Chapter 1

Introduction

In recent years, alternatives to the relational database model employed by SQL have received increasing attention. One such model for data is that of graph databases, which allow queries to pattern match over the graph formed by relations between objects in the the database. SQL is ill-optimised, both in syntax and execution, for highly transitive graph oriented queries. Join syntax, while general, is unwieldly and SQL lacks the ability to easily specify and perform the repetition based queries often used in graph applications. Another trend in the industry is a move towards languages with purely functional, strongly typed features. Fields which require the ability to quickly produce non-trivial, correct code have seen the use of high level languages such as Haskell, Ocaml, and Scala progress from a research interest to a serious paradigm. In these use cases, such as automated trading and safety critical systems, an advanced type system is used to make stronger guarantees about the runtime behaviour of the program and enhances the ability to find bugs at compile time compared to more traditional imperative languages.

As the usage of functional languages for real-world code grows, there is an increasing need to integrate common patterns into the functional paradigm. Access to databases is one such example that requires solving. Frequently, when a functional program requires access to a database, such access is implemented ad-hoc for the specific usage case. For example, a rigid wrapper may be set up to read trade data from a database in a finance firm. This approach reduces generality and limits the kinds of abstractions that may be used. In this example, the wrapper might only support the specific schema required for the types representing the kinds of trade that might appear; the wrapper used for the trading system could not be used to access a database containing a social network. In addition to this, while read-only databases are trivially purely functional data-structures, it is harder to achieve referential transparency when updates are included into the database specification. This is because most database systems use imperative semantics to best make use of the hardware upon which they are implemented.

There has been some exploration into generalised libraries and wrappers for accessing databases in a purely functional way. These may allow a means to construct SQL queries in a type-safe manner **?**, or provide manipulation of the database's state in a referentially

transparent manner **?**. However, they are typically aimed at relational (SQL) databases and do not allow safe generalisation over the types stored in the database.

In this dissertation, I introduce a general, performant, graph database system, written in Scala, that exposes its underlying graph as a purely functional data-structure. Furthermore, it allows for tighter integration into the client application by allowing the application programmer to store their own types in the database, hence reducing the need for marshalling and unmarshalling. Queries constructed using the library are type-checked by the Scala compiler at compile-time, and execute according to a concrete semantics. In addition to these features, the system is constructed in a modular fashion, allowing for the back-end implementation technologies to be interchanged and the domain specific query language to be extended. My initial goals were to build a read-only database with these features and a single back-end which translates the submitted queries to SQL. However, I have significantly exceeded these goals by adding referentially transparent write operations to the database and writing several variations on a back-end which stores the graph using the LMDB key-value datastore.

Chapter 2

Preparation

This chapter discusses the early stages of the project. Once I had settled on a definition of type-safety and settled on using the Scala language for the project, I carried out a brief survey of graph database technologies to construct the language of queries that I intended to implement. Following this, I devised a set of semantics and constructed a system architecture.

The Scala Programming Language

I quickly decided to use Scala for this project. Scala is a JVM based object oriented and functional language. It provides a very advanced type system, allowing for complex abstractions to be built safely. Furthermore, due to its inter-operability with Java, the latter's libraries can be called from Scala, giving an advantage over other comparable languages such as Haskell and Ocaml, especially when there is a need to take advantage of low-level functionality. Finally, I was also familiar with some of Scala's more advanced features from previous experience.

Definition of Type-Safety

When we talk about type-safety, we mean to say that a correctly typed program will not produce run-time errors of a given class. Common breaches of type-safety in typical programs are the use of unchecked exceptions; the overuse of primitive types, especially strings, as parameters to functions; and unmarshalling data from type-erased sources, such as HTTP message bodies. If we do not know whether we should expect a value to be an integer or a string and we try to read it as an integer, there is a possibility of throwing an exception. With this definition in mind, accessing an external, imperative, database from Scala is not type-safe. The database will have some collection of error modes that often manifest as exceptions and typically stores data in a type-erased form that needs to be unmarshalled. Finally, queries to external database systems are typically

constructed as a primitive string that cannot be checked automatically at compile-time. This project addresses these issues by using carefully constructed result container types to correctly handle error cases, including arbitrary exceptions using a pattern-match-able type hierarchy, a Domain Specific Language that is type checked at compile-time by the Scala type checker, and safe, type-class based marshalling.

Existing Graph Databases

This section gives a brief introduction into the main characteristics and classifications of existing graph databases.

Classes of Database

Existing graph databases typically fall into two classes: property graphs and edge-labelled (EL) graphs **?**. EL graphs typically store only a label on each edge, whereas property graphs can store more attributes on edges and are hence closer semantically to relational databases. EL graphs can be more succinctly modelled mathematically, since they can be represented purely using mathematical relations; they also lead to a cleaner syntax (see the DSL syntax subsection of the query language section). Finally, any data expressible using a property graph can be represented using an EL graph by introducing additional nodes to hold attributes.

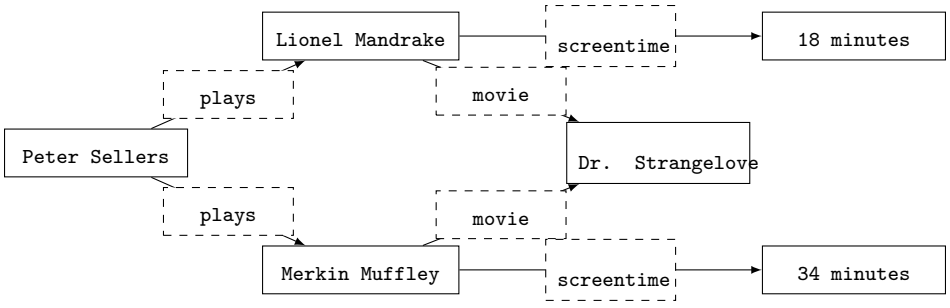

Figure 2.1: An example of an edge-labelled graph. Reproduced from Angles et al. **?**.

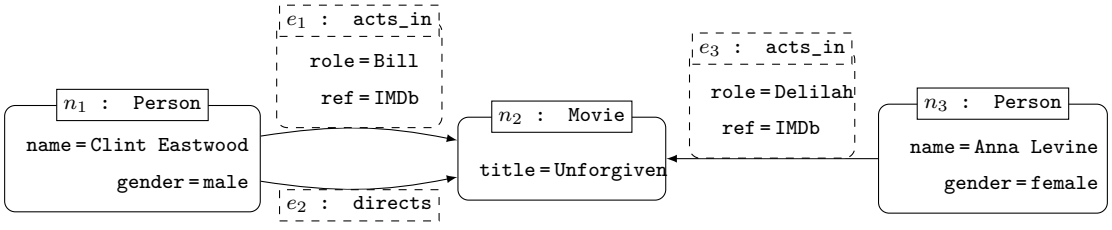

Figure 2.2: An example of a property graph. Reproduced from Angles et al. **?**.

Schema

Current graph database systems often do not have rigid schema **?**. Instead, they use dynamic schema. This provides more flexibility in the shape of data that can be stored in the database; it also allows the types of nodes stored to evolve over time. The downside of this is that it requires additional typing checks at run-time. This also fails to fit into our definition of type-safety, since we have no guarantees about any objects extracted from the database. I have instead opted for a rigid schema which is built using Scala objects and is checked by the Scala compiler at compile-time.

Mutability

Current graph databases tend to be mutable. In order to express concepts that require immutability, such as time-dependent data, the user must impose extra constraints, such as adding time-stamps to the schema of nodes and relations. A lack of immutability also complicates concurrent semantics and implementation of ACID (Atomic, Consistent, Independent, and Durable) transactions. I decided to introduce immutable data structures to the database, which yields ACID properties effectively for free. This is explained in section 2.

Query Languages

Typically, graph database systems have their own query language, the most ubiquitous of which is Neo4j's language, Cypher[1]. This provides ML-style pattern matching of queries against the stored graph, allowing the user to extract arbitrary fields from nodes. In most cases, these are used by generating a query string and submitting it to the database engine. This is inherently not type-safe. The job of ensuring that absolutely every query a given program could generate is valid is intractable to undecidable for non-trivial programs. Therefore, I decided to embed the query language in the host language. This allows the query language's type system to be checked by the host compiler, so we can get much stronger guarantees about program correctness.

Immutability

In order to implement immutability, I needed a way of creating an immutable snapshot of the database at a given point in time. This is achieved using a system of views. A `View` is such an immutable snapshot. When we read from the database, we read from a particular given view. When we write to a particular view of the database, we copy the view, update it, and return the new view. Reads and writes now never interfere with each other. By utilising structure sharing, the impact of storing views can be mitigated.

[1] https://neo4j.com/docs/developer-manual/current/cypher/

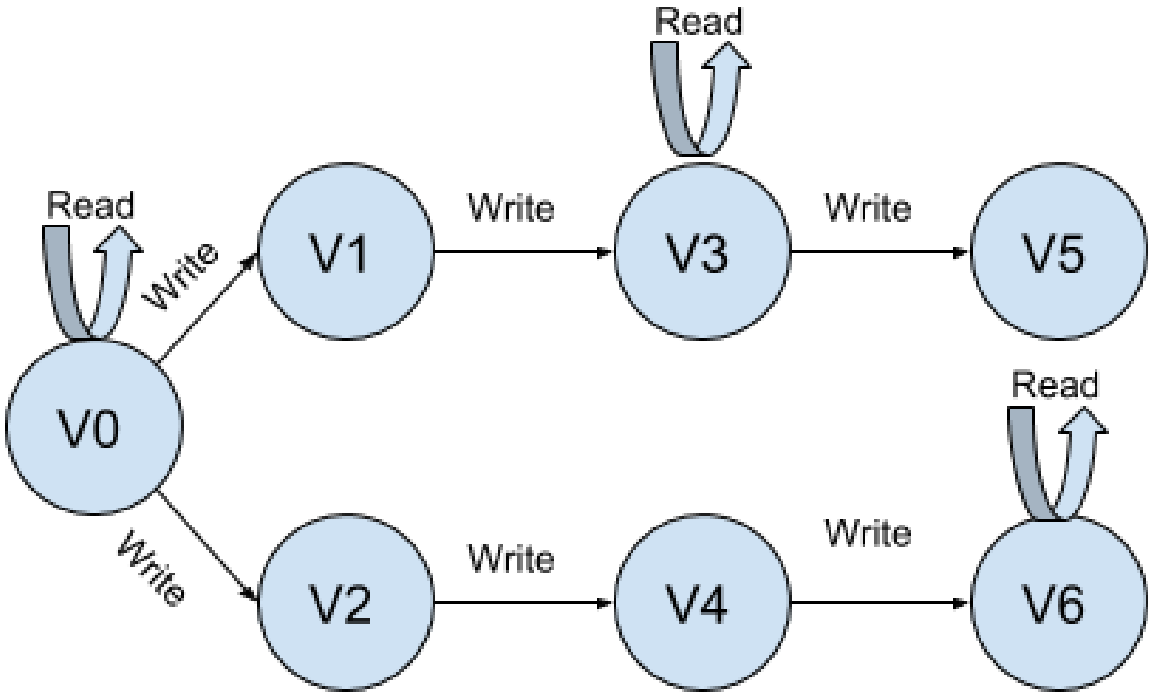

Figure 2.3: An example of reads and writes to an initial view. Separate reads and writes do not interfere with each other.

Query Language

In this next section, I shall define the query language that I constructed and introduce a set of semantics for evaluating queries.

Domain Specific Language Syntax

I spent some time iterating over a DSL syntax and how I wanted queries to look and be structured. My goals were to have a highly composable and expressive, yet small, language. At first I experimented with a Neo4j-style language, with pattern matching syntax, an example of which is below. The query takes a number of fields to fill in, in this case a pair of actors, and finds all valid ways to fill them in subject to the graph constraints.

```
 val coactor = RelationQuery {
    (x: Pattern[Actor], y: Pattern[Actor]) => {
       val m = ?[Movie]
        (x ~ ActsIn ~> m) && (y ~ ActsIn ~> m)
    }
}
```

Find two actors such they both act in the same movie

This is fairly cluttered syntax and difficult to read. Furthermore, the Scala type inference

system is fairly limited, meaning that users would frequently be required to insert type annotations in this syntax. It would also have been further complicated by adding the property graph features of Neo4j. Furthermore, we would not be able to easily and safely replicate Neo4j's ability to extract an arbitrary number of objects of different types from along a path defined by a query. Doing so would require use of structures such as heterogeneously typed lists.

As a result, I settled upon on an edge-labelled-relation-oriented approach which neatens the above query significantly.

```
val coactor = ActsIn --><-- ActsIn
```

Find two actors such they both act in the same movie

This is much more readable and concise. With this syntax, and the assumption that `ActsIn` relates `Actors` to `Movies`, Scala's typesystem can infer that `coactor` relates `Actors` to other `Actors`. There is also syntax for repetitions, intersections and unions. Examples of this syntax can be seen in appendix **??**.

Semantic Definitions

To correctly implement the DSL, we need to mathematically model its operation. Hence, the above DSL queries should correspond to an intermediate representation. I have picked a set of expression constructors to do this. These fall into two kinds: `FindPair` and `FindSingle`, indicating whether they return pairs or individual values.

FindPair queries

$P \to Rel(R)$ Find pairs related by the named relation R
| $RevRel(R)$ Find pairs related by the named relation R in the reverse direction
| $Chain(P, P)$ Find pairs related by the first subquery followed by the second
| $And(P, P)$ Find pairs related by both of the sub-queries
| $AndRight(P, S)$ Find pairs related by P where the right value is a result of S
| $AndLeft(P, S)$ Find pairs related by P where the left value is a result of S
| $Or(P, P)$ Find pairs related by either of the sub-queries
| $Distinct(P)$ Find pairs related by P that are not symmetrical
| Id_A Identity relation
| $Exactly(n, P)$ Find pairs related by n repetitions of P
| $Upto(n, P)$ Find pairs related by up to n repetitions of P
| $FixedPoint(P)$ Find the transitive closure of P

(2.1)

where n denotes a natural number. These queries look up a set of pairs of objects.

FindSingle queries

$$\begin{aligned} S \rightarrow\ & Find(F) \text{ Find values that match the findable F} \\ \mid\ & From(S, P) \text{ Find values that are reachable from results of S via P} \\ \mid\ & AndS(S, S) \text{ Find values that are results of both subqueries} \\ \mid\ & OrS(S, S) \text{ Find values that are results of either subquery} \end{aligned} \tag{2.2}$$

`FindSingle` queries look up sets of single objects.

Other Definitions
In order to model the operation of queries correctly, we need to define the environment in which they operate.

Object Types
Object types are the "real world" types stored in the database. These correspond to the user's Scala classes.

$$\tau \rightarrow A \mid B \mid C \mid ..$$

Named relations
Named relations are the primitive relations between objects

$$R \rightarrow r_1 \mid r_2 \mid ..$$

Findables
`Findable`s are names for defined partial functions

$$F_A \rightarrow f_1 \mid f_2 \mid ...$$

$$f \colon A \rightharpoonup \{True, False\}$$

for some given object type A. (i.e. a findable is an index into the database.)

Schema
A schema, Σ, is made up of three partial functions:

$$\Sigma_{rel}: R \rightharpoonup \tau \times \tau$$
$$\Sigma_{findable}: F \rightharpoonup \tau$$
$$\Sigma_{table}: \tau \rightharpoonup \{True, False\}$$

which give the types of relations and `findable`s, and validiate the existence of a type. When it is obvious from the context, I shall simply use $\Sigma(x)$ to signify application of the appropriate function.

Views

A view $v \in V_\Sigma$, for a given schema Σ represents an immutable state of a database. It consists of a pair of partial functions. Firstly the named-relation look up function:

$$v \in V_\Sigma \Rightarrow v_{rel}(r) \in \wp(A \times B) \text{ if } \Sigma(r) \downarrow (A, B) \tag{2.3}$$

That is, if a relation r is in the schema, then $v(r)$ is a set of pairs of objects with object type $\Sigma(r)$. Here, and from this point onwards I am using $\wp(s)$ to represent the power-set of a set, and $f(x) \downarrow y$ to mean f is defined at x and $f(x) = y$

The next function of a view is the type look-up function, it returns all objects in the view of a given object type:

$$v \in V_\Sigma \Rightarrow v_{table}(A) \in \wp(A) \text{if } \Sigma(A) \downarrow True \tag{2.4}$$

That is, $v(A)$ is a set of objects of type A stored in the view, and A is a member of the schema Σ. Again I shall overload these two functions where it is clear from the context which is to be used.

