ASSIGNMENT OF BACHELOR’S THESIS

Title: Analyze Scala Code Using Graph Database
Student: Bc. Otakar Vinklář
Supervisor: Ing. Filip Křikava, Ph.D.
Study Programme: Informatics
Study Branch: Computer Science
Department: Department of Theoretical Computer Science
Validity: Until the end of winter semester 2021/22

Instructions

In an OOPSLA paper [1], Krikava et al analyzed 18M lines of Scala code to find how developers use Scala implicits. The data were modeled as object trees stored in binary files using a Google protocol buffers. From these trees, they extracted relational data that were consequently analyzed in R. While it has worked, such an approach does not scale very well. It is cumbersome and the relational schema is not optimal to query for language patterns.

The goal of this thesis is to evaluate the possible use of a graph database (Neo4j) for querying patterns in large code networks.
This will require to design a graph database schema for the data from the OOPSLA paper, a tool that can feed the database from the binary files and recreate part of the analysis using the CYPHER query language. The original approach and the approach developed in this thesis should be then compared.

References

[1] 2019: F. Křikava, H. Miller, J. Vitek, Scala Implicits Are Everywhere, In PACMPL Issue OOPSLA 2019

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 17, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Analyze Scala Code Using Graph Database

Bc. Otakar Vinklář

Department of Theoretical Computer Science
Supervisor: Ing. Filip Křikava, Ph.D.

June 4, 2020

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on June 4, 2020 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2020 Otakar Vinklář. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Vinklář, Otakar. *Analyze Scala Code Using Graph Database.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Aby bylo možné vyvinout takový programovací jazyk, který budou lidé rádi využívat, je nutné znát, jak je tento jazyk používaný. V článku [KŘIKAVA, Filip; MILLER, Heather; VITEK, Jan. Scala implicits are everywhere: a large-scale study of the use of Scala implicits in the wild. Proceedings of the ACM on Programming Languages. 2019, vol. 3, no.OOPSLA, pp. 1–28.] Křikava a jeho kolegové provádí rozsáhlou studii použití implicitních konverzí a implicitních parametrů v jazyce Scala. Poslední část řešení této analýzy je ale těžkopádná, protože relační datový model není vhodný pro reprezentaci dat s velkým množstvím vzájemných vazeb. Tato práce se snaží zlepšit řešení této analytické části. Za úkol si dává především zjednodušit vyváření nových dotazů. Tyto problémy řeší za použití grafové databáze, která umožňuje ukládat data s velkým množstvím propojení. V této práci byla vybrána grafová databáze Neo4j s jejím dotazovacím jazykem Cypher. Tato práce ukazuje, že grafová databáze a její dotazovací jazyk nabízí vysokoúrovňové rozhraní pro statickou analýzu kódu.

Klíčová slova statická analýza kódu, Scala Implicits, Scala, grafové databáze, Neo4j, Cypher

Abstract

For a programming language to become as convenient as possible and to evolve it is necessary to understand how the language is used by the programmers. In paper [KŘIKAVA, Filip; MILLER, Heather; VITEK, Jan. Scala implicits are everywhere: a large-scale study of the use of Scala implicits in the wild. Proceedings of the ACM on Programming Languages. 2019, vol. 3, no.OOPSLA, pp. 1–28.] Krikava et al colleagues conduct large scale study on Implicits – Scala programming language feature. The analysis part of the underlying solution proves to be cumbersome, as the relational data model is not suitable for highly connected data. This thesis follows the underlying implementation with the aim to improve the analysis part, in terms of flexibility to support the creation of new queries. This thesis confronts these challenges with the use of graph database, which supports storing data with high amount of relationships. Particularly Neo4j graph database implementation with its Cypher query language is chosen for this purpose. This thesis shows, that the graph database with its query language offers a high-level interface for static code analysis.

Keywords static code analysis, Scala Implicits, Scala, graph database, Neo4j, Cypher

Contents

Introduction **1**

1 Background **3**
1.1 Scala . . . 3
1.1.1 Overview . . . 3
1.1.2 Type System . . . 5
1.1.3 Implicits . . . 7
1.2 Graph Databases . . . 12
1.2.1 Overview . . . 12
1.2.2 Neo4j . . . 14

2 Design **19**
2.1 Overview of the OOPSLA19 solution . . . 19
2.1.1 Introduction . . . 19
2.1.2 Analysis Pipeline . . . 20
2.1.3 Analysis Shortcomings . . . 22
2.1.4 Architecture . . . 22
2.2 Moving to Graph Representation . . . 23
2.2.1 Understanding the Implicits Data Model . . . 23
2.2.2 Graph Database model . . . 23
2.3 Graph Examples . . . 25
2.4 Example Queries . . . 27

3 Implementation **33**
3.1 Interface for Data Import . . . 33
3.2 Unique Nodes . . . 34
3.3 Import Phases . . . 34
3.4 Nodes Reference Cache vs Graph Searching . . . 35
3.5 Project Structure and Testing . . . 35

4 Assessment **37**
4.1 Running the Import and Viewing the Graph 37
4.2 Import Analysis 37
4.3 Query Analysis 39

Conclusion **41**

Bibliography **43**

A Acronyms **45**

B Contents of Enclosed Memory Device **47**

List of Figures

1.1 Scala's class hierarchy 6
1.2 Implicits are everywhere – implicits usage 8
1.3 Sub-graph example of the social network data model in graph db . 13
1.4 Example of labeled property graph model. Every node has at least one label and every relationship has its type and direction. Both the nodes and the relationships can have properties. 15
1.5 Node record in the node store file. 16
1.6 Relationship record in the relationship store file. 16

2.1 Analysis pipeline overview 20
2.2 Implicits data model 24
2.3 Instance of implicits data model 25
2.4 Graph model of extracted implicits in Neo4j database. Every label is represented by a node and every relationship by an edge. 29
2.5 Example of implicit conversion 1.5 in the created graph. Node labels are displayed displayed in bold font with the most important property value bellow. 30
2.6 Example of implicit parameter usage 1.6 in the created graph. Only a subgraph is shown. 31
2.7 Real example of type class usage 1.8 in the created graph. Only a subgraph is shown. 32

4.1 Dependency of total import time on input file size. 39

List of Tables

1.1 Neo4j in action experiment - Comparison of graph and relational DB 14

4.1 Neo4j authentication information 37
4.2 Time analysis of importing portions of the whole corpus. 39
4.3 Query times 40

List of Listings

1.1 Object construct in Scala - example 4
1.2 Function as first class citizen in Scala – example 5
1.3 Type inference in Scala - example 5
1.4 Operator names in Scala - example 6
1.5 Scala implicit conversion example 9
1.6 Scala implicit parameter example 9
1.7 Scala type class example 10
1.8 Scala type class jsonable 11
1.9 Example Cypher query - finding person who drives given car . 17
1.10 Example Cypher query - shortest path social network 18
2.1 Analysis Cypher - overall information 27
2.2 Analysis Cypher - number of implicit conversions callsites . . . 28
2.3 Analysis Cypher - implicit conversions to String 28
4.1 Manual for running the graph database import. 38
4.2 Analysis Cypher - implicit conversions to java Map 40

Introduction

Thanks to code-sharing websites such as GitHub or SourceForge, billions of lines of code are available to us at our fingerprints.

This is a great opportunity for programming language designers and researchers to understand how the different programming languages are used in the wild. As a result, languages can be evolved based on a knowledge of their actual use capturing possible unintended usage patterns. However such knowledge is hard to mine from the data. It is not always obvious what questions to ask. Also, it is not obvious how to ask and the data can quickly become overwhelming.

This thesis looks at large scale code analysis in the paper “Scala implicits are everywhere” [1]. The process of that analysis is the following:

1. download of all the projects
2. filtering duplicate projects [2]
3. filtering out not needed information and extract analysis specific information
4. extracting information for the specific question of the analysis and save it in the relational data format – CSV (Comma-separated values) format and
5. finally loading of this data and analysing it using R – data analysis tools.

The last two steps of this approach have some difficulties. One of them is the need to create specific extraction of data for every analysis question. This problem, in combination with the fact that this extraction takes a lot of time, makes this step tedious and exhaustive. Further, the final analysis part lot of low-level code and lacks a higher-level interface for the analysis.

This thesis tries to tackle the problems of the last two steps with the use of a graph database. All the information can be stored in the graph database

without the need to extract information specific to every question. This is mostly due to the fact that a graph database is able to naturally store all the relationships in the source code. Also, graph database query language can provide a higher-level interface for the analysis.

This thesis is a follows up on a work done by Krikava et al. in a paper called "Scala implicits are everywhere: a large-scale study of the use of Scala implicits in the wild" [1]. In this paper, they have analyzed the usage of the Scala feature called implicits. According to the paper, 7 280 Scala projects hosted on GitHub, corresponding to over 18.7 million lines of Scala code, were analyzed to understand the use of Scala implicits.

To accomplish such a task pipeline had to be built. First projects had to be downloaded from GitHub, then incompatible and duplicate projects had to be filtered out. After that projects were compiled and semanticdb[1] was created. Subsequently, data needed for implicit analysis were extracted with an analysis at the end. The last analysis task had been cumbersome - comma-separated values (CSV) files specifically for a given "question" had to be generated for the final analysis using R. The disadvantages of this process were the need to adjust the generation of CSV to fit the need of new analysis question and the complicated R code for connecting related entities.

This thesis implements the above-mentioned graph database approach for the given extracted data of Scala implicits. The goal of this thesis is to:

- Investigate the possible use of a graph database.
- Implement the import of the extracted implicits data to the Neo4j graph database.
- Re-implement part of the analysis using Neo4j's query language Cypher.
- Compare the two approaches to the analysis in terms of time efficiency and simplicity.

First, the thesis provides an understanding of Scala and graph databases in the Background chapter. Then the solution for using graph databases for the analysis is presented in the Design chapter. Next implementation decisions are discussed in the Implementation chapter.

[1]Semantic and syntactic information about the program, extracted by Scala Meta – Simple Build Tool (SBT) plugin

Chapter 1

Background

The following sections present the background and context for this thesis. It starts with a brief introduction to the Scala programming language that is necessary for understanding the rest of the thesis. Then it introduces graph databases.

1.1 Scala

The next few subsections are going to introduce the general-purpose programming language Scala, which is the subject of the analysis. Also, part of the pipeline is written in Scala, as well as the filling of the graph database, which is the main task of this thesis. The biggest emphasis is on the Scala implicits.

1.1.1 Overview

Scala is a modern general-purpose programming language, which has been designed by Martin Odersky with the first release in 2003 [3].

Before Scala[2], Martin Odersky developed simple language called Funnel. But the usage of this language was not as high, because the language was not practical for common users. The lesson was that, for language to be successful, a large base of standard libraries has to be available.

