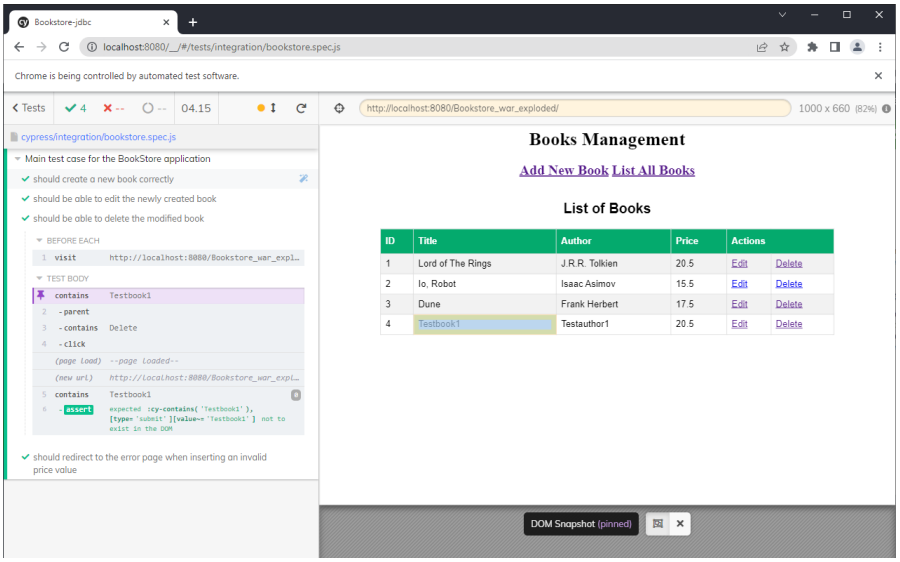

Figure 5.3: End to end test for book deletion

5.4 Evaluation of the Changes

For the evaluation of the changes we performed the following steps:

1. We created a repository on GitHub and committed the JDBC version of the Bookstore application.
2. We removed the JDBC version locally and replaced it with the Cassandra version and performed a new commit. In this way only the changes between the two versions are committed.
3. We analyzed the commit and the diff files to estimate the changes needed for switching from one extension to the other.
4. We also performed a new commit returning from Cassandra to JDBC to see if the change estimate also matched in the opposite direction.

5.4.1 Results Metrics

To estimate the commit size, the number of lines of code added, deleted and modified was taken as an approximation[30]. For the transition from QueryBuilder JDBC to QueryBuilder Cassandra the following results were obtained:

Table 5.1: Commit size - from QueryBuilder JDBC to Cassandra

File	Changes	Additions	Deletions
conf/Entities.xml	23	0	23
docker-compose.yml	17	6	11
pom.xml	44	39	5

Continues on next page.

Table 5.1, continued from previous page.

File	Changes	Additions	Deletions
sql/Bookstore.sql	15	6	9
Book.java	17	11	6
BookstoreCassandraSession-Provider.java	34	34	0
MySqlDatabaseConnection-Provider.java	23	0	23
CassandraSessionProvider (service file)	1	1	0
CassandraEntity (service file)	1	1	0
DatabaseConnectionProvider (service file)	1	0	1
resources/config.json	4	4	0
Total	180	102	78

Table 5.1 represents the size of the commit when changing the configuration from QueryBuilder JDBC to QueryBuilder Cassandra. The files whose name is colored in green represent added files, while the ones colored in red represent files that have been deleted. The files whose name is not colored represent files that have only been modified. As explained earlier, the same experiment was run in the opposite direction (from QueryBuilder Cassandra to JDBC) to have a double check for correctness. The results of the configuration in the opposite direction are consistent with the ones showed in table 5.1, the number of deletions and additions are simply reversed.

5.4.2 Qualitative Analysis

This section evaluates the quantitative results described in 5.4.1. The transition from JDBC to Cassandra involves changing different types of files, so the role and impact these files have on the application have to be categorized.

Environment configuration

The following are the files that are related to the configuration of the application, for this reason they are not closely related to the actual configuration of the framework.