Typing

In order to ensure the correctness of a given query, there is a type system to verify it. Typing rules take two forms. Firstly typing of pair queries:

$$\Sigma \vdash P: (A, B)$$

Which means "under the schema Σ, pair query P returns a subset of $A \times B$". The second is for single queries:

$$\Sigma \vdash S: A$$

Which means "under the schema Σ single query returns a subset of A"

The rules of the first kind are as follows:

$$(\text{Rel})\frac{\Sigma(r) \downarrow (A, B)}{\Sigma \vdash Rel(r) \colon (A, B)} \tag{2.5}$$

$$(\text{Rev})\frac{\Sigma(r) \downarrow (B, A)}{\Sigma \vdash Rel(r) \colon (A, B)} \tag{2.6}$$

$$(\text{Id})\frac{\Sigma(A) \downarrow True}{\Sigma \vdash Id_A \colon (A, A)} \tag{2.7}$$

$$(\text{Chain})\frac{\Sigma \vdash P \colon (A, B) \Sigma \vdash Q \colon (B, C)}{\Sigma \vdash Chain(P, Q) \colon (A, C)} \tag{2.8}$$

$$(\text{And})\frac{\Sigma \vdash P \colon (A, B) \Sigma \vdash Q \colon (A, B)}{\Sigma \vdash And(P, Q) \colon (A, B)} \tag{2.9}$$

$$(\text{Or})\frac{\Sigma \vdash P \colon (A, B) \Sigma \vdash Q \colon (A, B)}{\Sigma \vdash Or(P, Q) \colon (A, B)} \tag{2.10}$$

$$(\text{Distinct})\frac{\Sigma \vdash P \colon (A, B)}{\Sigma \vdash Distinct(P) \colon (A, B)} \tag{2.11}$$

$$(\text{AndLeft})\frac{\Sigma \vdash P \colon (A, B) \Sigma \vdash S \colon (A)}{\Sigma \vdash AndLeft(P, S) \colon (A, B)} \tag{2.12}$$

$$(\text{AndRight})\frac{\Sigma \vdash P \colon (A, B) \Sigma \vdash S \colon B}{\Sigma \vdash \Sigma \vdash AndRight(P, S) \colon (A, B)} \tag{2.13}$$

$$(\text{Exactly})\frac{\Sigma \vdash P \colon (A, A)}{\Sigma \vdash Exactly(n, P) \colon (A, A)} \tag{2.14}$$

$$(\text{Upto})\frac{\Sigma \vdash P \colon (A, A)}{\Sigma \vdash Upto(n, P) \colon (A, A)} \tag{2.15}$$

$$(\text{FixedPoint})\frac{\Sigma \vdash P \colon (A, A)}{\Sigma \vdash FixedPoint(P) \colon (A, A)} \tag{2.16}$$

The rules for types of Single queries are similar:

$$(\text{Find})\frac{\Sigma(f)\downarrow(A)}{\Sigma\vdash Find(f)\colon A} \tag{2.17}$$

$$(\text{From})\frac{\Sigma\vdash P\colon(A,B)\Sigma\vdash S\colon A}{\Sigma\vdash From(S,P)\colon B} \tag{2.18}$$

$$(\text{AndS})\frac{\Sigma\vdash S\colon A\Sigma\vdash S'\colon A}{\Sigma\vdash AndS(S,S')\colon A} \tag{2.19}$$

$$(\text{OrS})\frac{\Sigma\vdash S\colon A\Sigma\vdash S'\colon A}{\Sigma\vdash OrS(S,S')\colon A} \tag{2.20}$$

Semantics

To fully define the query language, we need to define the semantics. I have defined two collections of semantics: operational style and denotational style.

Operational Semantics

Now we shall define a set of rules for determining if a pair of objects is a valid result of a query. We are interested in forming an approximation relation $a \lhd_A Q$ to mean"a is a valid result of query Q with type A". This is dependent on the current view $v : View_\Sigma$. Hence we define $(a,b) \lhd_{(A,B),v} P$ for pair queries P and $a \lhd_{A,v} S$ for single queries S:

$$(\text{Rel})\frac{(a,b)\in v(r)}{(a,b)\lhd_{(A,B),v} Rel(r)} \tag{2.21}$$

$$(\text{Rev})\frac{(b,a)\in v(r)}{(a,b)\lhd_{(A,B),v} RevRel(r)} \tag{2.22}$$

$$(\text{Id})\frac{a\in v(A)}{(a,a)\lhd_{(A,A),v} Id_A} \tag{2.23}$$

$$(\text{Distinct})\frac{(a,b)\lhd_{(A,B),v} Pa\neq b}{(a,b)\lhd_{(A,B),v} Distinct(P)} \tag{2.24}$$

$$(\text{And})\frac{(a,b)\lhd_{(A,B),v} P(a,b)\lhd_{(A,B),v} Q}{(a,b)\lhd_{(A,B),v} And(P,Q)} \tag{2.25}$$

$$(\text{Or1})\frac{(a,b)\lhd_{(A,B),v} P}{(a,b)\lhd_{(A,B),v} Or(P,Q)} \tag{2.26}$$

$$\text{(Or2)}\frac{(a,b) \lhd_{(A,B),v} Q}{(a,b) \lhd_{(A,B),v} Or(P,Q)} \tag{2.27}$$

$$\text{(Chain)}\frac{(a,b) \lhd_{(A,B),v} P(b,c) \lhd_{(B,C),v} Q}{(a,c) \lhd_{(A,C),v} Chain(P,Q)} \tag{2.28}$$

$$\text{(AndLeft)}\frac{(a,b) \lhd_{(A,B),v} Pa \lhd_{A,v} S}{(a,b) \lhd_{(A,B),v} AndLeft(P,S)} \tag{2.29}$$

$$\text{(AndRight)}\frac{(a,b) \lhd_{(A,B),v} Pb \lhd_{B,v} S}{(a,b) \lhd_{(A,B),v} AndRight(P,S)} \tag{2.30}$$

$$\text{(Exactly\_0)}\frac{(a,b) \lhd_{(A,A),v} Id_A}{(a,b) \lhd_{(A,A),v} Exactly(0,P)} \tag{2.31}$$

$$\text{(Exactly\_n+1)}\frac{(a,b) \lhd_{(A,A),v} P(b,c) \lhd_{(A,A),v} Exactly(n,P)}{(a,c) \lhd_{(A,A),v} Exactly(n+1,P)} \tag{2.32}$$

$$\text{(Upto\_0)}\frac{(a,b) \lhd_{(A,A),v} Id_A}{(a,b) \lhd_{(A,A),v} Upto(0,P)} \tag{2.33}$$

$$\text{(Upto\_n)}\frac{(a,b) \lhd_{(A,A),v} Upto(n,P)}{(a,b) \lhd_{(A,A),v} Upto(n+1,P)} \tag{2.34}$$

$$\text{(Upto\_n+1)}\frac{(a,b) \lhd_{(A,A),v} P(b,c) \lhd_{(A,A),v} Upto(n,P)}{(a,c) \lhd_{(A,A),v} Upto(n+1,P)} \tag{2.35}$$

$$\text{(fix1)}\frac{(a,b) \lhd_{(A,A),v} Id_A}{(a,b) \lhd_{(A,A),v} FixedPoint(P)} \tag{2.36}$$

$$\text{(fix2)}\frac{(a,b) \lhd_{(A,A),v} P(b,c) \lhd_{(A,A),v} FixedPoint(P)}{(a,b) \lhd_{(A,A),v} FixedPoint(P)} \tag{2.37}$$

And the `FindSingle` rules:

$$\text{(Find)}\frac{a \in v(A) f(a) \downarrow True}{a \lhd_{A,v} Find(f)} \tag{2.38}$$

$$\text{(From)}\frac{a \lhd_{A,v} S(a,b) \lhd_{(A,B),v} P}{b \lhd_{A,v} From(S,P)} \tag{2.39}$$

$$\text{(AndS)}\frac{a \lhd_{A,v} Sa \lhd_{A,v} S'}{a \lhd_{A,v} And(S,S')} \tag{2.40}$$

$$(\text{OrS1})\frac{a \lhd_{A,v} S}{a \lhd_{A,v} Or(S, S')} \tag{2.41}$$

$$(\text{OrS1})\frac{a \lhd_{A,v} S'}{a \lhd_{A,v} Or(S, S')} \tag{2.42}$$

Denotational Semantics The operational semantics clearly demonstrate membership of a query, but do not give a means to efficiently generate the results of query. To this end, we introduce denotations $[\![P]\!]$ and $[\![S]\!]$ such that

$$\Sigma \vdash P\colon (A, B) \Rightarrow [\![P]\!]\colon View_\Sigma \to \wp(A \times B)$$

and

$$\Sigma \vdash S\colon A \Rightarrow [\![S]\!]\colon View_\Sigma \to \wp(A)$$

Such denotations should be compositional and syntax directed, whilst still corresponding to the operational semantics. The denotations for `FindPair` queries are as follows:

$$[\![Rel(r)]\!](v) = v(r) \tag{2.43}$$

$$[\![RevRel(r)]\!](v) = swap(v(r)) \tag{2.44}$$

$$[\![Id_A]\!](v) = dup(v(A)) \tag{2.45}$$

$$[\![Chain(P, Q)]\!](v) = join([\![P]\!](v), [\![Q]\!](v)) \tag{2.46}$$

$$[\![And(P, Q)]\!](v) = [\![P]\!](v) \cap [\![Q]\!](v) \tag{2.47}$$

$$[\![Or(P, Q)]\!](v) = [\![P]\!](v) \cup [\![Q]\!](v) \tag{2.48}$$

$$[\![AndLeft(P, S)]\!](v) = filterLeft([\![P]\!](v), [\![S]\!](v)) \tag{2.49}$$

$$[\![AndRight(P, S)]\!](v) = filterRight([\![P]\!](v), [\![S]\!](v)) \tag{2.50}$$

$$[\![Distinct(P)]\!](v) = distinct([\![P]\!](v)) \tag{2.51}$$

$$[\![Exactly(n, P)]\!](v) = (\lambda pairs.join([\![P]\!](v), pairs))^n [\![Id_A]\!](v) \tag{2.52}$$

$$[\![Upto(n, P)]\!](v) = (\lambda pairs.join([\![P]\!](v), pairs) \cup pairs)^n [\![Id_A]\!](v) \tag{2.53}$$

$$[\![FixedPoint(P)]\!](v) = fix(\lambda pairs.join([\![P]\!](v), pairs) \cup pairs) \text{ in the domain } closure(A, v) \tag{2.54}$$

And similarly with single queries:

$$[\![Find(f)]\!](v) = \{a \in v(A) \mid f(a) \downarrow True\} \text{ for } \Sigma(f) = A \tag{2.55}$$

$$[\![From(S,P)]\!](v) = \{b \mid (a,b) \in [\![P]\!](v) \& \wedge a \in [\![S]\!](v)\} \tag{2.56}$$

$$[\![AndS(S,S')]\!](v) = [\![S]\!](v) \cap [\![S']\!](v) \tag{2.57}$$

$$[\![OrS(S,S')]\!](v) = [\![S]\!](v) \cup [\![S']\!](v) \tag{2.58}$$

with the following definitions:

$$swap(s) = \{(b,a) \mid (a,b) \in s\}$$

$$dup(s) = \{(a,a) \mid a \in s\}$$

$$join(p,q) = \{(a,c) \mid \exists b.(a,b) \in p \wedge (b,c) \in q\}$$

$$distinct(s) = \{(a,b) \in s \mid a \neq b\}$$

$$filterLeft(p,s) = \{(a,b) \in p \mid a \in s\}$$

$$filterRight(p,s) = \{(a,b) \in p \mid b \in s\}$$

I have also proved in appendix **??** the correspondence of these two formulations of semantics.

Commands

In addition to the query construction language, there are five commands which make use of the queries:

$$\begin{aligned} findPairs(P,v) &= [\![P]\!](v) \\ find(S,v) &= [\![S]\!](v) \\ shortestPath(a,a',P,v) &= \text{An } \texttt{Option} \text{ of Shortest path, if it exists, from } a \text{ to } a' \\ &\quad \text{in the graph defined by } [\![P]\!](v) \\ allShortestPaths(a,P,v) &= \text{The set of the shortest path to each element reachable} \\ &\quad \text{from } a \text{ in the graph defined by } [\![P]\!](v) \end{aligned}$$

The definition of the write command is slightly more complicated. It needs to return an updated view with its look up functions now returning the inserted objects. Write takes a collection, rs, of correctly typed named relation instances.

$$rs = \{(a_i, r_i, b_i) \in (A \times R \times B) \mid 0 < i \leqslant n\} \text{ for some } n \text{ being the size of the set.}$$

and updates the view as so:

$$\begin{aligned} write(rs, v) = & v\,[A \mapsto v_{table}(A) \cup \{a_i \mid 0 < i \leqslant n\}] \\ & [B \mapsto v_{table}(B) \cup \{b_i \mid 0 < i \leqslant n\}] \\ & [r \mapsto v_{rel}(R) \cup \{(a_i, b_i) \mid 0 < i \leqslant n\}] \end{aligned} \tag{2.59}$$

Summary

To summarise, this past section defines a query DSL, a type system for checking validity of queries, and a set of semantics for executing queries. Furthermore, I have then defined a set of commands which allow queries to execute and views to be generated.

Starting Point

In addition to Scala, I have made use of several libraries and tools, some of which were not foreseen in my project proposal. Some were used for the actual construction of the project, whereas others were used for auxiliary tasks such as testing and debugging or generating datasets. All of these tools have licenses allowing free usage (the exact license for each is given in the footnotes) and no modifications to these libraries were performed.

SBT[2]	Scala Build Tool (SBT) is a package manager and build tool for Scala which allows access to Java libraries.[3]
PostgreSQL[4]	An open-source, multi-platform SQL implementation.[5]
LMDB[6]	An open source, highly optimised key-value datastore.[7]
Scalaz[8]	A library for Scala providing typeclasses and syntax to aid advanced functional programming.[9]
JDBC[10]	Java's standard library support for SQL connections. Used to interact with a postgres database. I had originally planned to use the higher level wrapper scalikeJDBC,[11] however, this did not allow such fine control over queries as I would have liked.[12]
LMDBJava[13]	A JNI library allowing access to the LMDB datastore. Previously, I experimented with another library, LMDBJni. However, upon discovery of a bug in the JNI code, I switched to LMDBJava.[14]
Junit[15]	A unit testing library for JVM languages.[16]
SLF4J[17]	A logging library for JVM languages.[18]
Spray-JSON[19]	A JSON library for Scala, used to generate benchmarking datasets.[20]
Python[21]	A scripting language used to generate test datasets.[22]

[2] //www.Scala-sbt.org/
[3] https://github.com/sbt/sbt/blob/1.x/LICENSE
[4] //www.postgresql.org/
[5] https://opensource.org/licenses/postgresql
[6] //symas.com/lmdb/
[7] http://www.openldap.org/software/release/license.html
[8] https://github.com/Scalaz/Scalaz
[9] https://github.com/scalaz/scalaz/blob/series/7.3.x/LICENSE.txt
[10] http://www.oracle.com/technetwork/java/javase/jdbc/index.html
[11] http://scalikejdbc.org/
[12] https://jdbc.postgresql.org/about/license.html
[13] //github.com/lmdbjava/lmdbjava
[14] https://github.com/lmdbjava/lmdbjava/blob/master/LICENSE.txt
[15] https://junit.org/junit5/
[16] https://junit.org/junit4/license.html
[17] https://www.slf4j.org/
[18] https://www.slf4j.org/license.html

Software Engineering

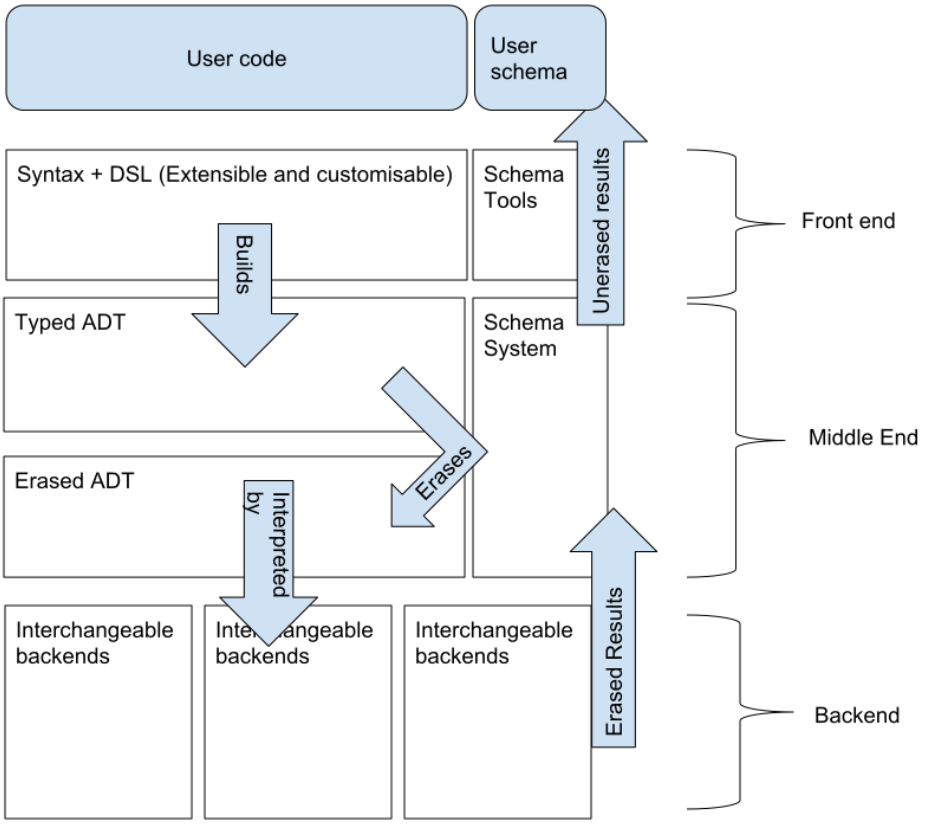

Figure 2.4: A map of the architecture of the system

The system was designed to be as modular and re-implementable as possible. To this end, it consists of several interchangeable components; a map of this is provided in figure 2.4. The front-end mostly consists of interfaces and syntax for building queries and the schema for the database. The front-end's DSL translates into the underlying typed ADT. This then has its compile-time type information erased to become an un-typed query ADT, which is easier and cleaner to interpret. Finally, there are three interfaces that a back-end system must implement: DBBackend, DBInstance and DBExecutor. These specify a root back-end object that opens an instance which represents a database connection. Each DBInstance has an executor that allows us to execute commands. The results of these commands are then un-erased to return them as the correct type.

I progressed through the project using an iterated waterfall method. Each time it came to add a new module, such as a back-end, I planned the overall execution method and storage

[19]https://github.com/spray/spray-json
[20]https://github.com/spray/spray-json/blob/master/LICENSE
[21]https://www.python.org/
[22]https://docs.python.org/3/license.html

structure, then wrote the interfaces for submodules. Next, I filled in the components and tested the new back-end against a growing set of unit and regression tests.

Requirements Analysis

Much of the early project hinged on being able to successfully construct the algebraic-data-types introduced above. Another high priority and medium risk task was to construct a schema representation that allowed the aforementioned ADT to be embedded into the Scala type-system as seamlessly as possible. With these completed, the further tasks could be completed sequentially.