This led to creation of Scala, which is inter-operable with Java – this gives Scala user the possibility to use the huge amount of libraries, which have already been created for Java. Also Scala is primarily compiled to Java bytecode and runs on top of Java Virtual Machine and with the same speed as Java.

Scala is mixed-paradigm, statically typed language with a sophisticated type system, flexible and elegant syntax [4]. Scala is a modern programming

[2]Martin Odersky is also the creator of new javac compiler and Java Generics [3].

language, that combines the Object Oriented Programming (OOP) and the Functional Programming (FP) with static typing and flexible syntax.

Mixed Paradigm

Scala fully supports both the OOP and the FP paradigm. It might look like that these two paradigms might be incompatible, but they can supplement each other pretty well – In the OOP functions have side effects and mutable state is common, while in the FP variables are immutable and function do not have side side effects. Many times functional approach is more reliable, but solution to some problems might be much simpler with mutable state.

OOP In addition to the Java's support for the OOP, Scala includes traits [5], which is an elegant way of implementing classes with mix-in composition. Also in Scala everything is an object. Unlike Java Scala is absent of primitive types. Further Scala does not use the "static" keyword concept, which associates member functions or variables to a class rather than the instance. Instead Scala has a concept of singleton object (can be seen in Listing 1.1), which supports the use cases, where only one instance of a type is needed.

FP In Scala functions are the first class citizen, in a way that they are objects and can be passed to other functions or assigned to a variable the same way as other "ordinary" values (example can be seen in Listing 1.2). Further provide closures and pattern matching, features well known from the functional programming.

```
object Math {
  def sum(a: Int, b: Int): Int =
    a + b
}

print(Math.sum(2, 4))
```

Listing 1.1: Listing shows how Scala replaces the concept of static keyword in Java with special singleton object. The example shows creation of `Math` object. Functions of the `Math` objects then can be called without creating any instance.

To make the language cleaner and less verbose Scala uses type inference to allow programmer not to define type of variable (shown in Listing 1.3), where it is not needed. Scala uses local type inference meaning the types have to be specified for the function parameters and also for the function return type when the return value is the result of a recursive call. Scala also allows function names to include non-alphanumeric characters. This with the possibility to

```
def addX(x: Int): Int => Int = {
  a: Int => a + x
}
def add2: Int => Int = addX(2)

print(add2(3)) // Prints "5"
```

Listing 1.2: Listing shows example of how the functions are treated as first class citizens in Scala. It is possible to define function in a block of code. Also it is possible to pass the function as a parameter or have a function as a return value. Listing shows `addX` function that returns function based on its input. Calling this function `addX` creates new function in a block of code, that can be referenced by the identifier `add2`.

leave out a dot, when specifying member function allows methods to look like build-in operators, as in Listing 1.4.

```
val number = 5 % 3

print(number.getClass.getSimpleName) // prints "int"
```

Listing 1.3: Listing shows how can Scala infer type, when the type is not specified. The example shows the creation of the value `number` without assigning the type. The type is inferred by the compiler.

1.1.2 Type System

Scala has a sophisticated type system. One of the reasons is the combination of ideas from FP and OOP.

Scala is a statically typed language, but thanks to type inference often feels like of dynamically typed language. with all the compile time type checking, support of auto-completion mechanisms in IDEs, better performance and lesser amount of unit test, that is needed.

In Scala, everything is an object and every class inherits from class `Scala.Any` the root of a class hierarchy, which can be seen in Figure 1.1. Subclasses of the `scala.Any` are divided into two groups. In the first group are the value classes with a super class `scala.AnyVal` from, which all inherit. Every primitive type in Java corresponds to a value class. In the second group are all the reference classes and they all inherit from the `scala.AnyRef`, which corresponds to the `java.lang.Object` class. The instances of `scala.AnyRef` are implemented as a pointer to an object in the memory, while a value instance is most of the time stored directly as a value, without the need of a pointer [6].

```
class Number(val number : Int) {
  def +(other: Number): Number = {
    new Number(this.number + other.number)
  }
  override def toString: String = {
    number.toString
  }
}

val numberA = new Number(1)
val numberB = new Number(2)

print(numberA + numberB) // prints "3"
```

Listing 1.4: Listing shows that in Scala it is possible to use non-alphanumeric characters in function name. The example shows class `Number`, which defines function with operator like name – +. Also class methods with one or zero parameters can be called without the dot after the instance identifier and without the parentheses around the parameter. Class `Number` can then look like a build in class.

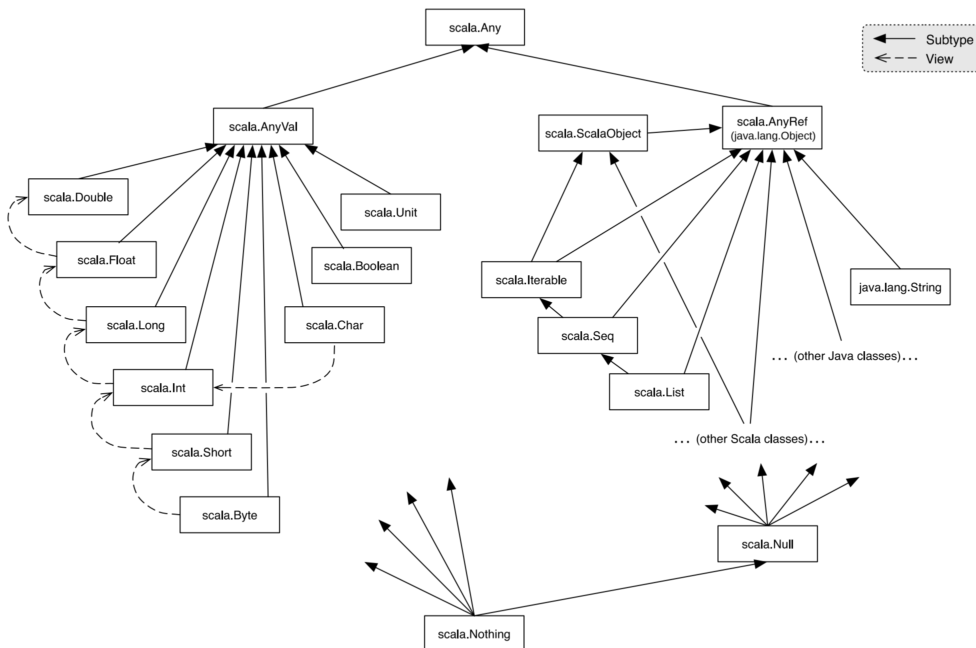

Figure 1.1: Scala's class hierarchy [6]

Figure 1.1 also shows class `scala.Unit`, which has the same role as a void in other languages. Further Figure 1.1 shows view relationships between value classes. These views correspond to the implicit type conversions, which can be used between specific value classes. These implicit type conversion methods are implemented in the `Predef` object, which is automatically imported by the compiler. These conversion of value types are only in the direction of a "wider" type. This means, that for example `Int` can be used everywhere, where `Long` or `Float` is needed.

As Scala has class `Any` which is a super type of all the classes, there is also a class `Nothing` which is a subclass of all the classes. This class consequently does not have any value and is used to fill a gap in the type hierarchy. One of the specific uses would be as a return type of a method, that always throws an exception. Or as a type of the last element in the List.

Also at the bottom of the Scala type hierarchy, there is a class `scala.Null` that is a subtype of every reference type, so it is not possible to assign the `null` value to any value type. This class corresponds to the Javas `null`.

1.1.3 Implicits

Scala implicits are implicit conversions and implicit parameters. Implicit conversions can let compiler without explicitly mentioning convert one type to another. With implicit parameters it is possible to leave out an argument and the compiler fills one for you. The Scala implicits, are not a unique Scala language feature – they also appears in other languages like Haskell or Coq, but in Scala this feature is commonly used. Figure 1.2 shows the usage of Scala implicits in the wild [1]. In the underlying large scale analysis 7 280 projects were analysed for the use of implicits. This analysis has found out that 78% of projects define the implicits themselves. Also that 98% of projects use Scala implicits.

Implicits did not come to Scala as some new feature, as implicit conversion was already built in the language from the beginning. Implicit conversion was built into Scala, as a solution to the late extension problem – Let there be `class C` and `trait T`, how to make class `C` extend the trait `T` after the class `C` is created, while it is not possible to change the class `C`? And that's, how implicit conversion was born. The implicit parameters were build into Scala not long after, as a simple solution for enabling type classes.

Implicit Conversion Given instance a of class `A` and class `B` and an defined implicit conversion from class `A` to class `B`, it is possible to use the instance `a` as if it was an instance of the class `B`, without explicitly calling conversion method. Listing 1.5 shows an example of such conversion. This whole "magic" works by compiler trying to find suitable implicit conversion type for an instance, whose type does not fit the way it has been used in the code. When the compiler finds one method for implicit

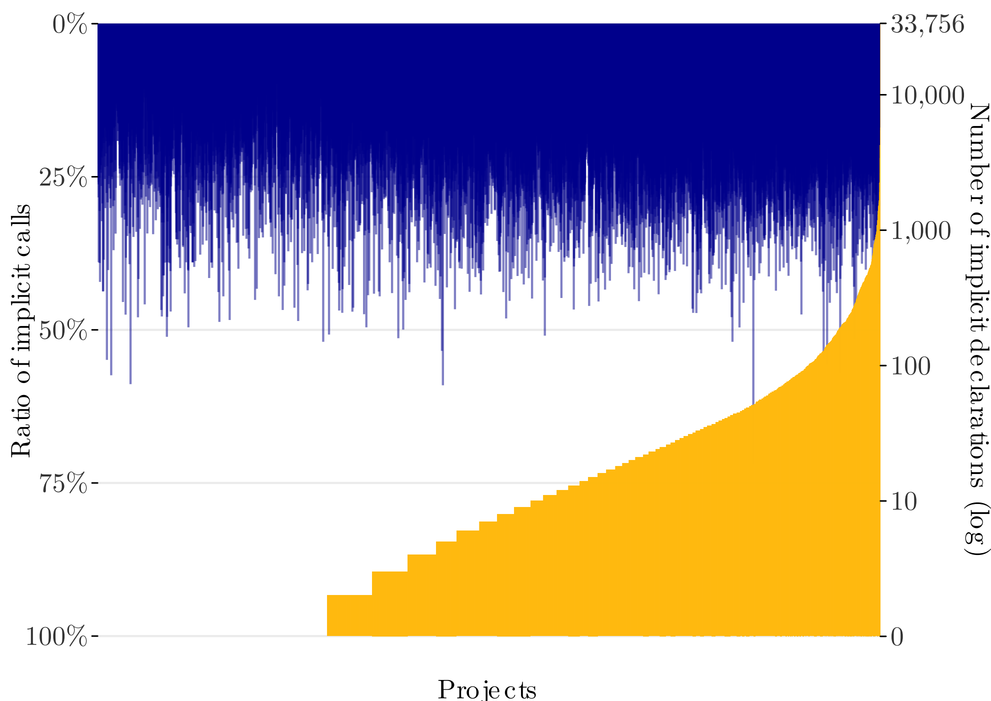

Figure 1.2: Figure shows large scale analysis of in Scala implicits usage [1].

conversion, which suits the needs, than the application of this method to that instance is added at the compile time.