- `docker-compose.yaml`: this file contains the docker definition that starts a docker container with the database. Switching from JDBC to Cassandra requires only changing the image definition and some configuration parameters.
- `sql/Bookstore.sql`: this file contains the SQL definitions for creating the database and the book table. The changes between the JDBC version and the Cassandra version only involve syntactical differences between the two query dialects.

It can be concluded that the environment configuration update required the modification of two files, although it is unclear whether these changes should be considered in the overall analysis, since these files are not specifically related to the QueryBuilder framework.

Framework configuration.

The following are the files that are related specifically to the configuration of the framework

- `conf/Entities.xml`: this configuration file is used by the JDBC version of the framework to map the entities, in Cassandra the same behaviour is achieved by using a marker interface, as explained in section 4.3.1, for this reason this file just needs to be deleterd.
- `pom.xml`: this file is used by Maven to configure the dependencies and the plugins of the project. In our experiment we had to remove two dependencies and add six. The two removed dependencies are related to JDBC and are respectively the driver for the connection to the MySQL database and the query-builder JDBC module. The added dependencies include two Datastax driver modules, used to connect to the Cassandra database, and two logging libraries, on which the drivers depend. Finally, the dependencies for the Cassandra query-builder module and for the `Json` parsing need to be added, the latter of which is used by the Cassandra framework to parse the configuration file.
- `resources/config.json`: this file is used by QueryBuilder Cassandra for configuring the maximum number of entities that can be ordered and the maximum number of secondary queries supported, the latter configuration attribute defines the maximum number of OR statements supported in a query. If this file is omitted, the framework reverts to a default configuration, but for the purposes of quantitative analysis we decided to include it in the count.
- `service files`: the service files are located in the `resources/META-INF/services` folder, their purpose is to declare the implementation of the framework interfaces, such that they can be found by the `ServiceLocator` class used by the framework. To switch from JDBC to Cassandra, the `DatabaseConnectionProvider` implementation needs to be deleted and the `CassandraSessionProvider` interface needs to be implemented, both interfaces are used for database connection. In addition, it is necessary to create a service file for Cassandra entities and insert the classes implementing the `CassandraEntity` interface.

It can be concluded that updating the framework configuration in the experiment required the deletion of two files, the creation of three new files, and the modification of one file. Also, it is worth noting that for each additional entity in the application, the Cassandra entity declaration must be added to the service file. This operation does not require an additional service file, but only one additional line for each `CassandraEntity`.

Domain classes.

As for the application domain, only one file needs to be modified for the transition, namely the `Book.java` file. The changes to this class mainly concern metadata; the following code snippet highlights in red the difference in the `Book.java` file when switching from QueryBuilder JDBC to QueryBuilder Cassandra:

```
1 // Book.java for QueryBuilder JDBC
2 @Table(name="book")
```

```
public class Book {
  @ID
  private int id;
  private String title;
    // Rest of class omitted ...
}

// Book.java for QueryBuilder Cassandra
@Table(keyspace = "bookstore", name = "book",readConsistency = "QUORUM",writeConsistency =
    "QUORUM",caseSensitiveKeyspace = false,caseSensitiveTable = false)
public class Book implements CassandraEntity {
  @PartitionKey
  private int id;
  private String title;
    // Rest of class omitted ...
}
```

Listing 5.1: Book.java changes from JDBC to Cassandra

The transition involved updating the metadata on the `Book` class. The `@Table` annotation is updated to the Datastax version and the `@ID` annotation is substituted with a similar annotation that fulfills a similar task, the `@PartitionKey` annotation. In addition, the marker interface `CassandraEntity` needs to be added to the Entity. If there were more than one entity, this operation should be repeated on all of them, in addition to declaring their existence in the service file, as explained in the analysis of the framework configuration files 5.4.2.

Framework specific classes.

These are the files that are specific to the framework extension:

- `MySqlDatabaseConnectionProvider.java`: this class is the implementation of the `DatabaseConnectionProvider` interface for the JDBC version of the framework, thus it can be deleted.
- `BookstoreCassandraSessionProvider.java`: this class is the implementation of the `CassandraSessionProvider` interface for the Cassandra version of the framework, and needs to be defined in order to connect to the Cassandra database.