Deliverable	Priority	Risk	Difficulty
Success Criteria			
Design Syntax and ADT	High	Medium	High
Design Schema Impl.	High	Medium	High
Design Module Interfaces	High	Low	Low
Memory Back-End (Reads)	High	Medium	Low
Extensions			
Memory Back-End (Writes)	Medium	Low	Low
Construct Unit Tests	Medium-High	Low	Low
SQL Back-End	Medium	Medium	Medium-High
LMDB Back-End	Low-Medium	Low	Medium-High
LMDB Optimisations	Low	Low	High
Benchmarking and Eval.	Medium-High	Medium	Medium

Scala Techniques

I have made use of several advanced programming techniques which are specific to the Scala language.

Type-Enrichment

The type-enrichment feature of the Scala language allows retroactive addition of methods to previously defined types. Type-enrichment is performed by creating an implicit class, taking a single underlying value of some given type. The methods defined in the implicit class can now be called as if they were methods of the underlying class, provided the implicit class is in scope.

```
object Outer {
    implicit class EnrichInt(i: Int){
        def neg: Int = -i
        def  succ: Int = i + 1
        def mod(n: Int) = i - (i / n)*n
    }
}
```

When Outer is imported, we can now do

```
5.neg == -5
6.succ == 7
10 mod 4 == 2
```

This allows us to add syntax to types where a small number of core methods have been defined.

Implicit Parameters

Another advanced feature of Scala that I have used is that of implicit parameters to functions and class constructors. Values such as vals, functions, and classes can be declared with an implicit tag. Functions and classes can declare additional parameters as implicit. When these functions are called, the implicit parameter can be omitted if and only if there is an unambiguous implicit value of the correct type in scope.

```
object foo {
    implicit val i = 5
}

object bar {
    def plusImpl(j: Int)(implicit k: Int): Int = j + k
}
```

```
import foo._

bar.plusImpl(4) == 9
```

This is typically used for purposes such as passing around values that are defined once and used many times in a program, such as an execution context, or a logging framework instance.

Functions can also be implicit and take implicit values

```
implicit def f(implicit a: A): B = ...
```

When an implicit value of type A is available, then an implicit value of type B is also available.

This can be made more interesting when we include parameterised generic types. This allows us to get the compiler to do work in the manner of an automated theorem prover (by the Curry-Howard correspondence) at compile-time.

```
object Defs {
    implicit val s: String = "foo"
    implicit val i: Int = 2
    Implicit def toList[A](implicit a: A): List[A] = List(a, a, a)

    def getImplicit[A](implicit a: A): A = a
}
```

```
import Defs._

getImplicit[List[Int]] == List(2, 2, 2)
getImplicit[List[List[String]] ==
        List(List("foo", "foo", "foo"),
        List("foo", "foo", "foo"),
        List("foo", "foo", "foo"))
```

Typeclass Pattern

A further combination of these two patterns is the typeclass pattern. We define a typeclass for a type by defining as methods on a trait the operations we want on the type. We can then define implicit objects which work as type class instances for the types we want. We can also use implicit functions to generate typeclass instances in a manner similar to a proof tree. We can then use the type-enrichment feature to add methods to values of a type that is a member of the typeclass.

```
trait Monoid[A] {
     def id: A
     def op(a1: A, a2: A): A
}
```

Definition of a monoid typeclass instance using F-type polymorphism

```
object Instances {
  implicit object IntMonoid extends Monoid[Int] {
    override val id: Int = 0
    override def op(a1: Int, a2: Int): Int = a1 + a2
  }

  implicit def ListMonoid[A] = new Monoid[List[A]] {
    override val id: List[A] = List()
    override def op(a1: List[A], a2:  List[A]):  List[A] =
      a1 ++ a2
    }


  implicit def PairMonoid[A, B](
    implicit MA: Monoid[A], MB: Monoid[B]
  ) = new Monoid[(A, B)]{
    override val id: (A, B) = (MA.id, MB.id)
    override def op(p: (A, B), q: (A, B)) =
      (MA.op(p._1, q._1), MB.op(p._2, q._2))
  }
}
```

Definition of several monoid instances, including a pair monoid that combines monoids.

```
object Syntax {
  implicit class MonoidOps[A](a: A)(implicit ma: Monoid[A]) {
     def *(n: Int): A = if (n == 0) ma.id else ma.op(a, *(n-1))
  }
}
```

Definition of a multiplication operation on instances of a monoid.

```
 (5, List("f")) * 3 == (15, List("f", "f", "f"))
```

Usage of the multiplication operation on a tuple.

It is clear that these patterns allow for very expressive structures and abstractions to be built in Scala. I use these frequently within the project to neaten code and to achieve type-safety.

Chapter 3

Implementation

In this section, I shall first focus on common techniques when building the database, then explain construction of the front- and middle-ends. After this, I shall look at common structures shared by the various back-end implementations and finally, I shall conclude with an examination of each of the back-end implementations that I have written.

Note on Purity and Concurrency

All of the back-ends aim to preserve the global immutability of the database. The immutable semantics of the database also mean that queries generally do not interfere with each other. This means that we can avoid keeping locks or creating large transactions. Hence, the back-ends do not require much work to maintain correct concurrency.

Functional Programming Techniques

Monadic Compilation

At several points in building a back-end, it becomes necessary to transform one algebraic type to another. This is typically performed by walking over a tree, whilst keeping some mutable state representing parts of the tree that are relevant. One example is converting an intermediate representation to SQL output code. Here, we may want to extract common sub-expressions into a dictionary, or pick out all the database tables that need to be accessed by the query. This can be encoded by folding a State Monad instance over the tree.

The state monad is an abstraction over functions that chain an immutable state through successive computations.

```
class State[S, A](r: S => (S, A)) {
      def runState(s: S): (S, A) = r(s)
}
```

```
object Example {
  Implicit def StateMonadInstance[S] = new Monad[State[S, _]] {
     def point[A](a: A) = new State(s => (s, a))
     def bind[A, B](sa: State[S, A], f: A => State[S, B]) = new State(
        s => {
          val (s', a) = sa.runState(s)
          f(a).runState(s')
        }
      )
    }
  }
```

To define a monadic compiler, we define a recursive function which chains state monad objects together.

```
def compile(adt: ADT): State[S, Res] = adt match {
   case basis => ...
   case Pattern(a, b) => for {
      ca <- compile(a)
      cb <- compile(b)
   } yield foo(ca, cb)
}
```

Constrained Future Monad

Part of the goal of type-safety is the recovery of error cases. Typically, this would be handled in a JVM program through the use of exceptions. However the presence of unchecked exceptions on the Java platform makes it difficult to ensure that all error cases are accounted for. A more functional approach is the use of the exception monad `?`. In Scala, this manifests as the built in Try monad and Scalaz's `\/` (`Either`) monad. `Try[A]` has two case classes: `Failure(e: Throwable)` and `Success(a: A)`, while `E \/A` has the case classes `-\/(e: E)` and `\/-(a: A)` (`Left` and `Right`). Since `Try`'s failure case is the unsealed `Throwable` trait, we do not really have a way to ensure we have handled all error cases at compile-time, whereas `Either` has a parameterised error type, which can be a sealed type hierarchy, which is then checked by the Scala compiler at run-time. For example, consider the simple interpreter below. All error cases are proved to be handled by the type-system.

```
sealed trait Error
case object DivByZero extends Error
case object Underflow extends Error

sealed trait NoError
```

A simple pair of error hierarchies. One signifying errors, and NoError indicating a lack of error. Note that *NoError* cannot be instantiated or sub-classed.

```
def eval(e: Expr): Error \/ Int =
  e match {
    case Div(a, b) => for {
        aRes <- eval(a)
        bRes <- eval(b)
        r <- if (bRes == 0) DivByZero.left else (aRes/bRes).right
      } yield r

    case Sub(a, b) => for {
        aRes <- eval(a)
        bRes <- eval(b)
        r <- if(bRes > aRes) Underflow.left else (aRes-bRes).right
    } yield r
}
```

A simple interpreter for a simple algebraic datatype for expressions that returns an error or a result

```
def evalAndPrint(e: Expr): NoError \/ String =
  eval(e) match {
    case \/-(i) => i.toString.right
    case -\/(e) => e match {
        case DivByZero => "division by zero".right
        case UnderFlow => "underflow".right
    }
  }
```

A simple result printer that correctly handles all error cases and reduces its error parameter to the un-instantiable *NoError* type

Typically, we deal with cases which might take a significant amount of time to return. So rather than using a simple `Either` monad, we lift it into an asynchrony monad. There are several options to choose from for an asynchrony monad. I chose the built in `Future` over more exotic alternatives, such as the Scalaz `Task`, since it is relatively widely used, and I have some familiarity with it from past projects. `Futures` also capture thrown unchecked exceptions, which makes handling them a little easier. Hence we are interested in passing around `Future[E \/A]` around, for a sealed type hierarchy `E`. There is also the issue of the Java libraries used (for SQL and LMDB access) throwing exceptions, and unexpected exceptions turning up in code. Fortunately, the Future container catches these, acting like an asynchronous `Try[E \/A]`. This causes issues as we do not have the sealed trait property of errors as we have above. To solve this, I introduced the `ConstrainedFuture[E, A]` monad, which has the requirement that the error case type parameter `E` implements the `HasRecovery` typeclass for converting any `Throwable` to an `E`.

```
trait HasRecovery[E] {
 def recover(t: Throwable): E
}
```

The underlying `Future` is kept private, and can only be accessed via the `run` method, which calls the `recover` method on any errors (tail recursively, so any exceptions thrown during execution are also handled). By this construction, we ensure all non-fatal error cases are contained in a type-safe way.

Operation Monad

As stated in section 2, typical database operations take a view as a parameter, inspect the view, return a value and may also insert a new view. This requires interplay between the `ConstrainedFuture` (to handle failure and asynchrony) and `State` (to chain together updates to the view representing current state) monads. Hence we use the `Operation[E, A]` monad, which wraps a function $(ViewId \Rightarrow ConstrainedFuture[E, (ViewId, A)])$ in a similar way to how `State` monad chains together functions $S \Rightarrow (S, A)$. Each of the commands on a database yields an operation.

Local and Global State

Although it would be preferable to only use purely functional folds, maps, and immutable data structures everywhere within the project, for certain, high frequency, performance critical tasks, using purely immutable structures slows us down. Recursive functions tend to use more stack space than the JVM has available in many situations. Hence for tasks such as retrieving values for result sets, pathfinding across large relations, and building indices, I have opted to make use of mutable data structures locally, using builders[1] for collections such as sets. This leads to a dramatic increase in speed, especially for when large numbers of elements are added sequentially to collections that are not lists. In these cases, the mutable state never leaks out of the functions that make use of the mutability.

Schema Implementation

One of the goals of the project was to allow close to arbitrary user objects (assuming that they are finite) to be inserted and retrieved from the database. This was achieved using the typeclass pattern.

[1] https://docs.scala-lang.org/overviews/core/architecture-of-scala-collections.html#builders

Schema Hierarchy

In order to work with the database, a class needs to have an instance of the `SchemaObject` typeclass.

```
sealed trait SchemaObject[A] {
// components for constructing DB
 def schemaComponents: Vector[SchemaComponent]  tables
 def any: Pattern[A] // Findable that matches any A
 def findable(a: A): DBTuple[A] // convert an object to a findable
 def name: TableName // name of the table
 def fromRow(row: DBObject): ExtractError \/ A // unmarshall an A
}
```

`SchemaObject` is sealed, so can only be implemented by implementing one of its subclasses. Currently, for the sake of simplicity, there are five: `SchemaObject0` .. `SchemaObject4`, with the number indicating the number of underlying database fields required. These simplify the construction of a type-class instance to marshalling values of the object type to tuples of `Storable`, primitive types. Examples can be seen in appendix **??**.

DBObjects

To store objects in the database under various back-ends, we need to have a type erased (at compile-time, but not run-time) version of the objects. Once we have a primitive representation of an object from the `SchemaObject`, we can fully unerase it by converting it to a `DBObject`. A `DBObject` is simply a collection of type tagged fields (`DBCell`). These can now be easily inserted or retrieved from a database.

```
type DBObject = Vector[DBCell]

type DBCell = DBInt of Int
| DBString of String
| DBBool of Boolean
| DBDouble of Double
```

Pseudo-ml definitions of `DBObject` and its components

Unerasure

In order to correctly retrieve values from the database, we need to be able to undo the erasure process. This can be performed using the unmarshalling methods derived from the `SchemaObject` for an object type.

Relations

In order for type checking to work, relations need to have type parameters for the object types they link. Hence, to define relations in the schema, the user needs to define objects that extend the relation interface.

```
abstract class Relation[A, B](
    implicit val sa: SchemaObject[A], val sb: SchemaObject[B]
) extends FindPairAble[A, B]
```

SchemaDescription

In order to build the database structures the back-ends need a definitive collection of schema to include. This is performed using a `SchemaDescription` object, which simply holds a collection of `SchemaObjects` and `Relations` which need to be used by the database.

Findables

For the sake of simplicity, I have only implemented findables which test if fields of objects match particular values. This allows us to look for specific objects or match particular fields. This also makes indexing easier to implement.

Query ADT

The intermediate representation terms described in section 2 are implemented in Scala by a pair of ADT hierachies: a typed and type-erased equivalent for each. The typed ADTs are parameterised by the object types which they look up. Using this parameterisation, I have encoded in type of the ADTs the inductive type rules in the semantics such that the Scala compiler checks the type of any generated query and does type inference for us. The definitions of these Scala classes can be found in appendix **??**. The only rules that cannot be checked at compile-time are whether the schema description contains the relation in instances of the (`Rel`) rule, the type `A` for the (Id_A) rule, and the (`Find`) rule, as we cannot predict the contents of the SchemaDescription at compile-time without dependent types. The typed ADTs are erased, with respect to a schema, into their unsafe equivalents in order to be executed. If an AST node makes reference to a non-existent table or relation, a run-time error is created in the `Either` return type. The constructors of the type-erased ADT nodes are private to the enclosing package, meaning that they can only be created by erasing a typed ADT.

Commands

As specified in the previous chapter, each back-end needs to implement five commands:

- `find(S): Operation[E, Set[A]]`
- `findPairs(P): Operation[E, Set[(A, B)]]`
- `ShortestPath(start, end, P): Operation[E, Option[Path[A]]`
- `allShortestPaths(start, P): Operation[E, Set[Path[A]]`
- `insert(relations): Operation[E, Unit]`.

Each command should return an `Operation` of the correct type.

DSL

The DSL mostly consists of syntactic sugar to make queries easier to read. It is implemented using the type-enrichment pattern. Both `Relation` and `FindPair` implement the trait (interface) `FindSingleAble`, so we can use type-enrichment to write new DSL operators. The main thing to note in the DSL is the use of arrows to chain relations together. Examples of the DSL can be seen in appendix **??**.

Common Generic Algorithms

During construction of the database, several common patterns of problems emerged with slight differences. Hence, I have written relatively optimised generic versions of these algorithms such that different back-ends can make use of them regardless of the underlying types. These algorithms are found in `core.utils.algorithms`.

Simple Traversal

The first set of generic algorithms to look at are the `SimpleFixedPointTraversal` algorithms. These compute the repetitions of a function for execution of the `FixedPoint`, `Upto`, and `Exactly` expressions of the ADT. They are labelled as "Simple" because they do the search from a single root. They carry out search mutably, and convert their output to an immutable set upon returning.

Exactly The simplest algorithm is for computing `Exactly` query nodes. Here, we simply expand a fringe set of values outwards, by applying the search step to every value in the fringe to get the new fringe. We also memoise the search step function in the case

of repetitions. After the required number of repetitions, the remaining fringe is returned. An illustration can be found in figure 3.1.

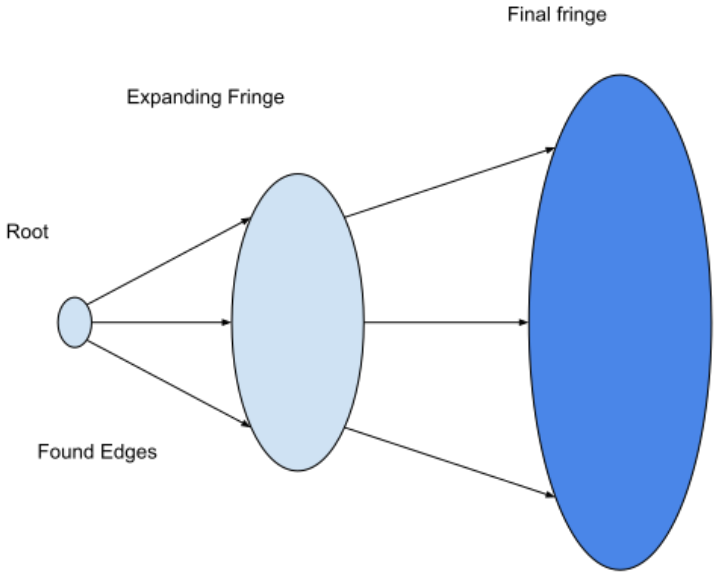

Figure 3.1: Execution of a simple `Exactly` query

Upto The next algorithm is for computing `Upto`. This is computed in a similar way by flat-mapping the functions over the fringe repeatedly to calculate an expanding fringe. The major differences here are that we keep an accumulator of all the found values. When a new fringe is calculated, we subtract the accumulator set from it to reduce the number of nodes that need to be searched to those that have not yet been searched, and then union the remaining fringe with the accumulator to get the new accumulator. After the required number of repetitions, the accumulator is returned. An illustration can be found in figure 3.2.