Implicit Parameters With implicit parameters, it is possible to omit some arguments, when calling a method. If an argument is missing compiler will automatically look in the implicit scope for a defined implicit value, that would match the needed argument type. An example of implicit parameters is shown in the Listing 1.6.

One of the most important use of implicits is the elegant solution for extending external libraries [7]. Further implicits can be used to make the code cleaner, by eliminating boiler-plate code and letting compiler do the trivial tasks. They can also be used for elegant creation of embedded domain specific languages.

But these features come at a cost. Compiling code that uses implicits can dramatically increase the compilation time [8].

Patterns with implicits

There are many patterns with implicits [1]. In the following, we mention two popular ones to illustrate the expressive power of the implicit parameters.

```scala
class A
class B {
  def foo():Unit = {
    println("Hello!")
  }
}

implicit def AtoB(a: A): B = new B()

val a = new A
a.foo() // Prints: Hello!
```

Listing 1.5: Example of implicit conversion. Value a of class A is implicitly converted to class B using the AtoB implicit conversion method

```scala
def sum(number: Int)(implicit addedNumber: Int) =
  number + addedNumber

implicit val number: Int = 5

print(sum(1)) // prints "6"
```

Listing 1.6: Example of implicit parameter. First function sum is defined, which sums up two integers and the second integer parameter is defined as implicit. Then implicit value for integer is defined. Finally when calling the sum function without the second parameter, no error occurs and implicitly defined integer value is inserted by the compiler.

Implicit context is the simplest use of implicit parameters. By the term "context" it is meant an object holding variables, which are needed through many functions. Passing around this object can be bothersome and can be quite verbose. Using implicit parameters it is possible to pass this context implicitly [9].

Another very handy use of implicits is a combination with *Type classes*. Type classes, which add "ad-hoc" polymorphism to Scala, are a type-safe design pattern used for decreasing coupling between classes [10]. "Ad-hoc" polymorphism means that function is defined over a group of types and the function acts differently for every type. Compared to parametric polymorphism, where the function is defined for a range of types but acts the same way for every type [11].

Type classes are a group of types, where every class implements some contract, but not within the class itself. This contract is implemented separately without affecting the original implementation of a class. Type classes are a design pattern that can be implemented in other languages than Scala, yet

the implicit parameters make it convenient – Listing 1.7 shows an example of implementation in Scala without the use of implicits. First it is needed to define interface with a type argument, which defines method, acting upon instance of given type. In Listing 1.7 this is done by `trait Converter[T]`, which defines method `convert(t: T): String`. Then it is necessary to implement the abstract method for arbitrary number of classes – in our example this is represented by `converterA` and `converterB` implementations. Next, it is necessary to implement the polymorphic function, which takes type argument, an instance of that type and implementation of correspondent trait – `convert[T]` method. Finally, it is possible to call the `convert` method on an instance of a class, which is in the type class group.

```
class A
class B
// Type class declaration
trait Converter[T] {
  def convert(t: T): String
}
// Instance of the type class
val converterA: Converter[A] = new Converter[A] {
  def convert(t: A): String = "Class A"
}
// Instance of the type class
val converterB: Converter[B] = new Converter[B] {
  def convert(t: B): String = "Class B"
}
// Function which uses the type class
// Type T is extended with a new method convert
// without changing the implementation of type T
def convert[T](instance: T, converter: Converter[T]): String =
  converter.convert(instance)

val a = new A

print(convert(a, converterA)) // prints "Class A"
```

Listing 1.7: Example of type class implementation in Scala

In Scala type classes can be implemented elegantly than in Listing 1.7, with the power of implicit parameters it is possible to omit the trait implementation argument (`converterA`) from the function call (`convert`) and the compiler inserts the needed parameter for us.

Listing 1.8 shows a real example, how type classes can be beneficial. When implementing conversion to JSON (JavaScript Object Notation) format we do

not want to or we cannot have the classes, we want to convert to JSON, implement a certain interface. This could be done using reflection, but type classes with the use of implicit parameters offer another simple solution. Listing 1.8 shows definition of `JSONable` trait, with convert method, which converts a type to its JSON representation. Next implementation of the `JSONable` trait is created for types `Int` and `List[T]`. The implementation of `List` abstract data type deserves attention, as it enables conversion of all the Lists without the need of implementing every type argument of the `List`. The compiler is able to derive new type, in our case new type class of type `Int` is created.

```scala
// Type class declaration
trait JSONable[T] {
  def convert(t: T):String
}
// Instance of the type class for type Int
implicit val JSONableInt: JSONable[Int] = new JSONable[Int] {
  override def convert(number: Int): String = number.toString
}
// This function derives new type classes during compile time
// The type class argument for List's type argument
// is filled implicitly
implicit def JSONableList[T](implicit tJsonable: JSONable[T]):
  JSONable[List[T]] =
  new JSONable[List[T]] {
    def convert(list: List[T]): String = {
      list.map(t => tJsonable.convert(t)).mkString("[", ",", "]")
    }
  }
// Method which uses the type class
def json[T](instance: T)(implicit converter: JSONable[T]): String =
  converter.convert(instance)

val numbers = List(2, 5, 6, 9)
// The second argument JSONable[List[Int]]
// is implicitly added during the compile time
print(json(numbers)) // prints "[2,5,6,9]"
```

Listing 1.8: Real example of type class usage in combination with implicit parameters.

1.2 Graph Databases

This section introduces graph databases, especially – Neo4j, which has been used in this thesis including Cypher, the Neo4j specific query language.

1.2.1 Overview

Graph databases are based on the graphs from the graph theory. Graphs are set of vertices (also called nodes), which are connected through edges. When representing data through a graph, entities are represented by nodes and the relationships between them with the edges. This very flexible data structure enables the modeling of many real-world data. It should be noted that everything could be modeled as a graph, but using a graph database is not suited for data, that can be modeled using simpler data structures [12].

The world of graph technologies could be divided into graph databases used for online transactional graph persistence and graph compute engines used for offline graph analytics. But this thesis will focus only on the graph databases.

Graph database management systems, in general, are databases that enable operations – Create, Read, Update and Delete methods upon a graph data model. Graph databases can be divided by the way how the data is stored underneath. The first group of databases are those which store graphs natively as graphs. But some save the graph into other databases like a relational database or a wide-column store database.

Use Case for the Graph Database

One of the reasons, why choosing a graph database, is the speed when doing a lot of entity "joining". With relational databases join intensive queries do not scale well with the amount of data, on the other hand, graph databases handle these kinds of queries with constant performance with growing data. That is mostly because the search in graph databases can be localized to a minor subset of the graph. Than the query time is proportional to the size of only the part of the graph, that is needed to be traversed, to answer the query. And not to the size of the whole graph.

To prove the point of the graph databases Vukotic and others in the book "Neo4j in Action" [13] done an experiment comparing the Neo4j graph database with a relational database. This experiment shows that for certain connected data and specific query the graph database is many times faster than a relational database.

In this experiment, they model a social-network containing people and friendship relationships between them. They analyze the time needed to find friends of friends of some specific person. The data set contains 1 000 000 people with each person having approximately 50 friends. The graph representa-

tion of the data set can be seen in Figure 1.3, which shows a sub-graph example of such a data set. The analysis concludes, that the relational database is not suited for deeper recursive queries.

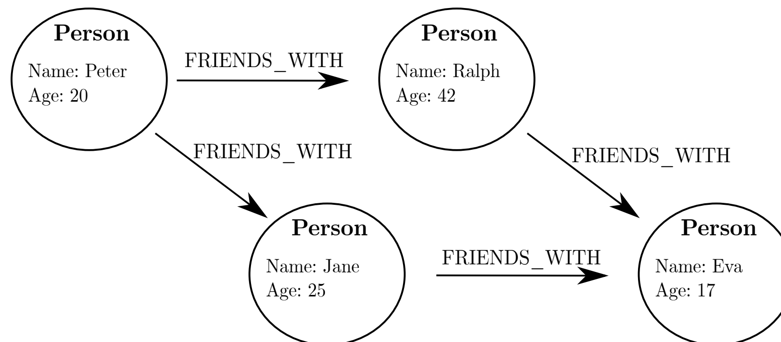

Figure 1.3: Example, how the social network data are represented in the graph database

Table 1.1 shows results of the analysis for different depths of the query – depth 2 means finding friends of friends of a specific person. The last column shows how many results does the query return.

For depth 2 the query returns 2 500 records and the difference between graph database and relational database is negligible. At depth 3 with 110 000 returned records the difference starts to show up. It takes around 30 s for the relational database to return the result, while the graph database manages this task in less than 0.2 second. For depth 4 and 600 000 returned results, the relational database, with 1535 s, cannot return the result in a reasonable time, for most of the possible applications. On the other hand, the Neo4j database with 1.4 s execution time still manages the query in a reasonable time.

At the depth 5 and 800 000 returned results, the relational database is unable to give results in time to be able to measure it. In contrast, in the Neo4j database, the query takes around 2 seconds. Which can still be a reasonable time for an online system. While the graph database was only traversing the nodes relevant to the query, the relational database needs to generate the Cartesian product of the whole `people` table for each level of depth. This means that the resulting set has $1\,000\,000^5 = 10^{30}$ rows, while only 800 000 rows are preserved.

The purpose of this demonstration was to show, that for certain data and certain questions relational databases are not suited, while graph databases excel. A more common question like, what is the average age of the social network user, would result in the relational database being much faster with the increasing number of records. In the relational database, this query would result in a fast table scan, because the rows are stored one by one in a table. In the Neo4j graph database, the same query would mean accessing every

node that is labeled as a user, for that every node accessing its linked list of properties in a different file and in this list looking for a property, which can be on an arbitrary position in this linked list.

Depth	Relational DB time (s)	Graph DB time (s)	Returned count
2	0.016	0.01	2500
3	30.267	0.168	110 000
4	1 543.505	1.359	600 000
5	Not finished	2.132	800 000

Table 1.1: Results of an experiment comparing graph and relational database [13]. Table shows execution times of graph and relation database for retrieving friends of friends in social network data model with 1 000 000 people, for different depths. Every person has approximately 50 friends.

1.2.2 Neo4j

Neo4j is one of the native graph database implementations, meaning the underlying data are stored as a graph. Another characteristic of Neo4j is, that it is based on a labeled property graph model and that it is a schema-less database.

The labeled property graph model is a model based on the directed graph. This graph consists of nodes and relationships. Every node is marked by at least one label and every relationship has exactly one relationship type. These directional relationships connect nodes with one starting node and one end node. Also, every node or relationship can have an arbitrary number of properties and its values. The ability of relationships to have properties makes them first-class citizens in the graph database. This is useful for extending the semantic information.