Both classes of the framework versions have the same role, namely connecting to the specific implementation of the database. Regardless of the number of entities this operation needs to be done only once, along with the service declaration. The `BookDAO` class, which is an extension of the framework-specific `Repository` class, remained unchanged during the transition. However, it is worth noting that if there were more entities, it would be necessary to provide more DAO classes.

5.5 Discussion

In section 5.4.2 the changes needed for the transition from QueryBuilder JDBC to Cassandra are evaluated with a qualitative approach. Each modified file is defined by category and contribution to configuration complexity. In addition, it is possible to distinguish between

operations that need to be performed only once and those that need to be performed several times, depending on the functionality and size of the application.

One-time operations: The operations that need only to be performed once are the creation of the class for the Database connection, the creation of the configuration file for the framework limits, the creation of the service files for the `CassandraSessionProvider` and `CassandraEntity` implementations and finally the configuration of the dependencies of the application. All of these operations are relatively simple to perform, as long as the person performing the configuration has the framework documentation available. The most complicated procedure is the configuration of the dependencies, but with the aid of a management tool like Maven[31], the complexity is reduced substantially. Moreover, these configurations are independent of the size of the application, e.g., the presence of multiple entities does not require the repetition of any of them.

Operations to be repeated: Unlike operations that need to be performed only once, there are some configuration procedures that, depending on the size of the application, may need to be repeated. Each application entity requires updating the annotations, adding the marker interface `CassandraEntity`, and declaring the entity name in the service file. It is important to point out that editing an annotation, which is a metadata configuration, is easier than editing lines of code within a method or class. In addition, it is noted that for each entity only two annotations need to be modified and an empty interface implemented, so this type of operation can be considered simple. Finally, a DAO interface must be defined for each entity, since the `Repository` interface of the QueryBuilder framework can only be configured with one entity type. It should be noted, however, that defining the DAO interface is relatively simple, especially if the application performs only CRUD operations, in which case the interface only needs to be declared and parameterized. The framework's method interpretation feature simplifies the creation of new queries to the database; for example, to add a search feature to the Bookstore application, where results can to be filtered by a price range, as with communicating and retrieving data from the database, it is only necessary to add a new method to the DAO interface.

```
1 List<Book> getBookByPriceAndPriceOrderById(@GreaterOrEquals Float ↘
    priceLowerRange, @LesserOrEquals Float priceUpperRange);
```

Listing 5.2: Adding a new functionality to the application

The method shown in code snippet 5.2 performs the same logical operation regardless of whether the storage system is JDBC or Cassandra. The application can ignore the database implementation details and can simply provide a lower and upper bound for the price.

It has been shown that transitioning from QueryBuilder JDBC to Cassandra requires several types of procedures to be performed, some are one-time operations, others must be performed multiple times depending on the size of the application. Overall, it was found that the configuration procedures are not complex in terms of the number of lines of code to be changed, and the number of files involved in the changes is also quite limited. The configuration that requires the most work is updating dependencies, but if an accurate list of dependencies is provided and if a management tool such as Maven is used, this can be done without any particular problems.

5.6 Limitations of the evaluation

This last section explains the limitations of the evaluation process that was used to asses the ease of transition from a relational database to Cassandra by using the QueryBuilder framework. One limiting factor has to do with the scope and the size of the application. In the experiment, the QueryBuilder `Repository` interface was used to declare only one custom method, namely `getBookOrderById()`, which creates a query with sorting based on the `id` attribute. Since the application is relatively small and requires only basic CRUD operations, only the predefined methods provided by the QueryBuilder `Repository` interface were used, and more complex queries were not experimented with.

Another limitation of our evaluation has to do with the fact that the application was built to handle only one type of entity; as explained in section 5.5, having multiple entities would have required more work in terms of configuration.