FixedPoint The final algorithm is to calculate `FixedPoint`. This works slightly differently. As before, we keep an accumulator of reached nodes, but unlike before, the generation number of each node is not important, only that a node is reachable is important. Hence, the fringe is a queue rather than a set, and we iterate until the fringe is empty. In each iteration, we pop off the top value of the fringe queue, compute all immediately reachable nodes. The difference of this reachable set is taken with the accumulator to find

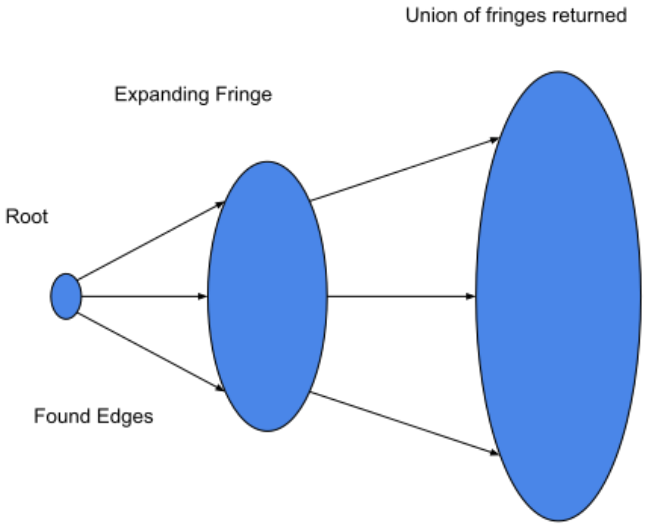

Figure 3.2: Execution of a simple `Upto` query

the newly reached nodes. These are now added to the fringe queue and the accumulator. When the fringe is empty, we return the accumulator.

Full traversal

The next set of algorithms build on the `SimpleFixedPoint` algorithms to return not just those nodes reachable from a single root, but the set of all reachable pairs with the left hand pair derived from a root set (using the left-hand optimisation explained in section 3). As such, the algorithms need to do some more work to reconstruct which nodes are reachable from each root, while still eliminating redundancies.

Exactly The first such algorithm is for computing `Exactly`. This is similar to the original version, except we now store a fringe for each root in a map of `Root => Set[Node]`. We also memoise the search function in a Map to avoid computing it redundantly. In each iteration, we simply expand the fringe for each root as before by mapping the fringe expansion loop body over the values of the fringe map. An illustration can be found in figure 3.3.

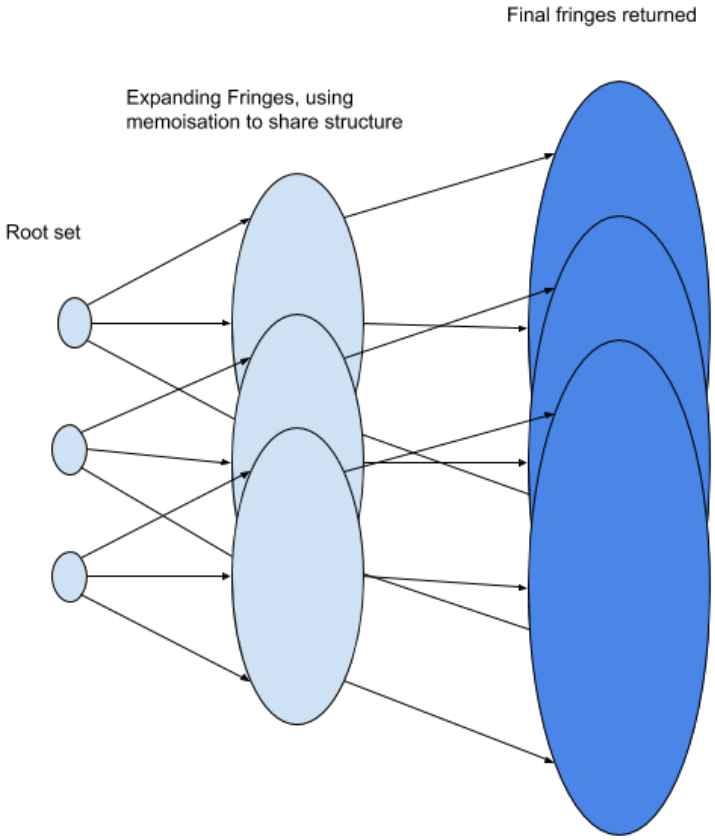

Figure 3.3: Execution of an `Exactly` query from multiple roots

Upto The next algorithm is to compute `Upto`. This is again accomplished like before, but with a map of root to accumulator set as the accumulator. As with `Exactly`, the fringes of each root are expanded simultaneously, sharing redundant results via memoisation, while the accumulators are unioned with the fringe of the appropriate root with each iteration. An illustration can be found in 3.4.

FixedPoint Finally, the `FixedPoint` implementation takes a departure from the parallel implementations above. The reachable set of each node is calculated sequentially using a similar algorithm to above. However the cached memo now contains all nodes reachable from previously processed roots, allowing for fast convergence of dense graphs.

Pathfinding

The `ShortestPath` and `AllShortestPaths` commands require us to search a subgraph generated by a relation. Since all edges have unit weight, the pathfinding algorithm reduces to breadth-first-search, which I have implemented in an imperative format while wrapping

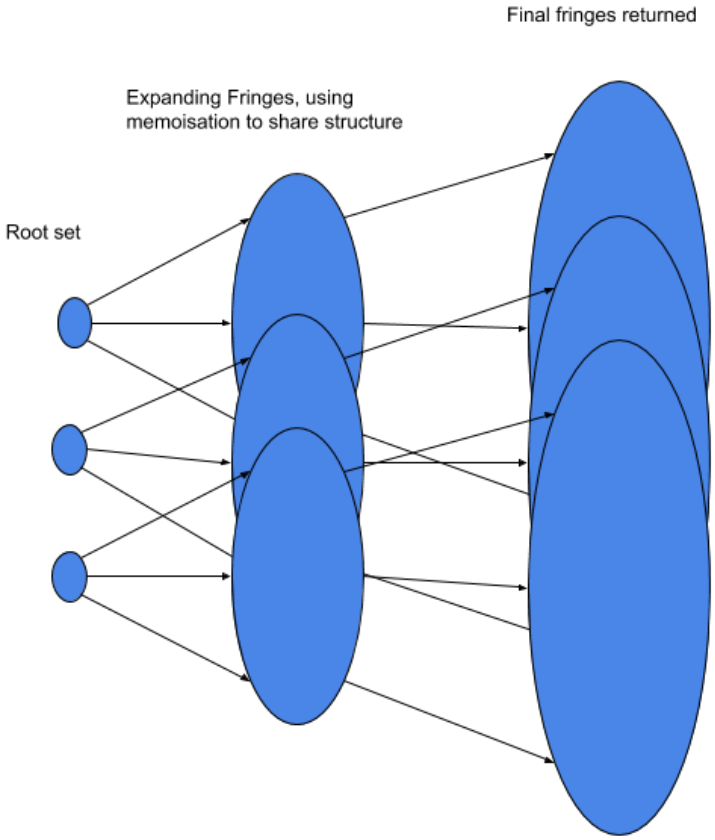

Figure 3.4: Execution of an `Upto` query from multiple roots

up error cases in an Either. These functions take, as a parameter, the search step `A ⇒ E \/Set[A]` which represents the edges going out of a node.

Joins

The problem of joining two sets of pairs based on shared intermediate values is a requirement for any back-end.

$$A\ join\ B = \{(a, c) \mid \exists b.(a, b) \in A \wedge (b, c) \in B\}$$

I have implemented a simple hash-join**?**. This operates by first assuming that the size of distinct left-most elements in the right set is smaller than the right-most elements of the left set. Next we build a hash-map index of the right set. We then traverse the left set, looking up right-most-values in the index, to complete the join. An issue with this approach, addressed later, is that we have to build the index upon each `join` call.

```
def joinSet[A, B, C](
    leftRes: Set[(A, B)],
```

```
    rightRes: Set[(B, C)]
): Set[(A, C)] = {
  // build an index of all
  // values to join on right, since Proj_B(right)
  // is a subset of Proj_B(Left)
  val collectedRight = mutable.Map[B, mutable.Set[C]]()
  for ((b, c) <- rightRes) {
    val s = collectedRight.getOrElseUpdate(b, mutable.Set())
    s += c
  }

  for {
    (left, middle) <- leftRes
    right <- collectedRight.getOrElse(middle, Set[C]())
  } yield (left, right)
}
```

Views and Commits

In the memory back-end, as will be explained later, views are easy implement as a map of `ViewId` to `MemoryTree` and simply updating the `MemoryTree`, allowing Scala's immutable collections to handle sharing of data in an efficient way. In the other back-ends, backed by non-functional technologies, we need other methods of sharing and inserting to immutable structures. A first observation is that within the operation monad model, each view only has one direct predecessor. We can think of the database as a tree of views with edges representing the successor relation. The path from the root view to a given view can be seen as the operations that generated the view. In this model, if the parent is known, then a child may be defined by the differences between the two. As the database only allows for updates and not deletions, this difference will always be a collection of relations and objects that were added. This collection can be wrapped up in a container called a `Commit`. At this point, we can discard the ancestors of a view and see a view as the collection of commits that generated it. This model also allows deletions and overwrites if desired; a commit containing deleted values may be replaced with a new commit which differs only by not including the deleted values.

Memory Back-end

The first back-end that I have implemented is a simple, naive memory-based back-end. This back-end follows the denotational semantics, and makes very few attempts to improve performance. This back-end serves as a test-bench back-end, used to create unit tests to test other back-ends. It also allowed me to practice implementation of type erasure and unerasure according to the schema in a controllable environment (that is, without interference from other languages or libraries as in the SQL and LMDB back-ends.)

Table Structure

A memory instance stores a concurrent map of `ViewId` to `MemoryTree`, which itself is a map of `TableName` (derived from the `SchemaDescription`) to `MemoryTable`. There is a `MemoryTable` for each object class in the `SchemaDescription`. A `MemoryTable` provides lookups using maps for `DBObjects` and `Findable`s, in the form of an index to set of objects for each column value. These lookups return objects containing a `DBObject` and the outgoing and incoming relations for the object, indexed by `RelationName`.

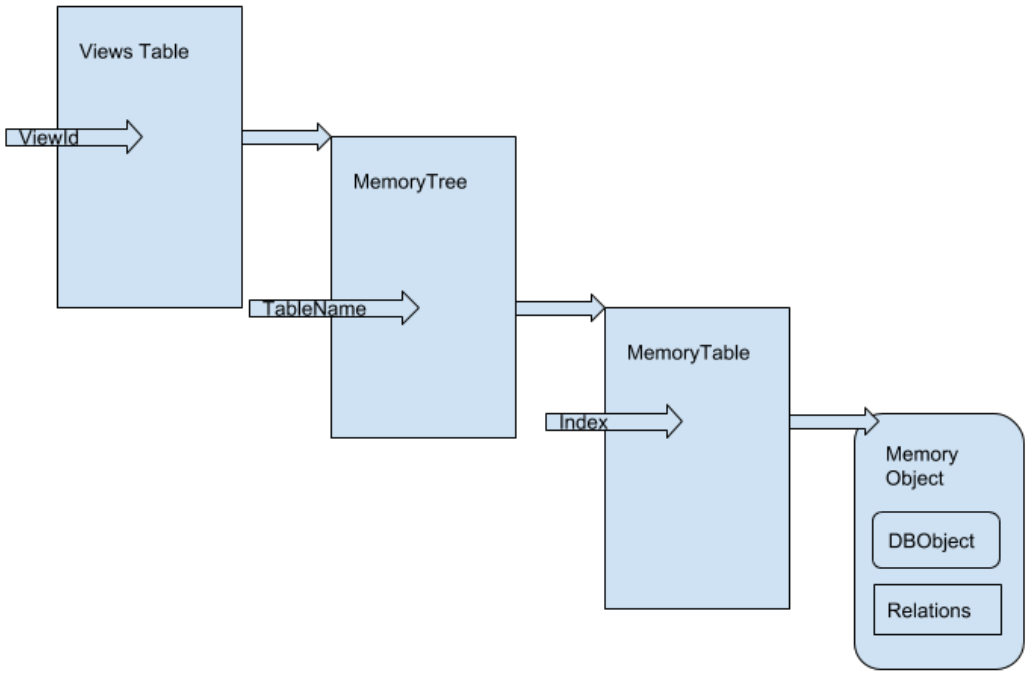

Figure 3.5: The lookup procedure for objects in the memory implementation

Reads

Reads occur by simply walking over the ADT recursively, following the denotational semantics. The only real departures from the denotational semantics are the left-optimisation (see section 3) and use of the generic fixed point algorithms to compute `Exactly(n, P)`, `Upto(n, P)`, and `FixedPoint(P)`. Results are also computed in the `Either` monad to allow for error checking (for cases such as missing tables). This interpreter can be found in appendix **??**.

Left Optimisation

One of the few optimisations here is the left-optimisation. When we compute the result set of a `FindPair`, we pass in the subset of left hand side variables we want to compute from. This mostly has an effect when we compute joins (`Chain`s). Consider joining a query of a few dozen unique right-most values to a large query with several million unique left-most values. The pairs of the right relation only occur in the resulting join if their left-most value is in the set of right-most values of the left relation. Hence pairs which do not have a correct left-most value need not be computed, greatly pruning the search of more complex queries. This pattern also makes an appearance in the original LMDB implementation.

A	B
1	1
2	3

Results of a small query.

$\bowtie$

B	C
1	4
2	3
3	5
4	6
7	8

A larger query.

$=$

A	C
1	4
2	5

Join of the queries.

A	B
1	1
2	3

$\bowtie$

B	C
1	4
3	5

The large query, left-optimised.

$=$

A	C
1	4
2	5

Join of the queries is unaffected.

Figure 3.6: Joining a large right-hand-side to a small left-hand-side without pruning the search of the right-hand-side results in redundant calculations.

Writes

Thanks to Scala's immutable collections library, updating the database is fairly easy. When inserting a collection of relations, we simply add, relation by relation, each object in the relation, if it does not exist, and update the outgoing and incoming relation map of each object. This update creates new immutable object tables and a new memory tree. This is stored to the map of `ViewId` to `MemoryTree` as a new view.

Storage

Objects are stored as `DBObject`s and their incoming and outgoing relations in wrapper `MemoryObject`s. `MemoryObject`s are hashed and compared by their `DBObject`s.

Mutability

The only mutability in the Memory implementation is for the views counter and the views lookup table. Access is kept transactional using a lock.

Pathfinding and fixed point traversal

The pathfinding and fixed point traversal methods are simply implemented by calls to the appropriate generic algorithms using the interpreted findPairs function as the search function.

PostgreSQL back-end

Upon completion of the initial memory back-end, I started work on a PostgreSQL based back-end. This compiles the ADT intermediate representation into SQL queries that are then executed by a Postgres database.

Table Structure

The construction of the underlying database uses several SQL tables. These can be partitioned into a set of control tables that will be present in all database instances and a set of schema defined tables.

Control Tables

Name	Schema	Description
Default	(ViewId)	Holds the mutable default ViewId
Commits Registry	(CommitId^{primary}, Dummy)	Stores which commits are valid, allows us to give each commit a unique id
Views Registry	(ViewId^{primary}, Dummy)	Stores which views are valid, allows us to give each view a unique id
Views Table	(ViewId, CommitId)	Stores which commits belong to which view.

Note: `ViewId` is a foreign key into the views registry, `CommitId` is a foreign key into Commits Registry. The Dummy column is needed to fix postgresql syntax when we try to get the next key for a table with one column.

Schema defined Tables

Name	Schema	Description
Object Table	(ObjectId^{primary}, generated)	Generated per schema object in schema description. A schema column is generated per schema object field, with an appropriate SQL type. This allows indexing by findables to get object Ids for queries, and extraction of objects from the database.
Auxiliary Table	(ObjectId, CommitId)	One auxiliary table is associated with each object table. It stores the object Ids associated with each CommitId, which is a many-many relation
Relation	(LeftId, CommitId, RightId)	Generated per relation in schema description. Each is associated with a commit.

Note: Object-, Left-, and Right- Ids are foreign keys to the `ObjectTable`'s ObjectId column.

Query Structure

In order to manage the complexity of queries, I have made use of SQL's Common Table Expressions feature. CTEs act like let expressions in ML **?**.