Figure 1.4 shows a simple example of the labeled property graph model with a news recommendation system data model. The model is very easy to read – there is a person named Peter of age 20, who likes the topic of Cars since 23. 8. 2019. So there are two nodes with labels `Person` and `Topic` with one relationship of type `Likes` from the node with label `Person` to the node with label `Topic`. Note, that not only the nodes have an arbitrary number of properties, but even the relationship can have properties.

Further, advantage of using a this graph database model is that it is not necessary to get the data model right for all business requirements ahead of time. It is possible to add new kinds of relationships, labels, and nodes to an already existing graph without affecting the already designed queries.

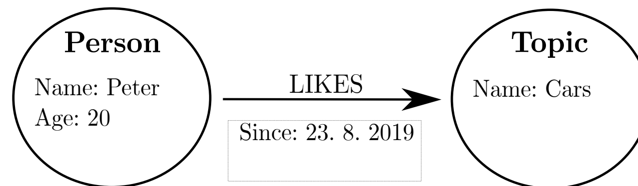

Figure 1.4: Example of labeled property graph model. Every node has at least one label and every relationship has its type and direction. Both the nodes and the relationships can have properties.

Architecture

In the Neo4j database every "database entity" is stored in its individual file – there are separate files for storing nodes, relationships, properties and labels [12]. This separation, especially of the nodes from their properties, helps to speed up traversals through the graph. The two cornerstones of the graph database, the nodes, and relationships, are stored as fixed sized records in a separate array.

As mentioned each node is stored in a `node store` file, where every record has a fixed size of 15 bytes. This fixed-size array format enables the lookup of an arbitrary node, with given `id`, in O(1).

As shown in Figure 1.5, the node can be as small as 15 bytes because it stores pointers to other entities of the graph.

0. byte First byte gives information, whether this record in the array is taken by some node, or new node can be created at this place.

1. – 4. bytes store a pointer to the linked list of relationships. This is essentially the id of the first node relationship.

5. – 8. bytes store the nodes first property id – a reference to the array of properties.

9. – 13. bytes Next five bytes store the information about the node's labels.

14. byte Last byte is reserved for flags. For example whether the node is densely connected [12].

The same concept applies to storing the relationships – each relationship record in relationship array has a size of 34 bytes. As shown in Figure 1.6 the relationship record stores the starting node id, the end node id, pointer to the relationship type, pointer to the previous and the next relationship from the list of relationships of both start and end node. The last byte indicates whether the relationship is the first in the relationship double linked list. Other data fields in the record are similar to the node record.

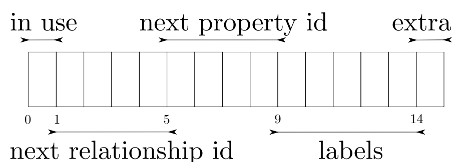

Figure 1.5: Node record in the node store file.

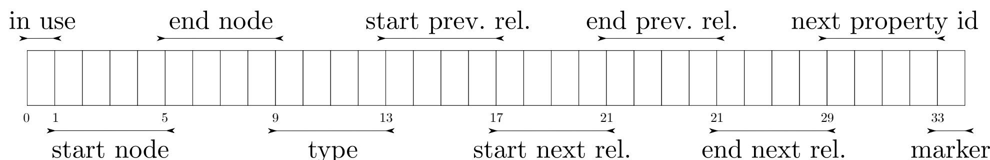

Figure 1.6: Relationship record in the relationship store file.

This consequently means, that traversing the graph is jumping in these fixed-sized node and relationship arrays from node to relationship and then back to the node array. Also, the cost of adding a relationship in Neo4j is higher than in the relational database. In Neo4j, the relationship needs to be added to both the nodes and to the relationship. Unlike in relational databases, where the relationship is stored only by one record in a table.

Cypher

Neo4j's query language of choice is the declarative language called Cypher. Although Cypher is currently specific to Neo4j, Cypher is open source with its specification published in the openCypher project [14].

This language allows both the retrieval of data and updating the graph. As seen in the example in Listing 1.9, Cypher is intuitive and its queries are able to be read even without the knowledge of the Cypher syntax. The query in Listing 1.9 finds all the nodes representing a person, who drive a car with the given registration number and returns the names of those persons.

Cypher uses ASCII (American Standard Code for Information Interchange) art for pattern matching to make the queries more readable. Every node is represented by a circle drawn by two round parentheses – `()`. In the example both the `PERSON` node and the `CAR` node are covered by the parenthesis. The word in the node after the colon is the node's label. And a list of node properties can be specified in curly brackets – `{}`. Also relationships are depicted as arrows – `-->`. In the example the `DRIVES` relationship points from the node it originates to the destination node. The specific information about the relationship is described in the square brackets, in the middle of the arrow – `-[]->`. The relationship type is specified the same way as node's label after the colon and the list of properties is encapsulated in the curly brackets.

```
match
(person:PERSON)-[:DRIVES]->(:CAR {registration_number: "1M36868"})
return person.name
```

Listing 1.9: Example Cypher query. The query finds all the nodes representing person, who drives a car with the given registration number and returns the names of those persons.

Further, every node can be referenced by a variable that is given before the colon and the entity type. In the example the node labeled as `PERSON` can be referenced by the variable `person`. The `match` keyword tells Cyper engine to find all the nodes, that match the given pattern (insert the references into the variables). The function of `match` is comparable to function of `select` in SQL. The result of such a query can then be extracted with the keyword `return`.

With Cypher one can use different functions and algorithms. One of those is a build-in algorithm for finding the shortest path in a graph. The shortest path algorithm comes from the Neo4j Graph Data Science library, which contains a large number of graph algorithms. Unfortunately, at the time of writing this thesis, many are not labeled as production-ready quality. Other examples of these algorithms are K-1 coloring, Page rank, or Strongly connected components.

Listing 1.10 shows the use of shortest path algorithm, on the data social network data model already mentioned in Section 1.2.1 (with example of the graph data in Figure 1.3). In this query, we are looking for all the shortest friendship paths from a user called Peter to a user called Adam. We want to know through which of their friends and other people they could know each other. First, it is necessary to find the nodes which represent the users Adam and Peter. That is done by simple pattern match with specifying the Person's property – name. After extracting the Peters and Adams nodes, algorithm `shortestPath` is applied between these two nodes using only the `-[:FRIENDS_WITH]-` relationship. In the end, all the shortest paths are returned in the form of a sequence of nodes. The power of such query in graph database compared to other database types, is in its simplicity and the speed. Only the neighbor nodes have to be traversed through the edges and not the whole database.

```
match
    (peter: Person {name: "Peter"}),
    (adam: Person {name: "Adam"}),
    path = shortestPath((peter)-[*:FRIENDS_WITH]-(adam))
return path
```

Listing 1.10: Example Cypher query for the social network data model 1.3. The query finds the shortest path of friends between the person with name Peter and the person with name Adam

The example in Listing 1.10 might look like an artificial query, but this information could be invaluable for some crime investigators to understand how these two persons might have met. Such queries show the power of Neo4j database and are reasonable use cases, why would one want to use graph database over another database type.

Chapter **2**

Design

This chapter starts with an introduction to the pipeline analysis solution [1] this thesis tries to improve. Then it presents shortcomings that emerge from the underlying implementation. Finally, it shows an alternative solution together with a few examples.

2.1 Overview of the OOPSLA19 solution

This section describes the solution of the implicits analysis done in Krikava's paper [1]. It is important to understand the solution, as this thesis proposes an alternative solution for the analysis part.

2.1.1 Introduction

The analysis of the Scala implicits is done using an analysis pipeline. It has the following steps (cf. Figure 2.1):

1. First projects are downloaded from the GitHub.
2. Next project metadata are gathered – most importantly identify the build system each project uses.
3. After that incompatible projects are filtered out.
4. Next duplicate projects are eliminated by the use of DéjàVu tool [2].
5. Afterwards projects are compiled and semantic info extracted while discarding the projects that failed to compile.
6. Finally, acquired data are analyzed.

Only the last analysis step is specific to the implicit analysis. This means, that the pipeline could be reused for other Scala analysis [1].

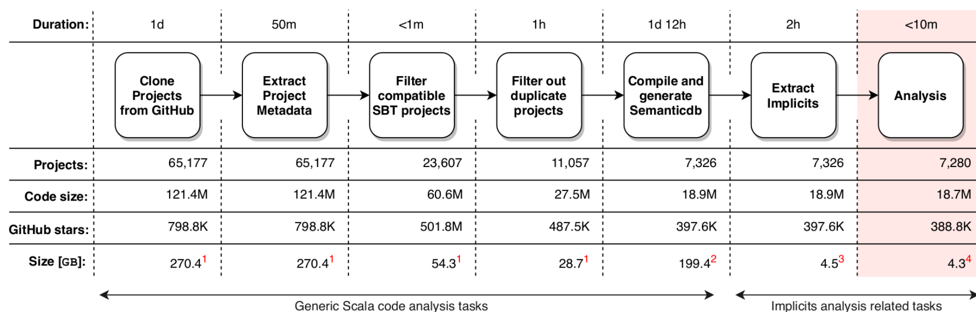

Figure 2.1: Pipeline with duration of each step and size of the corpus and size of related files at the given stage [1].

2.1.2 Analysis Pipeline

Cloning projects form GitHub

To be able to download all the Scala GitHub projects, GHTorrent project has been used [15]. GHTorrent is an API for accessing realtime GitHub metadata – the events (for example – commits, forks, and pushes) and entities (for example – users, repositories, organizations, and issues). This enables finding all the URLs for all public Scala GitHub projects, which are then cloned. Further specified numbers correspond to the downloading of projects made between January and March 2019.

Extracting metadata

To be able to interconnect all the Scala projects, metadata had to be extracted. Metadata is general information about the project modules, project dependencies, and source paths. Metadata are needed in the process of building the projects and in linking the same sources in the analysis step.

Filtering compatible projects

Some projects that were not compatible with the analysis pipeline had to be removed. Most of the time the incompatibility is due to early versions of Scala or SBT build tool. Projects with other build systems like Maven or Gradle are not supported by the pipeline. This meant that about half of the 65 177 original projects were discarded.

Elimination of duplicate projects

Elimination of duplicate projects can be a difficult task. Finding out the original project could be much more difficult, that one could imagine. Even

after discarding all forks of a project, some unofficial forks might appear. These unofficial forks are the result of someone downloading original project files and uploading them to GitHub once again. For example, 102 copies of the biggest Scala project, Spark, were detected. Keeping all the copies would result in having 37.6% of the whole corpus identical [1]. The following rules were used to decide, whether the project should be kept:

- Project must have more than one commit.
- Project must have been active for more than two months.
- Project must exist in Scaladex [16] or have less than 75% of file-level duplication or must have obtained more than 5 GitHub stars.
- Project must be in Scaladex or have less than 80% duplication or more than 500 stars on GitHub.