Finally, another limitation of the evaluation process is that the experiment was conducted by the developers of the framework extension themselves, who have firsthand knowledge of how the software works and how it should be configured. To overcome this limitation, the same experiment should be proposed to a group of participants who are given instructions on how to configure the framework and who have to reproduce the configuration process. Then, by collecting the data, the assessment of the complexity of the framework configuration would be more realistic.

Chapter 6

Conclusion and Further Studies

In this thesis we have seen the implementation of a new extension for an existing Java framework for polyglot persistence, namely Esfinge QueryBuilder. The plugin allows the framework to create a persistence layer for Cassandra, treating it as if it were a SQL-based relational database. The importance of developing such a tool lies in the fact that it reduces the configuration and maintenance effort for systems that need to migrate from one type of database to another, or even for applications that use more than one of these storage systems simultaneously. A persistence layer that can abstract the details of the underlying storage technology can enable the creation of flexible applications that can freely choose the type of database according to the requirements of the domain, e.g., in some cases a NoSQL storage technology might be more appropriate than a SQL-based one.

We have seen that Cassandra, compared to other database technologies, required the treatment of some special cases. In particular, it was pointed out that Cassandra does not support join queries, order-by queries performed on most column types, queries with `OR` connectors, and finally some comparison types, such as `STARTS`, `CONTAINS`, `LIKE` and others. It was shown how these particularities were considered in the design of the Cassandra framework extension, which required the implementation of an abstraction layer based on the Chain of responsibility pattern. The implementation details described how a total of four processing units were developed, each of which handles one of Cassandra's four main particularities, for example, it was shown how the `OrderingProcessor` class performs sorting of the results obtained, which enables the full expression of the concept of order-by queries.

We also showed how we leveraged the test-driven development approach to create a framework extension that mirrored the behavior of the existing ones and were able to ensure that the core functionalities of the framework extension were implemented correctly.

It was shown how we performed an experiment integrating the new framework extension into a real application and how we were able to evaluate the ease of configuration in migrating from a relational database to Cassandra. Specifically, the evaluation showed that the solution developed, in order to migrate from a relational database, needed some specific configuration. Quantitative results showed that the number of files to edit, delete, and create for the transition was quite limited. The qualitative analysis, on the other hand, showed the impact of each file on the migration effort, highlighting for example, that two of the changed files had to do only with the configuration of the database environment, such that they could be ignored by the overall analysis. In addition, it turned out that the changes made to the application classes during the transition, which theoretically could fall under

complex operations, almost exclusively involved changes to metadata, which are more immediate than changing lines of code in a method, and therefore can be considered simple. Finally, it was found that more complex operations, such as updating dependencies, were not among the procedures that need to be possibly performed more than once during migration.

The goal of this thesis was to reduce the effort of migrating an application from a relational database to Cassandra, while reducing the configuration problems of such migrations and, at the same time, being able to express all queries that would not normally be supported by Cassandra, such as those involving particular types of comparisons. It can be concluded that the goal was successfully achieved, because we were able to assess through our evaluation that the configuration part of the transition problem was limited to minor changes in metadata, creation of a limited number of classes, service declarations, and updating dependencies. In addition, our evaluation process confirmed that the main CRUD operations of the test application could be implemented using the Cassandra extension. Finally, the test suite including unit tests and integration tests demonstrated that the framework extension can express queries that are not normally supported by Cassandra, such as those with sorting, joins, or null parameters.

6.1 Future work

As shown in section 5.6, our evaluation process had some limitations, so a possible future work would be to extend the configuration experiment to a diverse group of developers. Also, since our evaluation only covered the transition from QueryBuilder JDBC to Cassandra, we might get other interesting results by testing the other storage technologies supported by the framework, e.g., we might evaluate the transition from MongoDB to Cassandra, which are both NoSQL databases. Possible future work might also involve extending the framework with further storage technologies which are currently not supported, for example Redis[15]. In addition, we could also introduce new features to make the Esfinge QueryBuilder framework even more suitable for polyglot persistence. For example, we could provide a universal type of annotations for entities, which could be translated for the specific database without the need to manually configure them.

Bibliography

[1] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 20(24):79, 2011.