In this example, I give the compiled result of

```
FixedPoint(Rel(Knows))
```

The first CTE used selects all the valid CommitIds.

```
WITH RECURSIVE VIEW_CTE AS
  (SELECT $commitId FROM $viewTable WHERE $viewId = $id)
```

The next collection of CTEs select the valid instances of each relation from the relation tables.

```
WITH Relation0 AS (SELECT left_id, right_id
FROM REL_0_0 JOIN VIEW_CTE ON REL_0_0.CommitId = VIEW_CTE.CommitId)
```

Valid instances are selected from the auxiliary tables to be returned from Id_A queries.

```
AuxTable2 AS (
  SELECT DISTINCT obj_id AS left_id, obj_id AS right_id
  FROM AUXUSERSPACE_People_0 JOIN VIEW_CTE
  ON AUXUSERSPACE_People_0.CommitId = VIEW_CTE.CommitId
)
```

CTEs are then created for any repeated subqueries (i.e. those used by *FixedPoint*, *Exactly*, and *Upto*)

```
View1 AS (SELECT * FROM Relation0)
```

Repetitive queries are built using inbuilt SQL constructs.

```
  View3 AS (
    (SELECT left_id, right_id FROM AuxTable2 )
    UNION
    SELECT View3.left_id AS left_id, View1.right_id AS right_id
    FROM
    View3 INNER JOIN View1 ON View3.right_id = View1.left_id
  ),
```

A CTE is created for the main query, which exposes a `leftId` and `rightId` column

```
main_query AS (SELECT left_id, right_id FROM View3 )
```

Finally, the main query CTE is joined to fragments which extract the left and rights.

```
(
  SELECT left_table.col_0 AS left_col_0,
         right_table.col_0 AS right_col_0
  FROM (
    (
      USERSPACE_People_0 AS left_table
      JOIN main_query
      ON left_table.obj_id = main_query.left_id
    ) JOIN USERSPACE_People_0 AS right_table
    ON right_id = right_table.obj_id
  )
)
```

The `main` query itself is built compositionally, in that each subquery exposes a `leftId` and `rightId`, allowing queries to be composed to form larger queries in a standard form. A full query might look like this:

```
WITH RECURSIVE
  VIEW_CTE AS (SELECT CommitId FROM VIEWS_TABLE WHERE ViewId = 1),
   Relation0 AS (SELECT left_id, right_id
    FROM REL_0_0 JOIN VIEW_CTE ON REL_0_0.CommitId = VIEW_CTE.CommitId
  ),
   AuxTable2 AS (
    SELECT DISTINCT obj_id AS left_id, obj_id AS right_id
    FROM AUXUSERSPACE_People_0
    JOIN VIEW_CTE
    ON AUXUSERSPACE_People_0.CommitId = VIEW_CTE.CommitId
  ),
   View1 AS (SELECT * FROM Relation0),
   View3 AS (
    (SELECT left_id, right_id FROM AuxTable2 )
    UNION
    SELECT View3.left_id AS left_id, View1.right_id AS right_id
    FROM
    View3 INNER JOIN View1 ON View3.right_id = View1.left_id
  ),
   main_query AS (SELECT left_id, right_id FROM View3 )
 (
```

```
  SELECT left_table.col_0 AS left_col_0,
         right_table.col_0 AS right_col_0
  FROM (
    (
      USERSPACE_People_0 AS left_table
      JOIN main_query
      ON left_table.obj_id = main_query.left_id
    ) JOIN USERSPACE_People_0 AS right_table
    ON right_id = right_table.obj_id
  )
)
```

Monadic Compilation

In order to generate these SQL queries, we need to convert the ADT query to a lower level intermediate representation for each of the SQL fragments to be created. While doing this, we need to gather and rename all of the relation and auxiliary tables that we extract from so that we can form the CTE queries, we also need to find repeated queries to hoist out. In order to do this, we use the monadic compiler pattern described in section 3. The compiler state is shown below. When the compilation is complete, depending on the context of the command, we append different extraction queries. For find pairs we need to extract the fields of both the left hand side and the right hand side objects, whereas for pathfinding operations, we only need to extract the `ObjectId`s along the path as opposed to whole objects.

```
case class CompilationContext(
            // Unique identifier for variable names
        varCount: Int,
            // Relations used by the query
        relations: Map[ErasedRelationAttributes, VarName],
            // tables needed for Findables in the query
        requiredTables: Map[TableName, VarName],
            // the aux (commit) tables needed for the query
        requiredAuxTables: Map[TableName, VarName],
            // Expressions to be repeated over (e.g. by Upto)
        commonSubExpressions: Map[SubExpression, VarName]
)
```

Writes

There are several steps in the implementation of writes, though I have not expended a great deal of effort making them fast. A significant part of this is the "insert or get" SQL

query, which looks up an object in the relevant table, returning its `ObjectId` if it exists and creating the object and returning the new ID if it does not.

```
WITH insertOrGetTemp AS (
  INSERT INTO $objectTable ($columnNames)
  SELECT $values WHERE NOT EXISTS (
    SELECT 0 FROM
    $objectTable WHERE $ValueConditions
  ) RETURNING $objId
) SELECT * FROM
  insertOrGetTemp
    UNION ALL (
      SELECT $objId FROM $name
      WHERE $ValueConditions
    )
```

We create a new view and commit, then we run a memoised `InsertOrGet` over all the leftmost objects to be inserted, then all the right objects to be inserted, yielding tagged relations between `ObjectIds`. For each inserted relation, the existing relation instances are removed from those to insert, and the remaining inserted to the relevant `RelationTable` with the correct `CommitId`. The auxiliary tables are now updated and, on success, the views table is updated.

Mutability

As with the memory back-end, all mutability except for the availability of views and the default view is hidden from the user. The SQL back-end uses commits to manage view mutability.

Pathfinding and Fixed Point Traversal

Pathfinding is implemented by constructing an SQL query to generate right-hand-side `ObjectIds` for a relation given a left-hand-side `ObjectId`. This query is used as the search function for the generic pathfinding algorithms. Once paths have been found, their `ObjectIds` are looked up in the database to find the full values along each path.

Fixed point traversal and repetitions are performed natively in SQL. `FixedPoint` and `Upto` are achieved using a recursive CTE, while `Exactly` is executed by explicitly joining together the required number of repetitions of the sub-relation's query.

```
WITH RECURSIVE
    RecursiveCTE AS (
```

```
    (SELECT *, 0 AS Counter FROM BasisCase )
    UNION
    SELECT
    RecursiveCTE.left_id AS left_id,
    InductiveCase.right_id AS right_id,
    Counter + 1 AS Counter
    FROM RecursiveCTE INNER JOIN InductiveCase
    ON RecursiveCTE.right_id = InductiveCase.left_id
    WHERE Counter < Limit
   )
```

An example of a recursive CTE computing `Upto`. `FixedPoint` *would omit the counter variable and the limit.*

Object Storage

Objects are stored as in the appropriate `ObjectTable` in a manner derived from the `DBObject` of each object. Each `DBCell` is converted to appropriate SQL type.

LMDB Back-ends

The final family of back-ends are the LMDB back-ends. LMDB is a lightweight, efficient memory mapped file based Key-Value datastore[2].

Common

Although I have written several versions of the LMDB back-end, there are several over-arching similarities.

LMDB API

Terminology LMDB terminology differs slightly from that of more mainstream systems. What would be called a database in SQL is known as an `Env` and a table known as a `DBI`.

Transactions LMDB's transactional model differs from more traditional systems. In short, we need to create a transaction to do read but not write to the database. This stems from LMDB delegating as much work as possible to the OS kernel's memory-mapped file system. All writes are immediately written to the database, whereas starting a read transaction gives a view of the `Env` as it was at the start of the transaction, ensuring that concurrent writes do not affect what is read from the database. Writes may be included in

[2]https://symas.com/lmdb/

transactions to allow atomic get-and-sets. In practice, the transactions used are very short lived.

JVM API The LMDB back-end uses the `LMDBJava` API[3] to allow access to an LMDB Database from JVM languages (such as Scala). For each `DBI`, we have a `ByteArray` to `ByteArray` key-value map, much like a `Map[Array[Byte], Array[Byte]]`.

Storage and Keys In order to write to the database using the LMDBJava API, we need to write the key and value to `ByteBuffer`s. To generify this process I have introduced two type-classes: `Keyable` and `Storable`.

```
trait Keyable[K] { def bytes(k: K): Array[Byte]}

trait Storable[A] {
 def bufferLength(a: A): Int
 final def toBuffer(a: A): ByteBuffer =
    {/*Allocates a buffer and fills it using declared methods*/}
 def writeToExistingBuffer(a: A, buf: ByteBuffer): Unit
 def fromBuffer(buf: ByteBuffer): LMDBError \/ A
}
```

`Keyable` is a type class which shows that an object can be converted into an array of bytes representing a component of a key. Components may be concatenated to form a key. `Storable` is a more general type-class that shows that objects can be converted into a `ByteBuffer`. `Storable` objects are typically more complex than objects used as keys. Furthermore, a higher marshalling throughput is required for such objects, so a lower-level, `ByteBuffer` based API is used. Typeclass instances exist to allow sets and lists of `Storable` objects to be stored.

Table Structure

Similarly to the SQL implementation, there are several tables, some generated based on the `SchemaDescription` and other, control, tables exist regardless. Each table is represented in memory by a subclass of the `impl.lmdb.common.tables.interfaces.LMDBTable` trait, which provides utility methods such as transactional reads and computations.

[3] http://github.com/lmdbjava/lmdbjava

Name	Key Type	Value Type	Description
Object Counter	Singleton	ObjId	Atomic counter holds the next ObjectId
Commit Counter	Singleton	CommitId	Atomic counter holds the next CommitId
View Counter	Singleton	ViewId	Atomic counter holds the next ViewId
Default table	Singleton	ViewId	Mutably stores the default view
Available Views	Singleton	Set[ViewId]	Mutably stores a set of available views
Views Table	ViewId	List[CommitId]	Maps a view to its constituent commits
Relations	(ObjectId, CommitId, RelationName)	Set[ObjId]	Single table stores all forward relations in a database
Reverse Relations	(ObjectId, CommitId, RelationName)	Set[ObjId]	Single table stores the reverse of all relations in the database

Singleton Keys denote tables with single key.

Name	Key Type	Value Type	Description
Retrieval Table	ObjectId	DBObject	Used to retrieve object data at end of a query
Empty Index	CommitId	ObjectId	Stores object Ids accessible a view. Used to compute Id_A.
Column Index	(Commit, DBCell)	ObjectId	Provides an index over a field of a SchemaObject

For each `SchemaObject` in the `SchemaDescription`, there exists a `RetrievalTable`, `EmptyIndexTable`, and a `ColumnIndexTable` per column in the `SchemaObject`.

Writes Writes occur in a similar way to the SQL implementation; views are managed using commits. First, we look up all the left- and right-hand-side objects to find their `ObjectIds`. (If needed, we create new entries in the retrieval table and update the index tables to insert the objects.) We then look up existing instances of the relations for insertion and only write novel ones to the relation table with the new `CommitId`. Upon success, we create a new view and insert it to the views table and the available views table.

Original LMDB Implementation

impl.lmdb.original

The original LMDB back-end implementation executed queries in much the same way as the in-memory back-end, the only difference being the flat LMDB table structure. Interpretation occurs by a very similar function to that of the memory back-end, instead passing around a list of `Commit`s to search rather than the memory tree. At the end of a query's execution, the resulting `ObjectId`s are looked up in the relevant retrieval table to extract the actual objects which are converted into user objects. Pathfinding and the repetition operators are handled in the same way as the memory back-end by calling the relevant generic algorithms.

Batched

impl.lmdb.fast This is the first variant upon my simple original LMDB back-end, it focuses on making small local optimisations rather than significant algorithmic changes.

Read Batching Firstly, the original implementation made sub-optimal use of LMDB reads. For example, when extracting objects at the end of query execution, it would look up each object individually with its own read transaction. This is sped up by batching together reads from separate keys in the same transaction.

Pre-Caching The result of executing a `FindSingle` is constant throughout the whole query, so by lifting out and pre-emptively executing any `FindSingle`s, we speed up execution by removing redundancy. More importantly, this helps enable the next operation.

FindFrom For the pathfinding commands, the transitive queries and `From FindSingle` query, we are not actually interested in finding all pairs matching the query, but the reachability function $A \Rightarrow Set[B]$. This function forms a kleisli arrow which takes a node and finds its directly reachable neighbours. Since little bookwork is needed to maintain the left hand side of any relation, this greatly simplifies the search. Hence, in these cases, we interpret the query using an alternative interpreter, the `FindFrom` method. For example, joins now become a concatenation of these arrows (`flatMap` in Scala parlance), `Id`

becomes the *return* function $(x \Rightarrow Set(x))$, `And`, `Or`, `Distinct` all become lower complexity methods, and, finally, we can make use of the `SimpleFixedPointTraversal` algorithms described in section 3. The pre-caching in the previous section is useful here, since due to the `flatMap`s, the computation of `FindSingle`s would be repeated many times otherwise.

Common Sub-Expression Elimination

impl.lmdb.cse A common performance issue in the previous implementation was that of redundant computation and lack of general common sub-expression elimination (CSE). When a common sub-expression is repeated outside of an `Upto` or `Exactly` block, then it is recalculated, leading to poor performance. In addition, there is significant redundancy exposed by the above `FindFrom` optimisation. Consider the calculation of $Rjoin(SjoinT)$. If a node n appears in the results of $S(m)$, $S(m')$ for distinct $(?, m) \triangleleft_{A,B,v} R$, $(?, m') \triangleleft_{A,B,v} R$ (m, m' are right hand results of R), then $T(n)$ is computed redundantly in the overall calculation. These issues of redundancy can be solved using a combination of global CSE and memoisation.

The Memoisation Problem When looking at a graph data structure, memoisation is harder than when looking to optimise pure functions. To optimise a pure function $f\colon A \Rightarrow B$, we only need to store a hash table `Map[A, B]`. To compute $f(a)$, we simply look up a in the map. If it occurs in the map, we return the mapped value. Otherwise, we call f, and insert the result to the map before returning. In a large graph, computing and storing all pairs generated by a query is extremely wasteful, since many of those pairs might never be used (effectively, we would have ignored the left-optimisation). Conversely, memoizing the full left-optimised function, $interpret(Q)\colon Set[A] \Rightarrow Set[(A, B)]$ is also wasteful, since there may be overlap in the sets used as keys. Instead, we memoise the `FindFrom` function of a query, $findFrom(Q)\colon A \Rightarrow Set[B]$, and reconstruct results from the pairs. This allows for the best overlap. To make the best use of memoisation and to avoid over-using memory by memoizing the same query twice, we need to also apply CSE to group together instances of the same sub-query.

Retrievers A solution to these problems is using an object called a `Retriever`. This exposes two methods which mirror those from interpreting a query.

```
trait RelationRetriever extends Logged {
     def find(from: Set[ObjId]): LMDBEither[Set[(ObjId, ObjId)]]
     def findFrom(from: ObjId): LMDBEither[Set[ObjId]]
}
```

For each subquery node in a query(e.g. *And(Id_A, Or(R_1, R_2))*), we generate a retriever. Retrievers for primitive relations (e.g. *Id_A* , *Rel(R)*) are un-cached, as LMDB allows for very fast re-computations of these. In the other cases, we use a

`CachedRelationRetriever`, which memoises an underlying lookup function. Using type-enrichment, I have implemented methods such as `join`, `and`, `or`, `exactly`, `fixedPoint` on `RelationRetrievers`, allowing them to be composed analogously to the query they are generated from. This memoises every node of the query to be executed, yielding a significant reduction of redundancy.

Monadic Compilation To avoid storing multiple `RelationRetrievers` for the same sub-expression, it is useful to reuse the same retriever for every occurrence of a subquery. Hence, an algorithm to construct a query should keep track of previously seen subqueries. This can be achieved using the monadic compilation pattern (Section 3). The compilation state stores a hashmap of all the subtrees that have already been computed (similarly to constructing a binary decision diagram in automated theorem proving).

```
class QueryMemo(
    val pairs: Map[UnsafeFindPair, RelationRetriever],
    val singles: Map[UnsafeFindSingle, SingleRetriever]
)
```

Compilation state

The compilation function checks the memo for a precomputed retriever for a query. If a matching retriever is not found, it recurses to compute and compose subtree results to produce a new retriever. The new retriever is added to the compilation state.

Complex Common Sub-Expression Elimination

impl.lmdb.fastjoins Despite these optimisations, there are still sources of redundancy.

Index Building Firstly, the original CSE implementation uses the excessively generic `joinSet` function to join result sets. This function wastes time by computing an index on every call. Instead, we can change the definition of a `Retriever`'s public interface to mitigate the need to build indices.

```
trait RelationRetriever extends Logged {
  def find(from: Set[ObjId]): LMDBEither[Map[ObjId, Set[ObjId]]]
  def findFrom(from: ObjId): LMDBEither[Set[ObjId]]
}
```

Indeed, the function to join a pair of retrievers becomes a simpler `flatmap` of one map to another. Functions such as the `union` and `intersection` of retrievers also become simpler. (*See impl.lmdb.fastjoins.retrievers.RelationRetriever.RelationRetrieverOps*)

Exactly This implementation also explores other formulations of the *Exactly(n, P)* query. `Exactly` effectively joins together n repetitions of the underlying query in a sequential fashion. Despite making good reuse of the query `P`, this fails to memoise the repetition of the *join*; each *join* has different input subqueries, so it cannot be memoised.

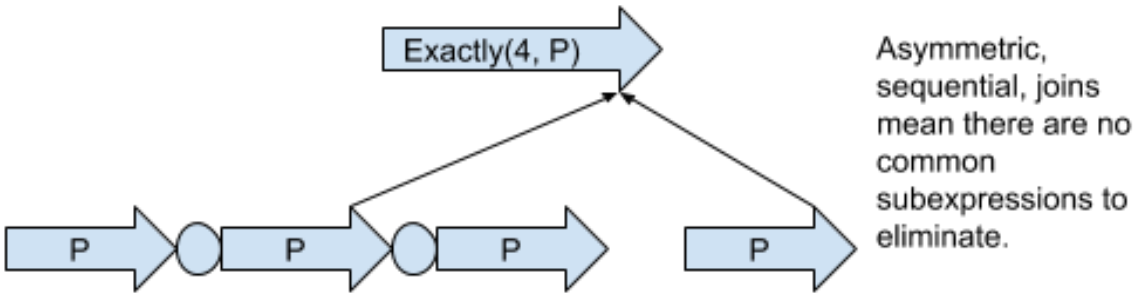

Figure 3.7: Sequential joins provide little common substructure to eliminate.

As joins over the set of results of subqueries form a *Monoid* (proven in appendix **??**), we associatively reorder the evaluation of *joins*, aiming for as much common substructure as possible. This can be performed in a manner similar to square-and-multiply exponentiation.

We calculate retrievers for P^{2^i} for each i less than the bitlength of n. We then join the relevant retrievers to to get a retriever for P^n. This only uses $O(Log_2(n))$ distinct joins, and makes very good reuse of the join functions.

```
def exactly(n: Int): RelationRetriever = repeat(n, outer, IdRetriever)

private def repeat(
  n: Int,
  acc: RelationRetriever,
  res: RelationRetriever
): RelationRetriever = if (n > 0) {
  val newRes = if((n & 1) == 1)  (res join acc) else res
  val newAcc = acc join acc
  repeat(n >> 1, newAcc, newRes)
} else res
```

Upto Optimisation We can also re-formulate an *Upto(n, P)* as an $Exactly(n, Or(P, Id_A))$. (As proven in appendix **??**.) We can hence use the same optimisation as above on *Upto*. We cannot, however produce a similar static formulation for `FixedPoint` as we cannot tractably predict when it will converge.

```
    def upto(n: Int): RelationRetriever =
        repeat(n, outer or IdRetriever, IdRetriever)
```

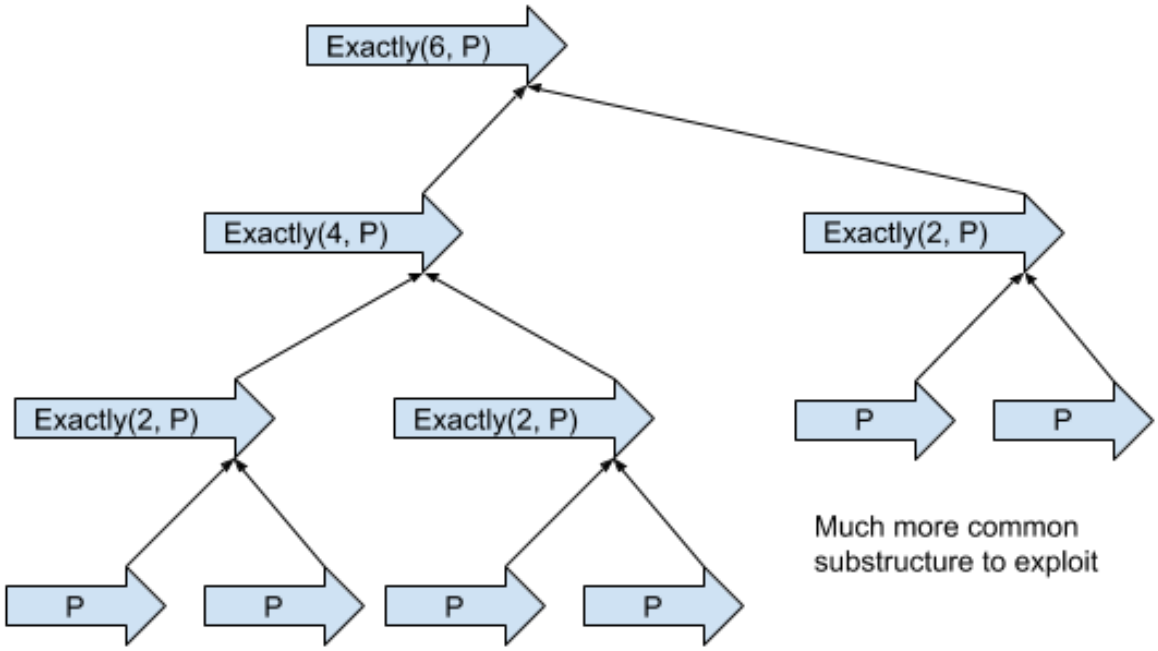

Figure 3.8: By reordering joins, we gain significant common substructure to eliminate.