These conditions for keeping a project were chosen empirically to retain all the large popular Scala projects without duplicates. 12 550 projects were eliminated in this step, while losing fewer than 2.8% stars.

Further, from the resulting 11 057 projects, 7 326 were possible to compile. The rest of the projects failed to compile.

Compilation and Semanticdb generation

It is not possible to observe implicit call sites, without the compilation step, as they are implicitly inserted by the compiler. So it would be impossible to do some lightweight solution as regular expression matching over AST nodes. For that purpose, an SBT plugin called ScalaMeta [17] has been used, which does exactly, what is needed to do the static analysis of Scala programs.

For each compilation unit, this plugin produces Sematicdb file, which stores syntactic and semantic information of the given compilation unit including a list of defined and referenced symbols, synthetic call sites, and parameters that are injected at the compile time. The Semanticdb defines the data model of the semantic information by the protocol buffer model [18]. Google Protocol Buffer is a serialization format, so this model both defines the data structure and is also used to serialize the data to a binary file.

This is the last step, which is general for the static analysis of Scala programs. The pipeline until this step can be reused for any different type of analysis of Scala programs. After the semantic information extraction, the data are adjusted to suit the needs of implicit analysis and stored into `implicits.bin` file.

It should be noted, that this step is quite resource-intensive. To create a Semanticdb file, projects need to be built and the Scala compiler is approximately an order of magnitude slower than Java compiler. In addition to that, the Semanticdb compiler plugin adds additional overhead.

Analysis

The final analysis step is done using R. `Implicits.bin` file could have been read using R, but it was more convenient to first extract specific CSV files, from the protocol buffer format, as R works well with data in relational form. This is the part of the pipeline, which is being improved. The next section describes the most important details of the Analysis step and its shortcomings.

2.1.3 Analysis Shortcomings

The main problem of the analysis step is that is it rather labor-intensive. The actual analysis is done in R by exploring data stored in CSV files. These files contain data extracted from the `implicits.bin` file. This means, that every time, there is a need for any additional data, one has to create a new CSV extractor in Scala, compile it and run it over the `implicits.bin` file. Often, the right information (or format of the information) is not found right at the beginning which leads to the repetition of this process. This is both resource-intensive and error-prone. Further, the creation of queries, which is done using R, is cumbersome. The different entities need to be merged at the run time by their keys. Also, there is a lot of data duplication – every module stores information about declarations from external dependencies, and these data are not shared between all the modules. For example, when all the modules would use type String from the Scala standard library, then the number of String declaration would be the same as the number of analyses modules.

2.1.4 Architecture

This subsection gives a slight insight into how the pipeline is structured. There is a GitHub repository [19] containing the OOPSLA solution with instructions, on how to build and run the pipeline.

It is implemented in Scala, R, and few auxiliary shell scripts. The orchestration of the pipeline is done by GNU make. To make running the pipeline convenient and to avoid resolving all the dependencies, the whole pipeline is run in a docker container. This makes it easy for anyone to experiment with the pipeline. The first outer layer of the solution are the make files. The whole process starts when the main corpus make file is run. This main make executes subroutines, which correspond to the stages of the pipeline. Each stage then consists of individual tasks. These tasks then correspond to calling Scala script (which consequently calls some build Scala code), R scripts, SBT plugin, or some other binary. Many of these tasks are run in parallel using the GNU parallel [20].

2.2 Moving to Graph Representation

The aim of this thesis is to address the drawbacks of the OOPSLA solution mentioned in Section 2.1.3. This section proposes moving the data into a graph database to solve the mentioned shortcomings. Concretely, we have chosen the Neo4j graph database and the Cypher query language.

2.2.1 Understanding the Implicits Data Model

The new solution uses the implicits data (the implicits.bin file), that have been already extracted from the projects. The implicits data are stored in a binary file of Google Protocol Buffer format. Figure 2.2 shows the underlying data model of the implicts using an entity relational notation.

The project is composed of modules (this is the way SBT organizes projects). A module is a container for source code. In our case, the module contains declarations and call sites. Every module consists of all the declarations, that have been declared in that particular module and all the external declarations that have been used in the module. All the declarations without any relation to the implicit analysis have been discarded – only relevant declarations have been kept. Every module also consists of implicit callsites, these are all the callsites, which represent all the actions of the compiler regarding the implicits. Further, every Module includes all the paths to resolve the origin of every declaration. Furthermore, every declaration contains information about its location, signature, and annotations. The signature can be one-off different type depending on what does the declaration represents – value, method, type or class.

Figure 2.3 shows a simplified example of the implicits data model seen in Figure 2.2. This example presents declaration `implicit def f(x: A): B` in a module M and a project P. The `implicit` modifier is encoded by the sixth bit of the declaration's `properties` property.

2.2.2 Graph Database model

Figure 2.4 shows the graph database model. Neo4j database does not have a schema, which validates the inserted data, instead, this is a representation of all the labels in the graph with all their corresponding relationships in the graph. Every label is represented by a node and every relationship by an edge. If and only if there is a relationship between nodes with two labels (not necessarily different), then this relationship connecting those labels is also in the presented schema. The node properties have been excluded. Also, this model does not give any information about the cardinality of the relationships.

Some labels are a subset of the other label, meaning if the node has one label then the node also has to have another label. For example when there is a node which has the label `ImplicitParameter` then it also has the `Parameter`

Figure 2.2: Data model of extracted implicit data from a Scala project. This data model is the starting point for designing the graph database model.

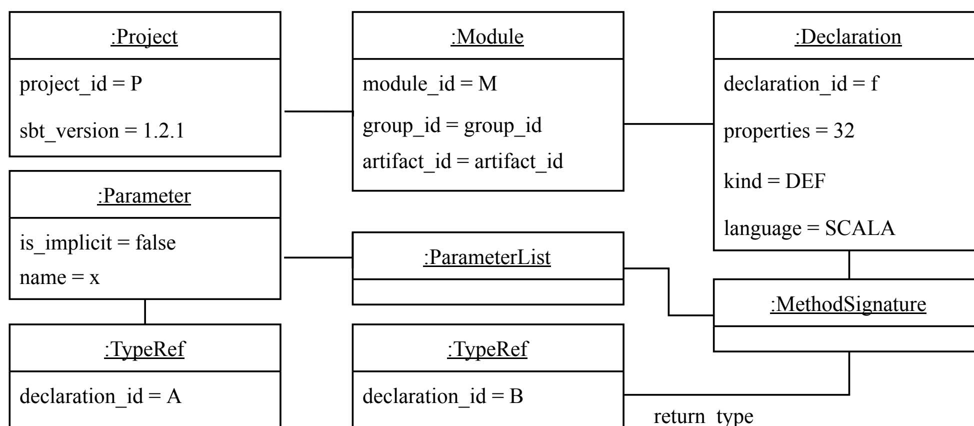

Figure 2.3: Simplified example of implicits protocol-buffer data model seen in Figure 2.2

label. These subset labels are displayed next to the corresponding node with the same font as the label. The second occurrence of this convention is by the `Declaration` label. Every node with label `ImplicitDeclaration` also has `Declaration` label and if a node is labeled as `ImplicitConversion` then it is also `ImplicitDeclaration` as well as `Declaration`. It should also be noted that only declaration nodes labeled as `ImplicitConversion` have the relationships `CONVERSION_FROM` and `CONVERSION_TO`. This is a convenient way how to add subtype to the graph. Further another way of dealing with polymorphism can be seen on the signature of a declaration. The `Signature` node is only an auxiliary node, which serves as junction and the signature is defined by the relationships to the other nodes. The type of the signature can be extracted from the `SignatureType` node.

2.3 Graph Examples

Figure 2.5 shows example of an implicit conversion from Listing 1.5 imported to the graph. For simplicity, we only include the most important nodes (this full graph contains 20 nodes and 32 edges).

First Figure 2.5 includes project node with `projectId` property. Project node is a source vertex, as there are only edges going from this node and no edge going to this node. This project node defines one module called `Conversion`. This module then declares three declarations (`AtoB`, `A` and `B`) and contains one implicit callsite (`AtoB(a)`). It can be seen that one of the declarations, `AtoB`, is `ImplicitDeclaration` as well as `ImplicitConversion` (these labels are next to the node and not inside). From this `AtoB` declaration originate relationships to `Signature` and also two `TypeReference` nodes,

which tell from and to which type does the conversion happen. It can be seen that the `Signature` node connects to the node representing the return type and the type of its one parameter. Notice, that the `TypeReference` nodes for the parameter type and the return type correspond to the conversion from and to `TypeReference` nodes. Further the implicit callsite node `AtoB(a)` represents a call which has been implicitly inserted by the compiler, when the conversion from the type `A` to the type `B` was needed (for calling the method `foo()` declared by class `B` on the instance a of class `A`).

Figure 2.6 show an example which has been presented in the Implicits Section 1.6 imported to the graph. Only a subgraph is displayed. The module defines two declarations – first declaration is a value with identifier name `number` of type `Int`. The second declaration is method `sum` with return type of `Int` and which has two parameter lists. Each list has one parameter of type `Int`. The second parameter Node with name `addedNumber` is also an ImplicitParameter. Finally, the module has one callsite which represents the compiler filling the implicit argument value to the second parameter of `sum` method, when the second parameter was not specified in 1.6.

At last Figure 2.7 displays real use of typeclasses presented in Listing 1.8 in the graph. `Module` and other nodes were omitted from the image. Figure 2.7 contains a lot of nodes and can be hard to read at first, but the graph can be read the same way as how Cyper does it – First looking up starting node which can be done faster when the most important nodes with the same label are colored the same. And then traversing the edges of the correspondent node.

As can be found in the 1.8, there is declaration of the `JSONable` typeclass in the graph in the blue declaration node. Then there is a `json` declaration which corresponds to the function, which uses the `JSONable` typeclass. This function has a type parameter `T`, first parameter is of this type `T` and the second implicit parameter `converter` is of typeclass type with type parameter `T`. Then there are two implicit declarations which correspond to the classes of the type class `JSONable - JSONableInt` and `JSONableList`. These declarations can be implicitly inserted to the the suitable implicit parameters. The `JSONableInt` is a simple value which is of type `JSONable[Int]`. And the The `JSONableList` is a function with the type parameter `T`. Further has one implicit parameter `tJsonable` of type `JSONable[T]`.

Finally `json` function is used on a list of integers, as in Listing 1.8. The second implicitly inserted parameter by the compiler is represented by the `json[List[Int]] callsite` node. The compiler first needs to create the `JSONable[List[Int]]` type class, by calling the `JSONableList[T]` method. This call requires implicit argument of type `JSONable[Int]`, which is also inserted by the compiler. This insertion is represented in the graph by the `JSONableList Callsite` node. The parent relationship between these two `Callsite` nodes gives an information that the child callsite had to be created first to create the parent callsite.