[2] Michael Schaarschmidt, Felix Gessert, and Norbert Ritter. Towards automated polyglot persistence. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, 2015.

[3] Chaimae Asaad and Karim Baïna. Nosql databases – seek for a design methodology. In *Model and Data Engineering*, Lecture Notes in Computer Science, pages 25–40, Cham, 2018. Springer International Publishing.

[4] Abdul Wahid and Kanupriya Kashyap. Cassandra—a distributed database system: An overview. *Emerging technologies in data mining and information security*, pages 519–526, 2019.

[5] Prof. Eduardo Martins Guerra. Esfinge querybuilder. `http://esfinge.sourceforge.net/Query%20Builder.html`.

[6] Mark Richards. *Software architecture patterns*, volume 4. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA . . . , 2015.

[7] William Crawford and Jonathan Kaplan. *J2EE Design Patterns: Patterns in the Real World*. " O'Reilly Media, Inc.", 2003.

[8] Bruce Eckel. *Thinking in JAVA*. Prentice Hall Professional, 2003.

[9] 1962-autor Evans, Eric. Domain-driven design : tackling complexity in the heart of software, 2004.

[10] Cătălin Tudose and Carmen Odubăşteanu. Object-relational mapping using jpa, hibernate and spring data jpa. In *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*, pages 424–431. IEEE, 2021.

[11] Mike Keith, Merrick Schincariol, and Jeremy Keith. *Pro JPA 2: Mastering the JavaTM Persistence API*. Apress, 2011.

[12] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[13] Oliver Gierke et al. Spring data jpa-reference documentation. *URL https://docs. spring. io/spring-data/jpa/docs/current/reference/html/.[utoljára megtekintve: 2017. 04. 21.]*, 2012.

[14] Deka Ganesh Chandra. Base analysis of nosql database. *Future Generation Computer Systems*, 52:13–21, 2015.

[15] Josiah Carlson. *Redis in action*. Simon and Schuster, 2013.

[16] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012.

[17] C Chodorow. Introduction to mongodb. In *Free and Open Source Software Developers European Meeting (FOSDEM)*, 2010.

[18] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide: time to relax*. " O'Reilly Media, Inc.", 2010.

[19] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, volume 2324, 2013.

[20] Pwint Phyu Khine and Zhaoshun Wang. A review of polyglot persistence in the big data world. *Information (Basel)*, 10(4):141, 2019.

[21] Kriti Srivastava and Narendra Shekokar. A polyglot persistence approach for e-commerce business model. In *2016 International Conference on Information Science (ICIS)*, pages 7–11. IEEE, 2016.

[22] Prof. Eduardo Martins Guerra. Esfinge querybuilder. `https://github.com/EsfingeFramework/querybuilder`.

[23] Apache Foundation. Cassandra documentation. `https://cassandra.apache.org/doc/latest/`.

[24] Inc DataStax. Datastax documentation. `https://docs.datastax.com/en/home/docs/index.html`.

[25] David Janzen and Hossein Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.

[26] Howard C Lovatt, Anthony M Sloane, and Dominic R Verity. A pattern enforcing compiler (pec) for java: using the compiler. In *Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling-Volume 43*, pages 69–78. Citeseer, 2005.

[27] Steve Vinoski. Chain of responsibility. *IEEE Internet Computing*, 6(6):80–83, 2002.

[28] Samuel Dalvai Prof. Eduardo Martins Guerra. Esfinge querybuilder cassandra thesis source code. `https://github.com/samdalvai/esfinge-querybuilder-thesis`.

[29] Narayanan Palani. *Automated Software Testing with Cypress*. CRC Press, 2021.

[30] Philipp Hofmann and Dirk Riehle. Estimating commit sizes efficiently. In *IFIP International Conference on Open Source Systems*, pages 105–115. Springer, 2009.

[31] John Ferguson Smart et al. An introduction to maven 2. *JavaWorld Magazine. Available at: http://www. javaworld. com/javaworld/jw-12-2005/jw-1205-maven. html*, page 27, 2005.