Summary

This chapter has explained overarching programming techniques and how back-ends were constructed. Furthermore it introduces several optimisations to LMDB back-ends. The first is the simple left-optimisation which prunes search trees for queries which match a small proportion of the graph. The second is elementary CSE-elimination which we expect to provide a speed up over queries with common sub-structure. Finally there is the `Exactly` and `Upto` reformulation which ought to improve performance over transitive queries.

Chapter 4

Evaluation

Unit Tests

The correctness of each back-end is tested using a suite of 45 unit tests which verify adherence to the specified semantics. These have wide range, testing over all the commands, all the possible ADT nodes, and the correct usage and separation of views. I have been using the regression test model, in that discovering a non-trivial bug, I've written a test case to target that bug, ensuring that it is not leaked into production again. These tests provide good evidence that all the implemented back-ends are correct locally.

Performance Tests

In order to evaluate the effectiveness of the various optimisations to the LMDB back-ends described above, it was necessary to run performance tests over wide range of queries. The various LMDB back-ends were tested against each other as a control and particularly against the SQL back-end as a standard to beat. The in-memory back-end was omitted as initial tests indicated that it runs with approximately the same algorithmic characteristics as the original LMDB implementation, which shares the same interpreter model. The LMDB implementation was able to index faster than Scala's tree based hash maps, which are used by the in-memory implementation. Furthermore, the original LMDB implementation is typically around 3x slower than the batched version across most tests, since other than the batched reads from disk, the majority of the algorithms it uses for interpretation are the same.

Hardware

Tests were run on the oslo machine belonging to Timothy Jones' group. Its specifications are shown below. All of the back-ends ran off an SSD.

Processor :	2x OPTERON 16-CORE 6376
Memory :	8x 16G DDR3 1600 ECC Reg Server memory

Datasets

To evaluate tests on non-trivial examples, I sought to construct datasets over which large queries were feasible.

IMDB and TMDB The first and most used collection of tests are derived from the TMDB movie database, the most popular 5000 of which were collected from kaggle **?**

The CSV data was processed using a python script into a simplified JSON format. A separate, larger dataset derived from IMDB was constructed from tests for another graph database **?** and manually (in python) converted to the same JSON format. I then wrote a Scala script which reads the JSON and writes the relations to a given database instance.

Slightly different parameters were given when generating each dataset.

Name	Movies Size	People Size	Notes
smallest	100	10351	100 movies with most actors
small	250	18167	250 movies with most actors
small _sparse	250	1029	250 movies with fewest actors
medium	1000	35626	1000 movies with most actors
medium_sparse	1000	6310	1000 movies with fewest actors
large	12862	50134	Largest dataset

The objects in the database are as follows:

Objects

Name	Fields
Person	Name: String
Movie	Name: String Language: String
Date	Time: Long
Place	Place: String
Genre	Genre: String

Relations

Name	From	To
ActsIn	Person	Movie
Directed	Person	Movie
HasBirthday	Person	Date
BornIn	Person	Place
HasGenre	Movie	Genre

UFC

A second dataset, built from UFC fight data **?**, was constructed in the same way to produce one JSON dataset.

Objects

Name	Fields
Person	Name: String, Height: Int, Weight: Int

Relations

Name	From	To	notes
Beat	Person	Person	
ShorterThan	Person	Person	Transitive reduction of the shorter than relation
LighterThan	Person	Person	Transitive reduction of the lighter than relation

The `ShorterThan` and `LighterThan` relations produce extremely sparse graphs which take a long time to converge under transitive closure. The graph has 1562 nodes and 3571 `Beat` relations.

Test Harness

In order to run tests against each other, I have written a typesafe test harness to run on oslo. This standardises the interface that individual benchmark instances must implement. Each test must provide `setup` and `test` methods which take a `DBInstance` parameter. The test method is further enumerated by a `TestIndex`. The test specification must give a maximum index and a mask to avoid running inappropriate back-ends, such as those that might take too long on a large data-set. The benchmarks that I have run test only the read speed; the time taken to construct the database is not included. Finally, the test harness compares the hash of the result of each query with that produced by the reference

implementation, hence verifying that all implementations produce the same result. This provides good evidence that the implementations follow the same specified semantics. For the sake of the following tests, the **reference** implementation is the most optimised **LMDB Backend**.

Results

Overall Picture An overall view of the results is that, as might be expected, the SQL implementation is the fastest over short, indexing heavy, queries. The most aggressively optimised LMDB implementation typically performed slightly worse than those with fewer optimisations over these queries. This likely due to the overheads of generating the memo. However, when longer graph queries expose the redundancies noted in the preceding sections, speed ups of orders of magnitude can be seen over both the other LMDB implementations and over SQL. This can particularly be seen in the cases of `FindPair` transitive queries.

A point to note is that I am ignoring evaluation of pathfinding queries. For simplicity, the pathfinding implementation re-evaluates the underlying step function query at each stage of the breadth first search. This seems to particularly punish the SQL implementation, to the extent that it is around 2000x slower than the LMDB reference implementation. Stateful solutions to speed this up might include creating a temporary table in the database to store the subgraph being searched. However, this would complicate the SQL implementation, since clearing up on error would require validating state. Hence, I am ignoring the pathfinding tests for the purpose of back-end comparison. As the differences in performance due to the features of different back-ends are relatively large and benchmarks test over a range of queries, I have omitted error bars in performance result graphs.

Redundancy This first test demonstrates the removal of redundancies in `FindSingle` commands. There is clearly a major increase in speed due to the introduction of CSE and memoisation. Since this is a `FindSingle` query, there is no usage of the optimised join formulations, so there is no speed gain by the most optimised LMDB back-end. Postgres fails to fully optimise away the redundancy, even with use of the `DISTINCT` modifier of queries, hence it ends up running out of temporary disk space on this query. Results can be seen in figure 4.1.

Name	Redundancy	
Iterations	4 each	

Queries	`KevinBacon » coactor;`
	
	`KevinBacon » (coactor ->-> (`
	`coactor ->-> coactor));`
	
	`KevinBacon » (coactor ->-> (`
	`coactor ->-> (coactor ->-> coactor)));`
	
	Where `def coactor = ActsIn -><- ActsIn`
	Finds increasing order co-actors with Kevin Bacon	

implementation	time (ms)	/optimised
LMDBOptimised	381,972	1
LMDBCSE	379,280	0.993
LMDBatched	9,318,741	24.4
Postgres	Did not finish	Did not finish

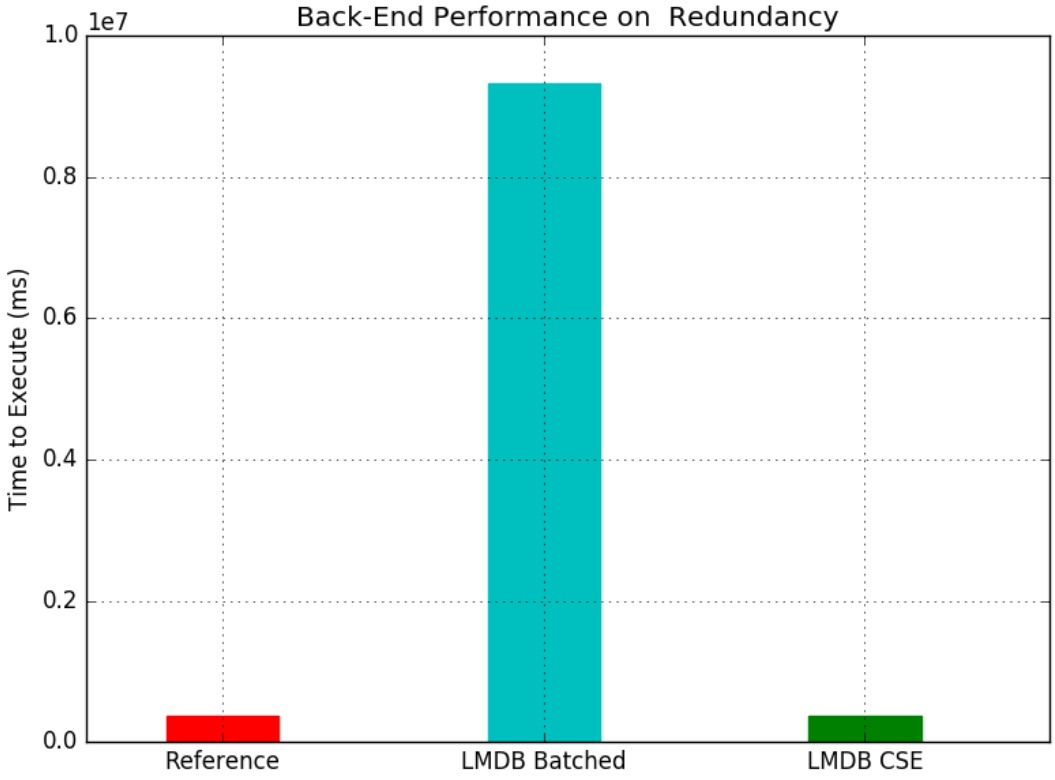

Figure 4.1: Read performance for redundancy queries

Intersections and Unions These two tests make use of the *And* and *Or* operators to combine several subqueries with some underlying common structure. Both resulted in similar performance patterns. The Postgres implementation comfortably outperformed all

LMDB implementations across this index-heavy test, while the CSE related optimisations appeared to add overheads greater than the redundancy they removed. I expect the reasons for this are that the redundancy exposed by the underlying sub queries is limited and that the implementation of unions and intersection in Scala sets is slower than the optimised C implementation used by Postgres. In addition, constructing an intersection and union over the memoisation in the LMDB implementation adds some overhead and reduces locality of reference. Results can be seen in figures 4.2 and 4.3.

Name	Intersections	
Iterations	40 each	

Queries	`(coactorWith(KevinBacon));`
	
	`(coactorWith(KevinBacon) & coactorWith(TomCruise));`
	
	`(coactorWith(KevinBacon) & coactorWith(TomCruise) & coactorWith(TomHanks));`
	
	Where
	`def coactorWith(a: Person) = ActsIn -> (a » ActsIn) <- ActsIn`
	`CoactorWith(a)` defines a relation between actors who are coactors in a movie that `a` also acted in.	

implementation	time (ms)	/optimised
LMDBOptimised	879,984	1
LMDBCSE	827,734	0.941
LMDBatched	791,678	0.900
Postgres	106,165	0.121

Name	Unions	
Iterations	40 each	

Queries	`(coactorWith(KevinBacon));`
	
	`(coactorWith(KevinBacon) | coactorWith(TomCruise));`
	
	`(coactorWith(KevinBacon) | coactorWith(TomCruise) | coactorWith(TomHanks));`
	
	Where
	`def coactorWith(a: Person) = ActsIn -> (a » ActsIn) <- ActsIn`	

implementation	time (ms)	/optimised
LMDBOptimised	853,664	1
LMDBCSE	824,799	0.966
LMDBatched	800,455	0.938
Postgres	112,488	0.132

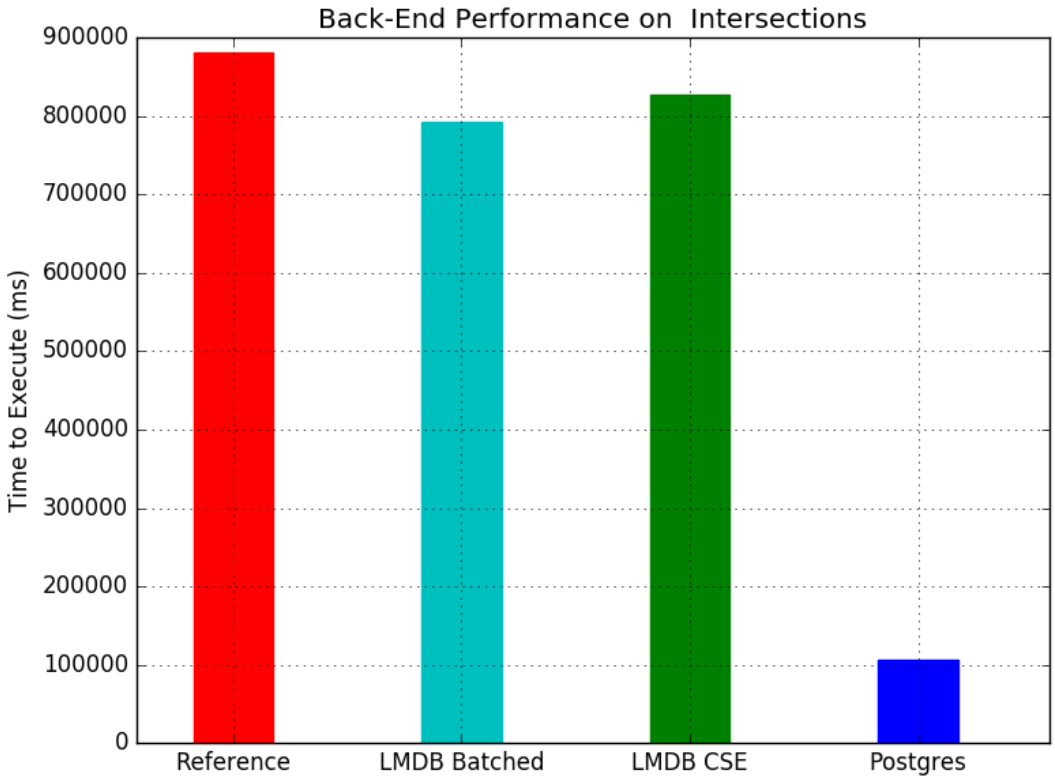

Figure 4.2: Read performance for intersection queries

Tests that involve repetitions These are a suite of benchmarks for testing the optimisations and performance on repetitive queries. A general overview is that the join-reordering optimisations strongly sped up benchmarks which ran `FindPair` queries. However, in `FindSingle` queries this formulation did not compete as strongly with the fast `SimpleFixedPointTraversal` methods. SQL performed strongly in *Upto* queries which use built in, optimised, recursive CTEs to calculate results, but less strongly in *Exactly* queries which do not have a built-in formulation.

Exactly Test This test runs `FindSingle` *Exactly* queries of varying lengths over the small movies database. As explained, the `SimpleFixedPointTraversal exactly` method is faster than any gains by the join reordering in the most optimised LMDB version. Postgres also performed poorly due to the lack of a recursive CTE for self joins. See figures 4.4 and 4.5.

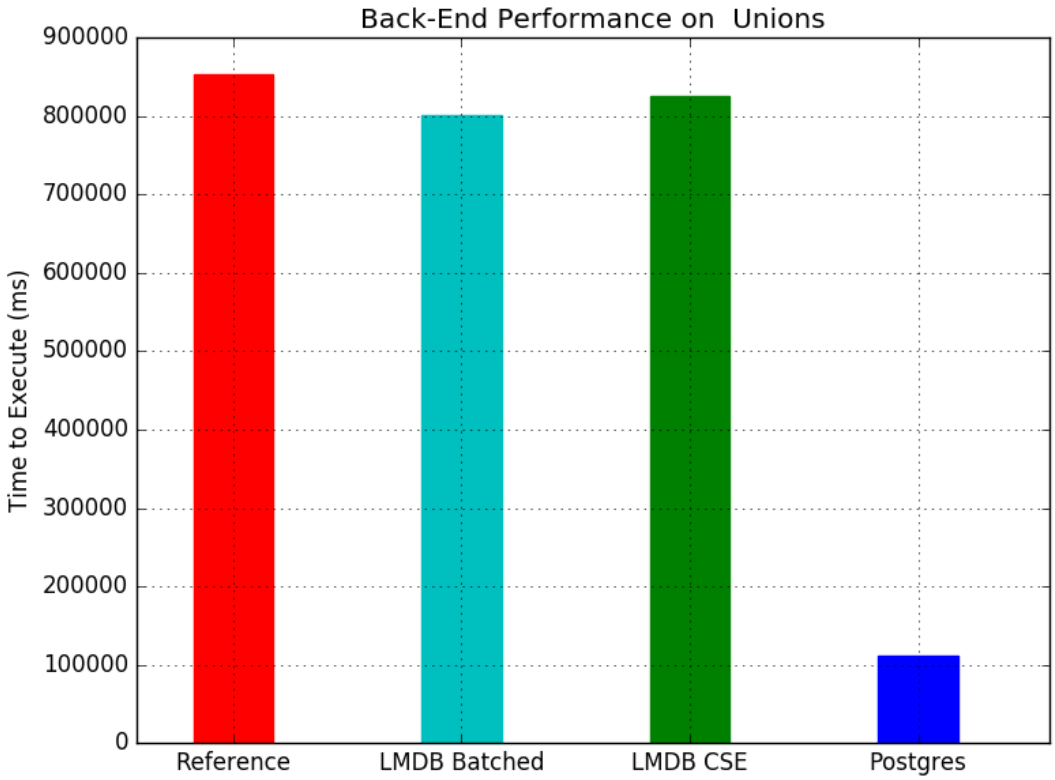

Figure 4.3: Read performance for union queries

Name	Exactly	
Iterations	50	
Queries	`TomHanks » ( ((ActsIn ->(KevinBacon » ActsIn))<- ActsIn) * (index.i % 10); )` Finds higher order coactors of Tom Cruise over the subgraph of movies acted in by Kevin Bacon	
implementation	time (ms)	/optimised
LMDBOptimised	1,752	1
LMDBCSE	1,798	1.03
LMDBatched	1,513	0.834
Postgres	102,461	58.5

Exactly Pairs This test instead runs a `FindPair`, variable-length *Exactly* query over the small movies database. Since this test involves bookkeeping of the root for each pair found, the *Exactly*-optimised LMDB implementation significantly outperforms the other LMDB implementations which use the generic algorithm. Postgres also performs well in this test, showing minimal slowdown from the `FindSingle`queries. The LMDB implementations, however, do experience a slowdown from `FindPair`. See figure 4.6.