2.4 Example Queries

This section is going to show and explain a few queries in the Cypher language. These queries are part of the implicits analysis, but this section won't go into the detail of the analysis nor into the results of such analysis.

Listing 2.1 shows Cyper query which for each project calculates the number of implicit call sites and number of implicit declarations. The numbers have to be counted in two `match` statements. These two match statements can be joined together with the use of with keyword, which enables transferring results of the first match to the second match statement. Number of nodes is counted by the `count` function.

```
match
    (p: Project)-[:HAS_MODULE]->(m:Module)
    -[:DECLARES]->(imp_decl:ImplicitDeclaration)
with
    p as project, count(imp_decl) as imp_decl_count

match
    (project)-[:HAS_MODULE]->(m:Module)
    -[:HAS_CALLSITE]->(imp_cs: CallSite)
with
    project, imp_decl_count, count(imp_cs) as imp_callsite_count

return project.projectId, imp_decl_count, imp_callsite_count
```

Listing 2.1: Cypher query for getting table with project and its corresponding number of implicit declarations and implicit callsites

Listing 2.2 shows a query calculating the number of call sites, which are both implicit conversions, and the type of conversion output is defined in Java. The first match expression finds all the declarations, which are implicit conversions and their conversion result type is Java in language. The second match finds all call sites which are declared by declarations found in the first query. At last, the return expression counts and returns the number of occurrences of the call sites from the previous match expression.

It should be noted, that some node labels are missing in the query. This is due to the fact, that the information needed for our exact query is already specified and any further specified labels or attributes are dispensable. Even worse, by specifying some more information could make the query slower, as the graph engine would filter nodes, where there is nothing to filter. This is due to the fact, that the graph database does not know the data model and cannot make the same assumptions about the data as someone who knows the exact structure. But these optimizations could also lead to incorrect query

results. This could happen due to due to some change in model, without editing the query.

```
match
    (conversionDeclaration: ImplicitConversion)
    -[:CONVERSION_TO]->()-[:TYPEREF_DECLARATION]-()
    -[:IN_LANGUAGE]->(:Language {name:"JAVA"})
match
    (callsite)-[:DECLARED_BY]-(conversionDeclaration)

return count(callsite)
```

Listing 2.2: Cypher query for getting the number of implicit conversion callsites, that convert to type defined in Java

Listing 2.3 shows a query for obtaining all the call sites, which are implicit conversions to the Scala String. First match expression gets all the declarations, that are implicit conversions to a type String. In the second match statement, call sites associated with this conversion are located. Note that in the first match expression, it is not necessary to give a clue to the graph database engine, that it is an implicit conversion, because it is clear that if declaration already has an edge called `CONVERSION_TO`, that it is an implicit conversion. But by specifying the `ImplicitConversion` label, it is possible to make the query faster, because of how does the searching works. First, it is needed to get the starting nodes from which the graph is traversed. This is done by the query planner, but can also be instructed manually. The goal is to get the least amount of nodes and avoid pointless iteration over nodes. So it might be beneficial to start the query by getting all the `ImplicitConversion` nodes instead of starting from the at the String `TypeReference` node, assuming that this string node is going to have a larger amount of connections, then there is of `ImplicitConversion` declarations. In the end, if the starting nodes are not specified explicitly, the planner chooses the more preferable starting point based on the available data.

```
match
    (:TypeReference {typeExpression: "scala/Predef.String#"})
    -[:CONVERSION_TO]-(declaration:ImplicitConversion)
match
    (declaration)-[:DECLARED_BY]-(callsite)
return callsite
```

Listing 2.3: Cypher query for getting the of implicit conversion callsites, where the result of the conversion is String

Figure 2.4: Graph model of extracted implicits in Neo4j database. Every label is represented by a node and every relationship by an edge.

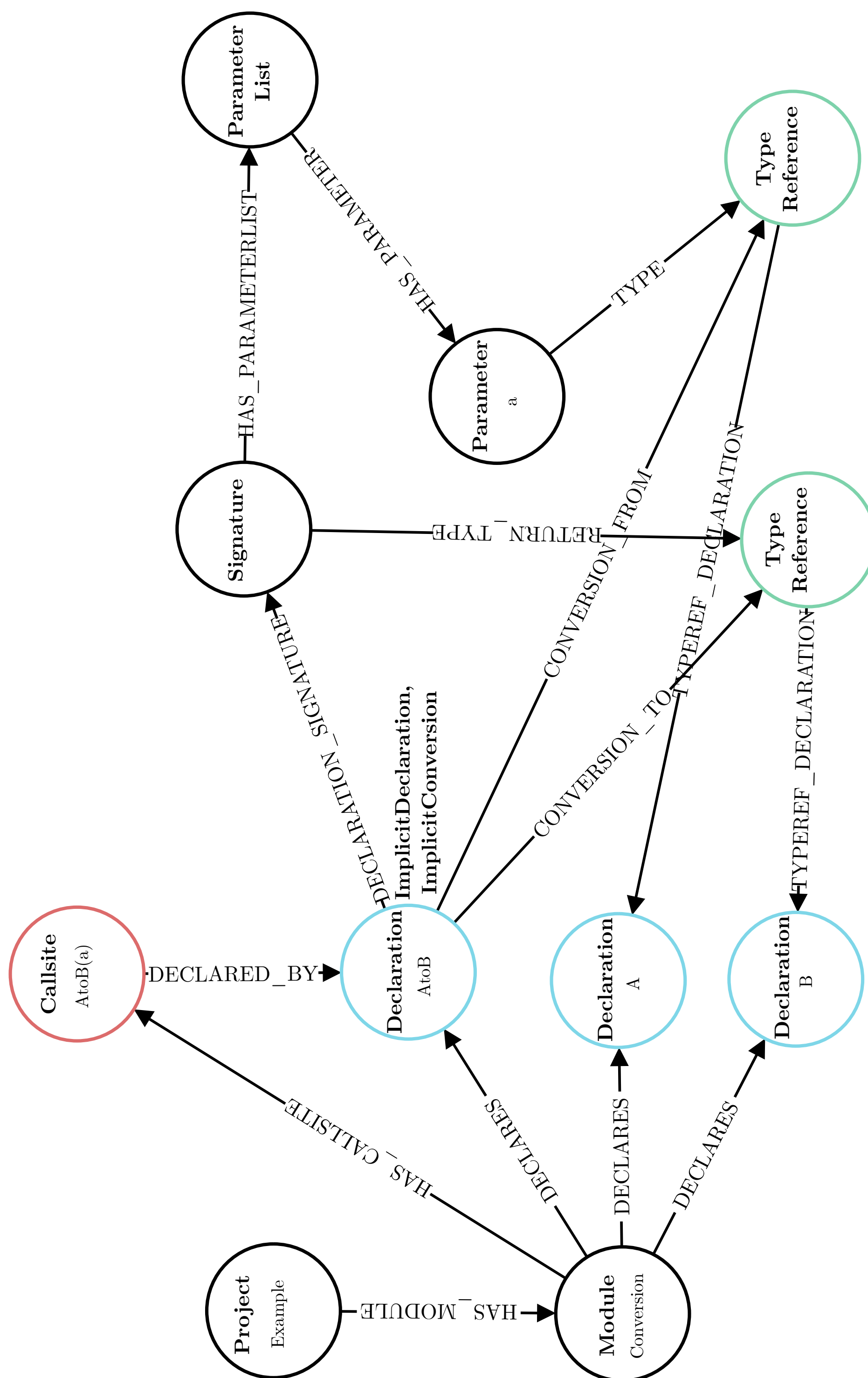

Figure 2.5: Example of implicit conversion 1.5 in the created graph. Node labels are displayed displayed in bold font with the most important property value bellow.

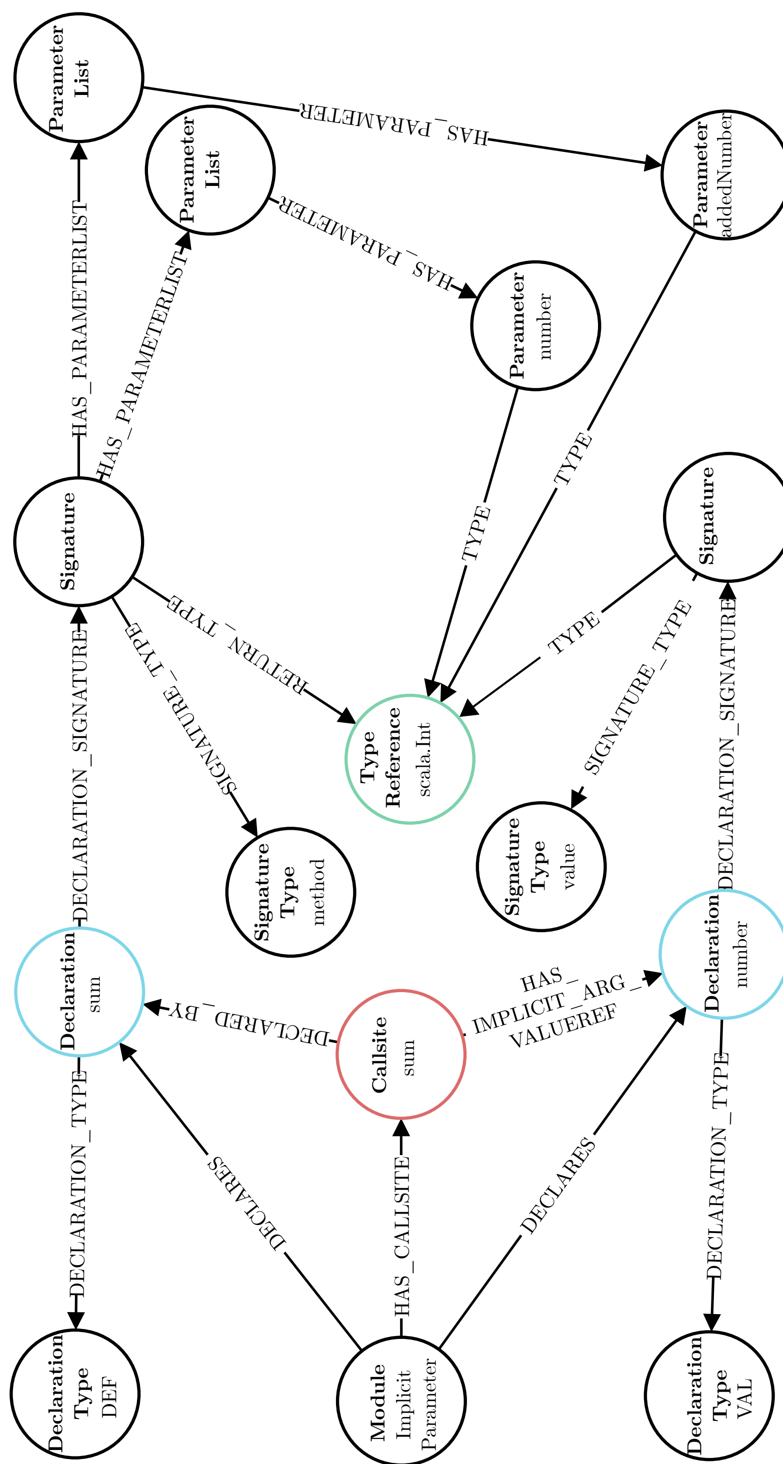

Figure 2.6: Example of implicit parameter usage 1.6 in the created graph. Only a subgraph is shown.

Figure 2.7: Real example of type class usage 1.8 in the created graph. Only a subgraph is shown.

Chapter 3

Implementation

The implementation part of the thesis deals with importing the extracted implicits into the Neo4J database following the schema from the previous chapter 2.4. Details and decisions concerning the data import are discussed in this chapter.

3.1 Interface for Data Import

There are two possible ways how to programmatically interact with the Neo4j database [13]. The first way is connecting remotely through HTTP (Hypertext Transfer Protocol) or Bolt protocol. To make programming more convenient, Neo4J provides high-level API for a number of mainstream programming languages. The advantage of this approach is, that your application can run independently on a different machine than the database. The disadvantage is the network latency, which can hinder the application performance. The other way is when Neo4J is embedded in an existing Java-compatible application, running on the same JVM. The advantages are minimal latency and access to the low-level Neo4j API directly accessing the database engine.

In this thesis, we have chosen the embedded approach. The reason is, that the step of importing data to the database is only a one-time process, so for this purpose, it is not necessary to separate the application logic from the database application. Also, it is possible to run the import application on the same machine as the database, as the application itself is not resource-intensive. Further to import a large amount of data in the least amount of time as possible the decreased latency compared to the server approach helps.

When choosing the embedded approach, the application can communicate either though Cypher queries or a low-level API directly accessing the database engine. The high-level approach can reduce a lot of code. On the other hand, specific implementation using the low-level API can lead to a dramatic performance boost. The low-level approach was chosen to benefit from the

better performance and the fact, that no complicated queries are required for the import.

3.2 Unique Nodes

The `implicits.bin` file consists of a list of projects and each project is a standalone unit that includes all the necessary information for this unit. This means that a lot of entities are duplicated in the `implicits.bin` file. For example, there might be the same declaration of `java.lang.String` for every project. However, we need each node to be unique. The reason is to consequently create simple queries using the Cypher, not to occupy unnecessary disk and memory, and lastly, but not least to let the Cypher queries be as fast as possible.

This problem is solved by merging nodes that have to be unique, which essentially means looking up the node in the database and when the node does not exist creating the node otherwise passing the found node.

There are entities which are not identified only by their node, but also by their neighbor nodes. For example, declaration, which is represented by the declaration node is also defined by artifact and group node. So when searching for a specific declaration node, it is first necessary to find the corresponding group node, then from its neighbors the artifact node and finally the declaration node. Other approach would be having the `groupId` and the `artifactId` as a property in the declaration node. For the declaration e have chosen the first approach to not duplicate the data.

Another entity, for which this problem occurs is type reference. The type reference is identified by its declaration and its type arguments, which are also type references. This has been solved by creating a property called `typeArgumentExpression`. This property represents a nested list of `declarationId`, which corresponds to their type arguments. With this it is possible to identify type reference node by neighbour declaration node and the `typeArgumentExpression`.

3.3 Import Phases

The whole corpus is imported sequentially, one project at the time, due to the fact, that the projects are sequentially stored in the `implicits.bin` file. Every project consists of modules that are the smallest compact unit of import – this unit consists of all the data that are needed for its import. The graph of this module unit is built and gradually connected in 4 stages.

Declaration creation All the declarations on which the module depends are created with their groups and artifacts

Declaration's signature creation and connection In this stage signature is created with its type references. Also, annotations are connected and it is decided whether the declaration is an implicit conversion.

Call sites creation In this step `Callsite` nodes are created.

Call sites connection At last the `Callsite` nodes are connected.

The gradual creation ensures, that all the dependencies are already created when they are needed. This is essential due to the fact, that there might exist recursive dependencies, which would lead to infinite cycles.

3.4 Nodes Reference Cache vs Graph Searching

During the import, it is necessary to look up already created nodes, as some nodes need to be unique. Two approaches were tried for searching nodes. The first approach uses the graph database itself to do the database lookup. The second approach uses a local cache to store all the necessary node references.

The first approach has a disadvantage in speed, as a lookup in a graph is slower than lookup in a map, because searching one node from the list of all the nodes with a given label is linear operation (to the number of nodes with given label) and look up in map is logarithmic (treemap) or constant operation (hash map). But it is necessary to consider the need to store the cache with all the node references in memory.

These two different approaches were compared with 4.1 MB input on PRL-PRG's server prl4 (Intel Xeon 6140, 2.30GHz with 72 cores, and 256GB of RAM). The conversion phase without the cache took approximately 5 seconds. While using the cache the conversion phase took around 1 second.

Storing the node reference itself is not possible due to the fact, that the reference to the node is only valid in the current transaction and it is not possible to import all the projects in one transaction – as it makes high demand on memory and the final writing to the persistent takes longer. The problem of not being able to store the references in the cache can, fortunately, be solved by only storing the node ids. Because of how the Neo4j stores the nodes in files and memory (has been explained in Section 1.2.2 about Neo4j architecture) the node id is also a pointer to the node and therefore the reference to the node equivalent to the node id. Neo4j core API offers a method, on the current transaction, `getNodeById` for getting the reference to the node from node Id.

3.5 Project Structure and Testing

The implementation has been realized in the same ScalaImplicitAnalysis project which implements the original pipeline for the paper [1], which is stored in

a GitHub repository [21]. The new “Neo4j-new” branch has been created to implement the graph database import.

Project Structure

The data importer was implemented in 1060 lines[3] of Scala code. The main part is implemented as a Scala library class `ImplicitsToNeo4j`. This library is divided into 8 classes. The entry point class `ImplicitsToNeo4j` is responsible for starting up and initializing the Neo4j database and then running conversion on every project from the input file implicits.bin. The responsibility for the conversion of projects to the database is on the `Convertor` class. This class implements the model mapping, while added business logic, that is not visible from the model itself is delegated to the `ModelLogic` class. Further Converter class itself does call the Neo4j API directly. Direct calls to the Neo4j API are done from the `Proxy` class, which is due to two purposes. The first is to create a higher-level interface for creating/merging nodes and setting their properties. The second reason is the implementation of the cache. The `Converter` class is not dependent on how the nodes are looked up. As mentioned “Proxy” serves as a higher-level interface for the Neo4j API, while using cache to store the nodes for quick look-up. Further, the project includes 2 enum classes, which represent all the node labels and relationship types.

Additionally, Scala scripts have been created to fix and validate projects from the input file and to run the Scala importing library easily from the command line without manipulation with jar files.

Finally, other scala or bash scripts were created, which automate reoccurring processes like running Neo4j server over created database files or extracting scripts which help with debugging of the import – extracting a portion of projects from the whole corpus.

Testing

During the development, automated unit tests were done to ensure the ongoing correct functionality. Most importantly every method of `ModelLogic` class is tested. Further integration test has been created to test the whole logic. In this test, a simple project is imported to the database and then the correct creation is verified by database queries. When dealing with the problem of converting the corpus of 7 513 projects, the conversion was first tested on increasing portions of the whole corpus. This conversion was run on a remote server and when a problem with the conversion of a certain project occurred, this project was extracted and the process of conversion was inspected locally with the use of a debugger.

[3]Computed using cloc (c.f. `https://github.com/AlDanial/cloc`), excluding empty lines and comments

Chapter **4**

Assessment

This chapter first describes how the import can be run. Next presents an analysis of running the import on a remote server. Finally shows an analysis of basic queries run on the created database.

4.1 Running the Import and Viewing the Graph

To illustrate the process of import, the following Listing 4.1 shows the necessary commands to get started with a sample project imported. As the import is run in a docker container, the only requirements are docker, git, and GNU make.

Finally, when the import is completed, the Neo4j server in docker can be started by running runNeo4jDocker script. The Neo4j server can be accessed via a web browser (`http://0.0.0.0:7474/browser/`) – Table 4.1 shows authentication which must be filled to access the database.

Connect URL	neo4j://0.0.0.0:7687
Authentication type	Username/Password
Username	neo4j
Password	test

Table 4.1: Neo4j authentication information for accessing running database (Starting the database is described in Listing 4.1).

4.2 Import Analysis

The original corpus has been imported to the graph database. Time analysis has been done on portions of the whole corpus to find out whether the solution can be scaled to more projects and to find out how much the solution could be

```
# Downloading projects
git clone https://github.com/OtakarVinklar/Implicit-analysis-wrapper
cd Implicit-analysis-wrapper
# Downloading scala-implicits-analysis project
# inside the Implicit-analysis-wrapper project
git clone --single-branch --branch new-neo4j \
https://github.com/PRL-PRG/scala-implicits-analysis

#building docker image prlprg/implicit-analysis-pipeline
make -C docker

# Building scala libraries
./run.sh make -C scala-implicits-analysis/libs

# Validating the implicits.bin file
./run.sh amm scala-implicits-analysis/scripts\
/implicits-validate-for-graphcreation.sc example
# Running the import
# the graph database files are created in the example folder
./run.sh amm \
scala-implicits-analysis/scripts/create-graph-db.sc example

# Running Neo4j server
./runNeo4jDocker example
```

Listing 4.1: Manual for running the graph database import.

improved. All the analysis was done on a server with the following specification – Intel Xeon 6140, 2.30GHz with 72 cores and 256GB of RAM.

As seen in Table 4.2 total time needed for importing the whole corpus, that is 4.6 GB of data and 7 513 projects, is approximately 9 minutes. Each part of the program took proportionately the same time, in comparison to different input sizes. The conversion part, which is the part of creating the graph in transaction memory, took 28 % of the total time. This is the part where adjustments to the code can be made. Next 66 % of the total time, took the committing. In this phase nodes and relations from the transaction memory are stored in the files. The performance of this part can be manipulated, by adjusting the size of each commit. The last 6 % of the total time, which is not present in the table, took reading the input file.

The graph in Figure 4.1 shows the dependency of total time on the size of the input file. It can be seen, that the time complexity of the import is linear with the size of the input.

Size	Count	Total [min]	Conversion time [min]	Commit time [min]
417 MB	752	0.902	0.254	0.592
986 MB	1 503	2.091	0.595	1.381
2.3 GB	3 757	4.493	1.272	2.943
4.6 GB	7 513	9.151	2.579	5.989

Table 4.2: Time analysis of importing portions of the whole corpus.