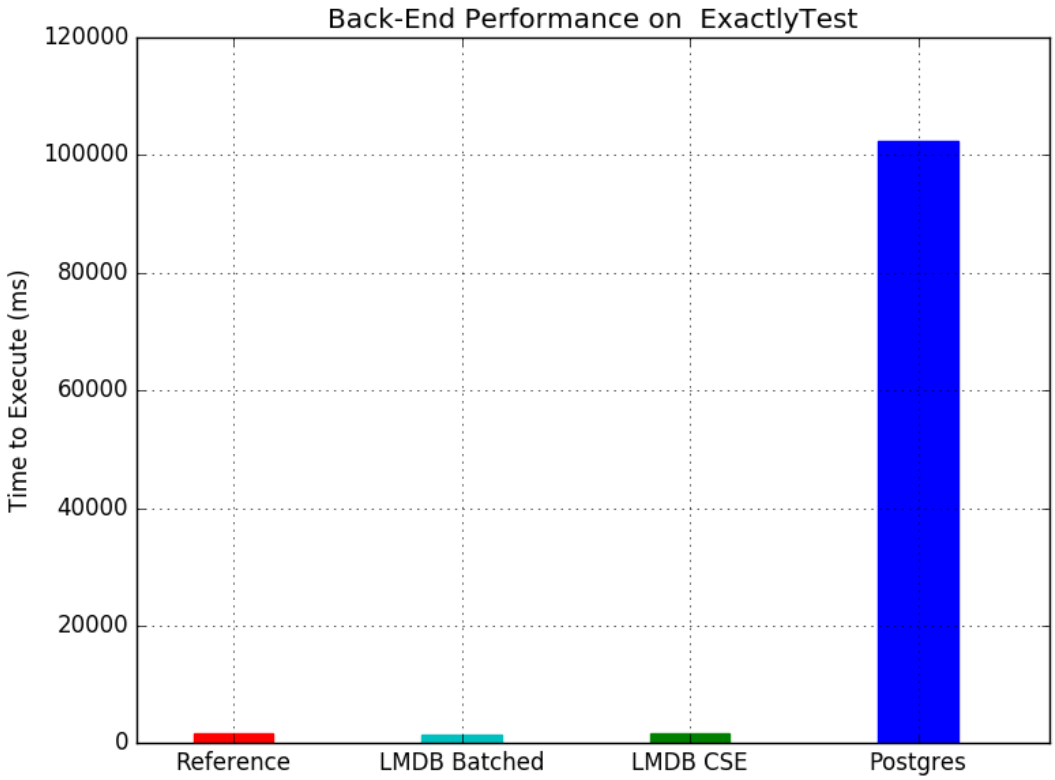

Figure 4.4: Read performance for `Exactly` queries

Name	ExactlyPairs	
Iterations	50	
Queries	`((ActsIn ->(KevinBacon » ActsIn))<- ActsIn) * (index.i % 10));` Finds higher order pairs of coactors over the subgraph of movies acted in by Kevin Bacon	
implementation	time (ms)	/optimised
LMDBOptimised	83,269	1
LMDBCSE	1,792,388	21.5
LMDBatched	1,787,390	21.5
Postgres	107,797	1.29

UptoTest This test runs *Upto* based `FindPair` queries over the large IMDB database. Here the `Upto`-optimisation of the final LMDB back-end provides a significant speed-up, such as 225x on the other LMDB back-ends and 1.5x over Postgres. See figure 4.7.

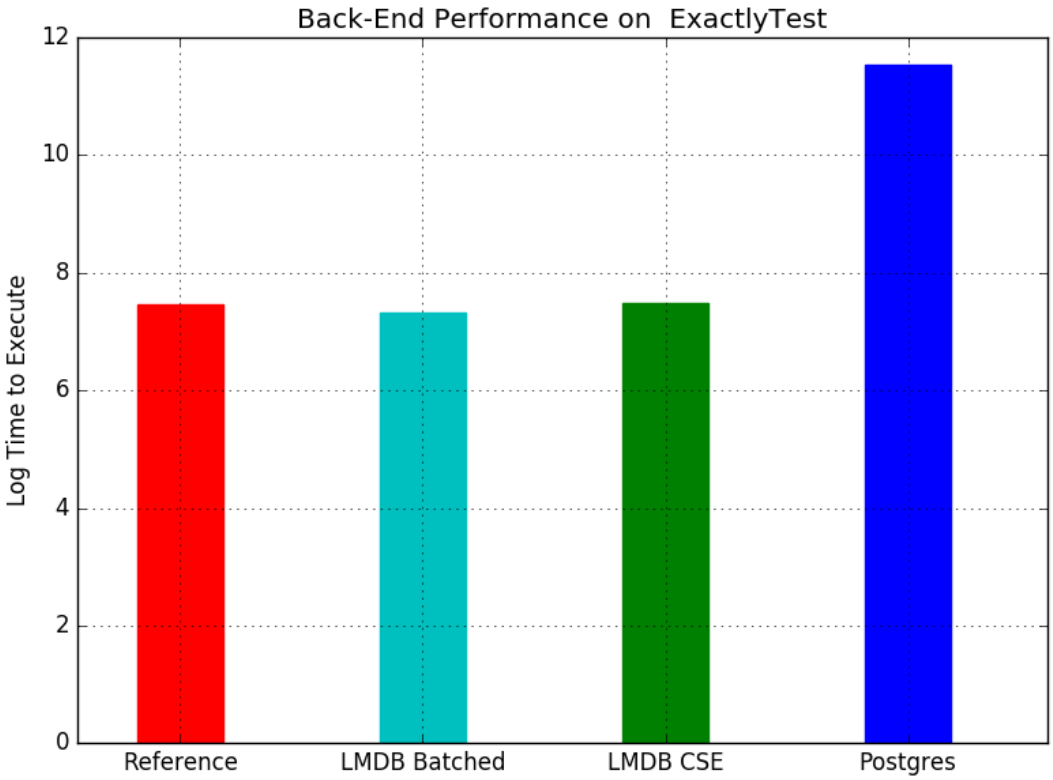

Figure 4.5: Log-Read performance for `Exactly` queries

Name	Upto	
Iterations	10	
Queries	`((ActsIn ->(KevinBacon » ActsIn))<- ActsIn).*( 0 -> index.i);` Finds actors linked by up to i insances of the coactor relationship in the subgraph of movies acted in by Kevin Bacon	
implementation	time (ms)	/optimised
LMDBOptimised	126,448	1
LMDBCSE	28,491,910	225
LMDBatched	28,423,658	225
Postgres	197,880	1.56

The effect of the `Upto`-optimisation can be seen further when we plot time to complete individual queries of `Upto(n, P)` for increasing n. Trends from the same test-run can be seen in figures 4.9 and 4.10.

UptoLarge

Since the more naive LMDB instances took seven hours each to complete the above test, I re-ran the benchmark for only the reference and Postgres implementations for a longer period of time, yielding a similar ratio of performance. These results can be seen in figure 4.11. Furthermore, plotting the results of individual queries (in figure 4.12 as before reveals

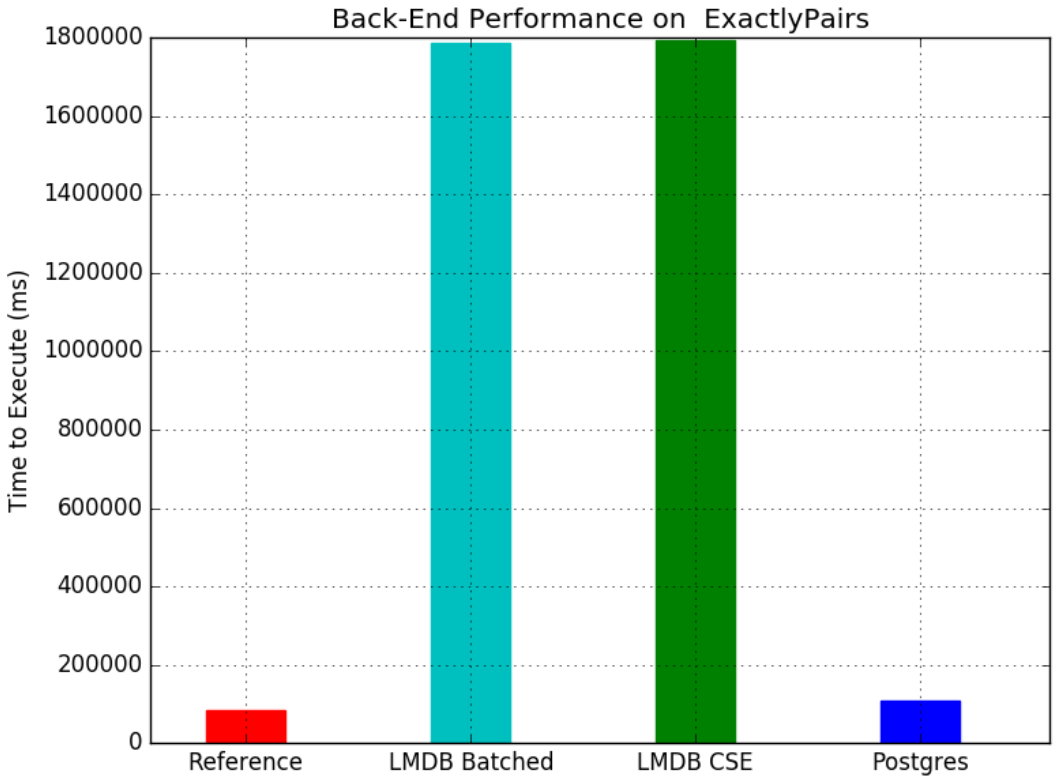

Figure 4.6: Read performance for exactly `FindPair` queries

that Postgres's implementation takes linear time to complete, versus log time for the reference implementation. This makes sense, as the retriever structure built by the LMDB back-end has a depth that grows with the logarithm of the query parameter.

Name	Upto	
Iterations	100	
Queries	`((ActsIn ->(KevinBacon » ActsIn))<- ActsIn).*( 0 -> (index.i % 10) );`	
implementation	time (ms)	/optimised
LMDBOptimised	1,059,091	1
Postgres	1,557,314	1.47

JoinSpeed This test is to demonstrate the cost of joins. We interleave queries that calculate two subqueries and then combine them. For half of the queries, the `And` operator is used, which is assumed to be fast. For the other half, a `Chain` is used. In theory, the difference between these two types of queries should give us an idea of the cost of a join, as seen in figure 4.13.

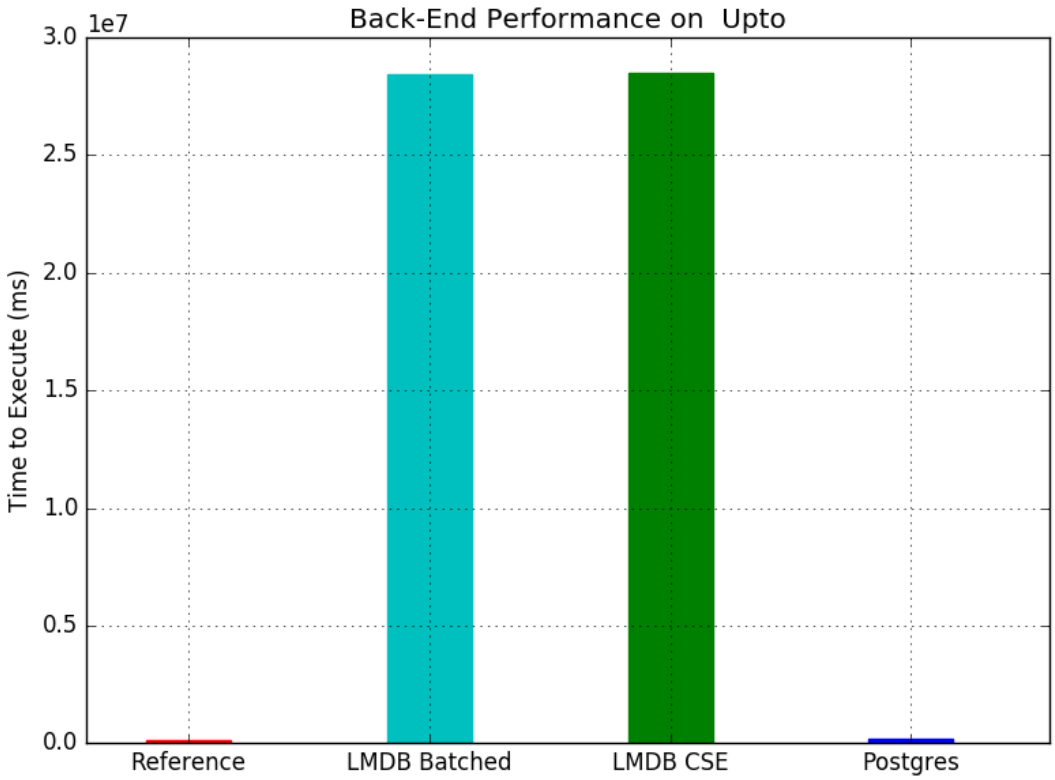

Figure 4.7: Read performance for `Upto` queries.

Name	Join Speed		
Iterations	150 each		

Queries	`(ActsIn -><- ActsIn) & (ActsIn -><- ActsIn);`
	
	`(ActsIn -><- ActsIn) ->-> (ActsIn -><- ActsIn);`		

implementation	median `And` (ms)	median `Join` (ms)	Ratio
LMDBOptimised	633	1,443	2.28
LMDBCSE	884	2,285	2.58
LMDBatched	953	2,028	2.12
Postgres	345	1,557	4.51

There are some things to note from these figures. Firstly, around 7% of the LMDB times are consistently around 1000-2000ms larger than the rest of the values. This can be seen in figure 4.14. I have yet been unable to determine if this is is due to the LMDB datastore itself or if there is another background task interfering. As a result of this bimodal distribution, erroneously large standard deviation values occured. Hence, I have used the median latencies of queries instead of mean. The data suggests that Postgres can execute index and intersection queries very quickly but suffers when join queries are used. The join-optimised LMDB implementation shows itself again to be the best at removing redundancy among the LMDB implementations.

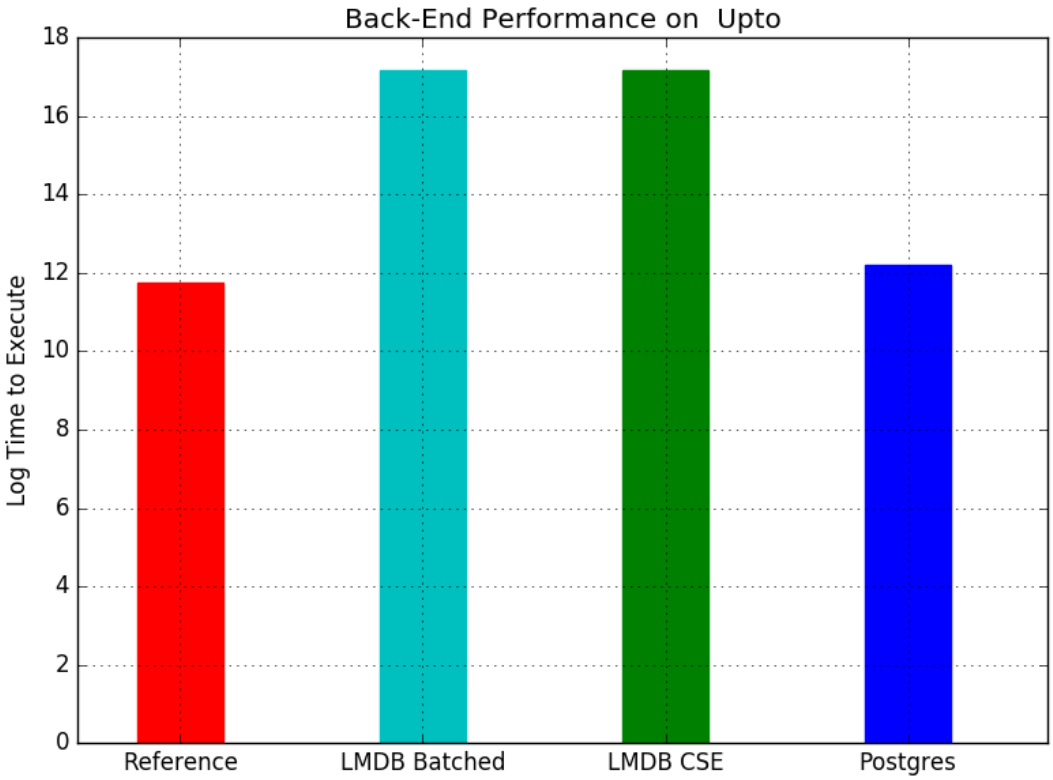

Figure 4.8: Log-Read performance for `Upto` queries.