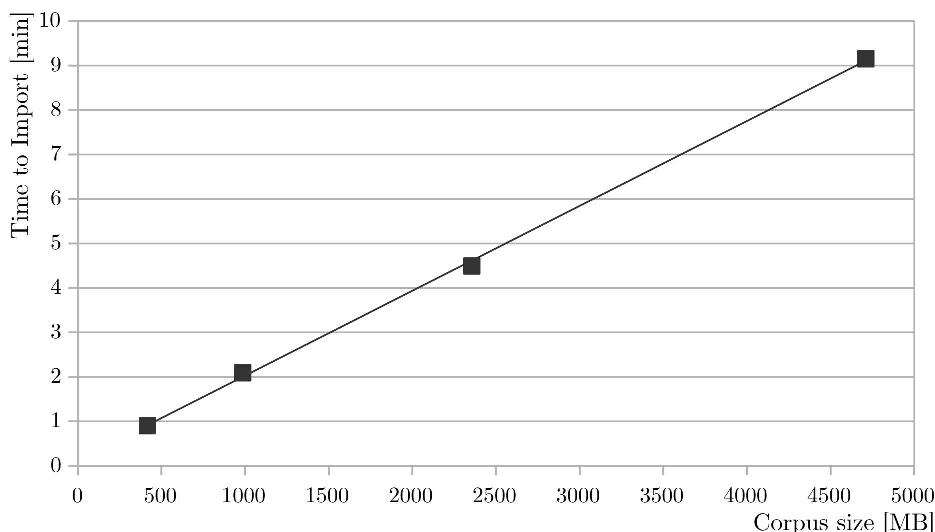

Figure 4.1: Dependency of total import time on input file size.

4.3 Query Analysis

This section discusses the performance of chosen queries to the graph database using Cypher. Table 4.3 shows times needed to complete each query.

The first query from Table 4.3, which finds and counts all the call sites related to the implicit conversions to Java types (Listing 2.2 takes 76 ms. This time is sufficient for most applications, which is mostly due to the specificity of the query. The query planner decides to start the query by finding out the `JAVA` `Language` node, which is a constant operation, as there are only two different `Language` nodes. Next only 17 160 `Declaration` nodes are found by traversing the `IN_LANGUAGE` relationship. It was possible to reduce the searching in the graph only to a minor portion of the whole graph. Next all 21 928 corresponding types are found. Further 1 554 all the `CONVERSION_TO` relationships are traversed to find out all the matching `ImplicitConversion` (1 554) nodes. Finally 20 183 related `Callsite` nodes are found by walking

Query name	Query time
Amount of callsites related to conversion to Java type (2.2)	76 ms
Implicit conversions to java.util.Map (4.2)	30 ms
Implicit callsites and declarations count per project (2.1)	5.5 s

Table 4.3: Query times

across the `DECLARED_BY` relationships and then all the `Callsite` nodes are counted. If the planner would decide to start the query from the other side by first getting all the `ImplicitConversion` nodes, it would take 540 ms to complete the query. This is due to the fact that the query starts with 72 385 `ImplicitConversion` nodes compared to 17 160 `Declaration` nodes, when starting the searching from the `Language` node.

The third query from Table 4.3, finds all the implicit conversions to the `java.util.Map` (Listing 4.2). This query takes 34 ms. If we did not specify or we did not know that this declaration is in java language, the query would take 580 ms. This is due to the starting nodes of the query and the further filtering of the nodes. When starting from the `Language` node first 17 160 declarations are found and then these are filtered to find the `Map` declaration. But, when not specifying the language, the query starts with all the 72 384 `ImplicitConversion` nodes.

```
match
    (d:ImplicitConversion)-[:CONVERSION_TO]-
    (tRef)-[:TYPEREF_DECLARATION]->
    (targetDeclaration:Declaration {declarationId:"java/util/Map#"})
    -[:IN_LANGUAGE]->(:Language {name:"JAVA"})
return d
```

Listing 4.2: Query for getting implicit conversions to `java.util.Map`

The last query in Table 4.3, which has been mentioned in Listing 2.1 takes 5.5 s to complete. This time is still reasonable for analysis application, but would not be for some real-time application. This poor performance is due to the fact, that this query is not suited for the graph database. A large part of the graph needs to be traversed to count all the implicit call sites and implicit declarations and this hinders the performance. First, every project needs to be found, which is a quick operation because each label is indexed. Next, for each project, relationship to all its modules needs to be traversed, and for each module every `DECLARES` relationship needs to be traversed and finally, non-implicit declarations need to be filtered out. Almost the same process (without the filtering of declarations) needs to be done a second time for the implicit call sites, as can be seen from Listing 2.1.

Conclusion

In 2019 Krikava et al published a large scale study on the use of Scala implicits in the real world [1]. The aim was to provide a retrospective on some of the design decisions and quantitatively summarize the use of this somewhat controversial language. They acquired a corpus of over 7K projects from Github with over 18M lines of code. The main problem they had was the rigidity of querying the data. Their solution of extracting a large number of CSV files into R turned rather labor-intensive and error-prone.

This thesis came up with a replacement of the last step to address these shortcomings. Extracted implicits data from the pipeline were refined and stored in the Neo4j graph database without any data duplication. This enabled a simple high-level analysis using Cypher query language. Graph database storage also allows for a simple adding of new relationships between entities. This enables flexible answering of new analysis questions in the future.

Further, the whole corpus containing 7 513 projects was imported to the graph database in less than 10 minutes. Finally, part of the original implicits analysis was recreated on the whole corpus using Cypher. These Cypher queries were analyzed in terms of performance. It was found, that the graph database is well suited for specific questions and offers a high-level interface for simple writing of analysis questions.

Bibliography

1. KŘIKAVA, Filip; MILLER, Heather; VITEK, Jan. Scala implicits are everywhere: a large-scale study of the use of Scala implicits in the wild. *Proceedings of the ACM on Programming Languages*. 2019, vol. 3, no. OOPSLA, pp. 1–28. Available from DOI: `10.1145/3360589`.
2. LOPES, Cristina V; MAJ, Petr; MARTINS, Pedro; SAINI, Vaibhav; YANG, Di; ZITNY, Jakub; SAJNANI, Hitesh; VITEK, Jan. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages*. 2017, vol. 1, no. OOPSLA, pp. 1–28. Available from DOI: `10.1145/3133908`.
3. ODERSKY, Martin. A Brief History of Scala. In: [online] [visited on 2020-05-02]. Available from: `https://www.artima.com/weblogs/viewpost.jsp?thread=163733`.
4. WAMPLER, Dean; PAYNE, Alex. *Programming Scala*. OReilly Media, Inc, USA, 2014. ISBN 978-1-491-94985-6.
5. SCHÄRLI, Nathanael; DUCASSE, Stéphane; NIERSTRASZ, Oscar; BLACK, Andrew P. Traits: Composable units of behaviour. In: *European Conference on Object-Oriented Programming*. 2003, pp. 248–274. Available from DOI: `10.1007/978-3-540-45070-2_12`.
6. ODERSKY, Martin et al. *An overview of the Scala programming language*. 2004. Technical report.
7. ODERSKY, Martin; SPOON, Lex; VENNERS, Bill. *Programming in Scala: a comprehensive step-by-step guide*. Artima, 2016. ISBN 978-0981531687.
8. NAGYA, Gergely; PORKOLÁBA, Zoltán. Performance Issues with Implicit Resolution in Scala. In: *Proceedings of the 10th International Conference on Applied Informatics*. ACM, 2017, pp. 211–223. Available from DOI: `10.14794/ICAI.10.2017.211`.

9. BILL, Venners; FRANK, Sommers. Implicit Design Patterns in Scala. In: [online] [visited on 2020-05-02]. Available from: https://www.lihaoyi.com/post/ImplicitDesignPatternsinScala.html.
10. OLIVEIRA, Bruno CdS; MOORS, Adriaan; ODERSKY, Martin. Type classes as objects and implicits. *ACM Sigplan Notices*. 2010, vol. 45, no. 10, pp. 341–360. Available from DOI: 10.1145/1932682.1869489.
11. WADLER, Philip; BLOTT, Stephen. How to make ad-hoc polymorphism less ad hoc. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 60–76. Available from DOI: 10.1145/75277.75283.
12. ROBINSON, Ian. *Graph Databases: new opportunities for connected data*. Shroff Publishers and Dist, 2016. ISBN 978-1491930892.
13. VUKOTIC, Aleksa; WATT, Nicki. *Neo4j in action*. Manning, 2015. ISBN 9781617290763.
14. NEO4J. openCypher. In: [online]. 2018 [visited on 2020-05-20]. Available from: http://www.opencypher.org/.
15. GEORGIOS, Gousios. GHTorrent. In: [online] [visited on 2020-05-20]. Available from: https://ghtorrent.org/.
16. CENTER, Scala. Scaladex. In: [online] [visited on 2020-05-20]. Available from: https://index.scala-lang.org/.
17. SCALAMETA. Scalameta. In: [online] [visited on 2020-05-20]. Available from: https://scalameta.org/.
18. SCALAMETA. Semanticdb data model. In: [online] [visited on 2020-05-20]. Available from: https://github.com/scalameta/scalameta/blob/master/semanticdb/semanticdb/semanticdb.proto.
19. F, Krikava. OOPSLA19 Artifact - Scala Implicits are Everywhere. In: [online] [visited on 2020-05-20]. Available from: https://github.com/fikovnik/OOPSLA19-artifact/.
20. AL., Ole Tange et. Gnu parallel-the command-line power tool. *Login*. 2011, vol. 36, no. 1, pp. 42–47.
21. PRL-PRG. OOPSLA19 Artifact - Scala Implicits are Everywhere. In: [online] [visited on 2020-05-20]. Available from: https://github.com/PRL-PRG/scala-implicits-analysis.

Appendix A

Acronyms

API Application programming interface

ASCII American Standard Code for Information Interchange

CSV Comma-separated values

FP Functional Programming

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

JSON JavaScript Object Notation

JVM Java Virtual machine

NOSQL Not Only SQL

OOP Object Oriented Programming

REST Representational State Transfer

SBT Simple Build Tool

SQL Standard Query Language

Appendix **B**

Contents of Enclosed Memory Device

All the contents of the enclosed SD-Card can be found in GitHub repositories mentioned in Listing 4.1 also with a guide how to use run the import.
Repository URLs and last commits:

```
URL: https://github.com/OtakarVinklar/Implicit-analysis-wrapper/
Branch: master
Last commit: c897de24c50dfacaeddda0a195ac1818582351ec

URL: https://github.com/PRL-PRG/scala-implicits-analysis
Branch: new-neo4j
Last commit: 031c1fb88741b975745ff3d9cb126fe8f04660e9
```

- `example` .... folder with example input file
  - `run.sh` .... script for running commands in docker
  - `runNeo4jDocker` .... script for running Neo4j server
  - `scala-implicits-analysis` .... source codes for running the import
  - `README.md` .... manual for running the import