Closing Thoughts These benchmarks provide some insights into the strengths and weaknesses of the various optimisations I have written. The results lend themselves to the idea of constructing an adaptive back-end that chooses its interpretation strategy based on heuristics of the performance of given queries. These performance statistics also show that specialised graph engines can easily outstrip the performance of generalised relational databases over particular queries within the domain.

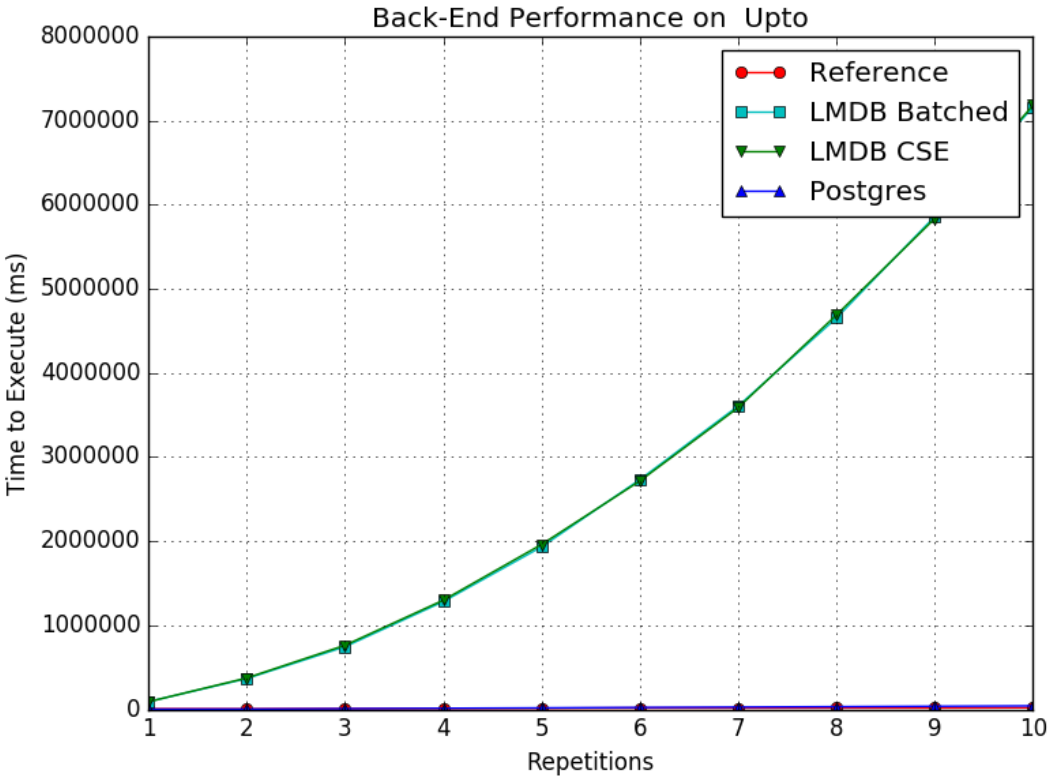

Figure 4.9: Read performance for `Upto` queries with increasing `n`.

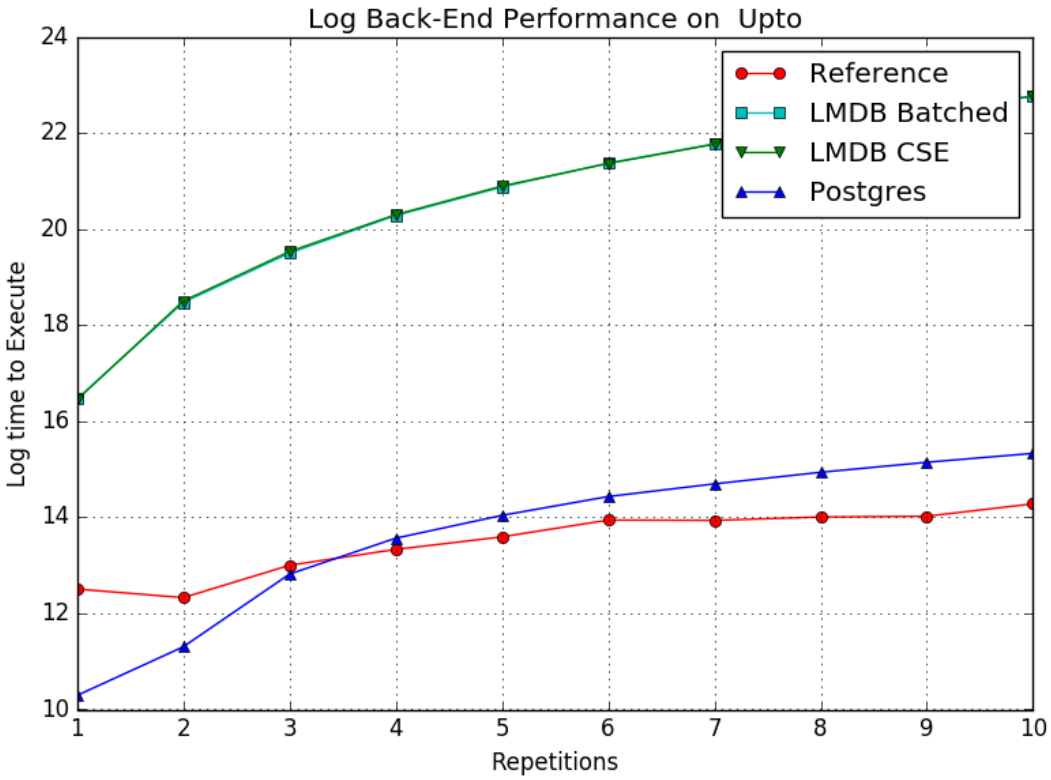

Figure 4.10: Log-Read performance for `Upto` queries with increasing `n`.

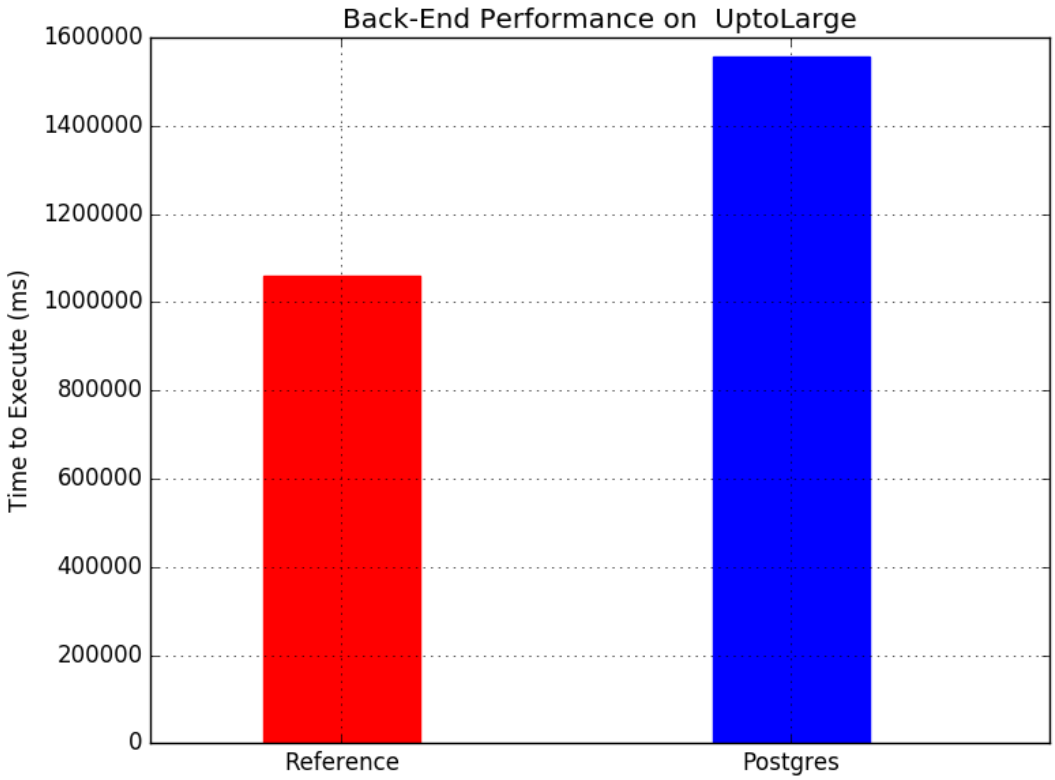

Figure 4.11: Read performance for large `Upto` queries

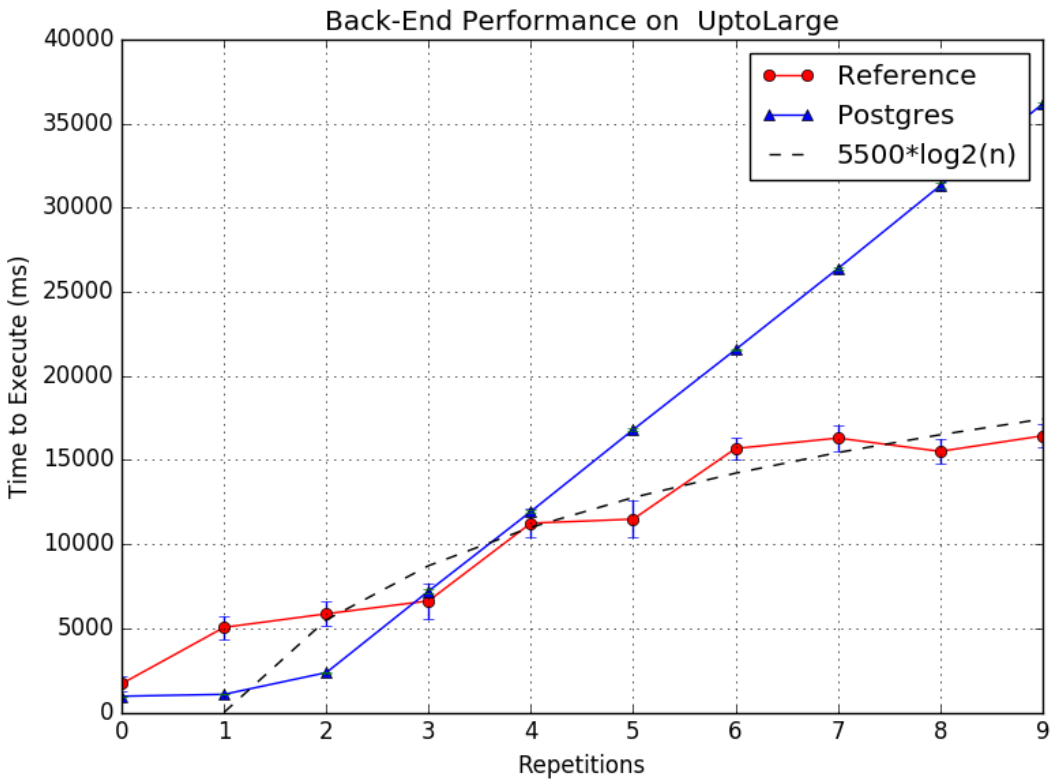

Figure 4.12: Read performance for large `Upto` queries with increasing n. Error bars are 1 standard deviation.

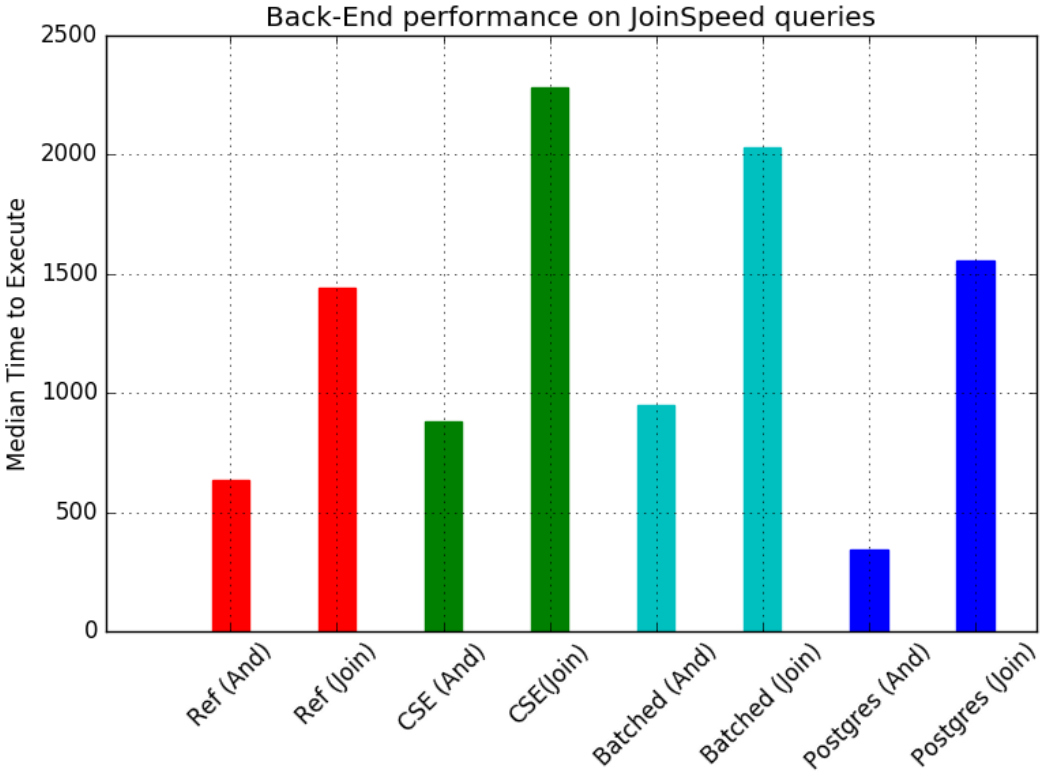

Figure 4.13: Overall read performance for Join-Speed queries

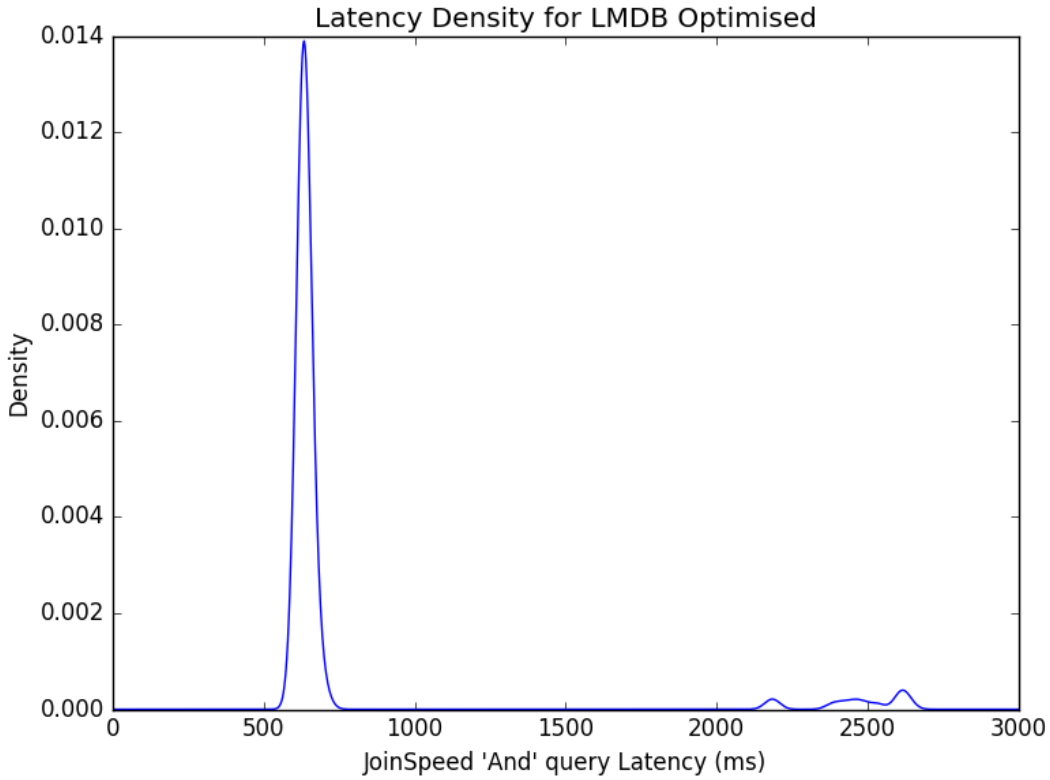

Figure 4.14: The distribution of values for the *And* query of JoinSpeed. Around 7% of values fall into the density on the right of the graph.

Chapter 5

Conclusion

Successes

To the best of my knowledge, the graph database system I have built over the last seven months is one of the, if not the, first explicitly purely functional graph database systems in existence. This project is a demonstration that complex, performant software projects can be built using purely functional languages, whilst keeping the benefits of strong type systems. I feel that despite having written the project in Scala, a high level, often inefficient language I have achieved an acceptable level of performance for graph traversal. Although the system would need significant work in the areas of security, networked access, and tuning to real-world use cases to become commercially viable, it stands as a solid prototype for potential future systems.

Further extensions

Interesting extensions to the core project that, given more time, I would have been interested to explore include:

- An increase in the use of laziness to allow larger queries over larger graphs. Currently, the size of queries are limited by the size of the result sets of sub-queries. However, the Postgres implementation typically runs out of temporary file space before this happens.
- The implementation of deletions and garbage collection of unused views in the database. Garbage collection in particular might be difficult, as the database is not required to store any state about client instances.
- Exploration of improvement of the schema system using a dependent types library, such as `Shapeless`.

- Rewriting the lowest layer of the LMDB back-ends' interpreters in C or another low level language to extract maximum performance from the LMDB system, without having to construct Scala collections at every step.

Lessons learned

With hindsight, I might have aimed to build a narrower project, focusing on a single back-end and comparing performance against a widely used graph database such as Neo4j. Although I very much enjoyed the chance to build a large project in a purely functional style, implementing the large number of moving parts in the project distracted from implementing more interesting optimisations and truly digging down to exploit all redundancy.

Concluding Thoughts

I hope for another chance to explore this space as I feel there are still many areas worth investigating.