Alexander Taylor

A Purely Functional Approach to Graph Queries on a Database

Computer Science Tripos – Part II

St John's College

March 16, 2018

Proforma

Name:	**Alexander Taylor**
College:	**St John's College**
Project Title:	**A Purely Functional Approach to Graph Queries on a**
Examination:	**Computer Science Tripos – Part II, June 2018**
Word Count:	
Project Originator:	Adapted from a project proposal by Dr Eiko Yoneki
Supervisor:	Dr Timothy Jones

Original Aims of the Project

The initial aim of the project was to produce a simple graph database system which wrapped over an SQL database, to expose an API treating the database as a purely functional datastructure. Operations were to be achieved in a monadic fashion, according to well defined DSL semantics. The success criteria were that a user could write a composable query with the DSL and receive correct results with appropriate error checking.

Work Completed

All success criteria were met and exceeded. I also implemented several extensions such as adding write functionality and and a collection of bespoke backends using the LMDB key-value store. These new backends were evaluated by comparing their against the compiled SQL queries generated by the SQL backend.

Special Difficulties

None.

Declaration

I, Alexander Taylor of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

List of Figures

Acknowledgements

Acknowledgements

Chapter 1

Introduction

This project has been to build a new graph database management system. written a purely functional style in the language Scala. I have successfully built a modular, composable read oriented DBMS which has a range of commands and a domain specific read query language.

1.1 Progress

I have achieved all of my initial success criteria, written several extensions (Writes, an extra backend, and optimisations to the backend), and achieved performance comparable to and sometimes exceeding that of postgreSQL across a range of benchmarks.

1.2 Motivations

Graph databases becoming of more interest recently, as the relational model of data is found not to match certain datasets of interest, such as social networks.

Similarly, purely functional, strongly typed programming is also progressing from a research interest to a serious paradigm used in industry. This is especially the case in fields where the ability to rapidly produce high quality code which has a high probability of correctness is vital, such as in automated trading. In such cases, an advanced type system can be used to make guarantees about the behaviour of the program at runtime and enhances the ability to find bugs at compile time rather than in the field, while the purely functional paradigm allows code to be written far more expressively and concisely than in lower level languages.

There exist several attempts to make access to relational databases purely functional or monadic in nature, but there is very little application to graph databases.

https://hackage.haskell.org/package/haskelldb http://www.bcs.org/upload/pdf/ewic˙dp95˙paper14.

1.3 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. Sources of a lack of type safety in typical programs include exceptions (These are inherently not type safe in scala, since exceptions are unchecked), the usage of primitive data types where specific values are expected, such as passing strings to function without wrapping or tagging them in more specific types, and unmarshalling data from a type-erased source. If we dont know whether were expecting a value to be an integer or a string and we try to read it as an integer, there is a chance to throw an exception. By this definition, 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 the database system 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 DSL that is checked at compile time by the scala type checker, and typeclass based marshalling.

Chapter 2

Preparation

2.1 Existing Graph Databases

2.1.1 Classes of Database

Existing graph databases typically fall into two classes: property graphs and edge labelled graphs. https://arxiv.org/pdf/1610.06264.pdf . Edge labelled graphs typically store only a label on any given edge, whereas property graphs can store more attributes on edges, hence being closer to close to relational databases. Edge labelled graphs can be more succinctly modelled mathematically, since they can be represented purely using mathematical relations. Edge labelled graphs also lead to a cleaner syntax (we shall cover this later). Finally, any data expressible using a property graph can be represented using an edge labelled graph by introducing additional nodes to hold attributes.

Insert and acknowledge edge-labelled vs property graph examples in GDrive

2.1.2 Schema

Current graph database systems often do not have rigid schema https://arxiv.org/pdf/1602.00503.pdf, introduction. Instead they use dynamic schema. This requires additional typing checks at runtime. This also fails to fit into our definition of type safety, since we have no guarantees about any objects extracted from the database. Instead, Ive opted for a rigid schema which is built using scala objects and hence is checked by the scala compiler at compile time.

2.1.3 Mutability

Current graph databases tend to be mutable. In order to express concepts that require immutability, such as time-dependent data, the user has impose extra constraints, such as time stamping nodes and relations. A lack of immutability also complicates concurrent semantics and implementation of ACID transactions. The introduction of immutable data structures gives us these effectively for free.

2.1.4 Query Languages

Typically, database systems have their own query language, such as cypher for neo4j and gremlin. These are typically used by generating a query string and submitting it to the database engine. This is inherently un-typesafe. The job of ensuring that absolutely every query that a given program could generate is valid is intractable to undecidable for non trivial programs. By encoding our query language in the host language and aligning the type system to be checked by the host compiler, we can get much stronger guarantees about program correctness. links to cypher and gremlin

2.2 Mutability

In order to implement immutability, we need a way of creating an immutable snapshot of the database at a given point in time. This is done using a system of views. A view is an immutable view of the database. 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. Read/Write diagram

2.3 Query Language

2.3.1 DSL Syntax

I spent some time iterating over a DSL syntax and how I wanted queries to look and be structured. My goals we 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. Ugly syntax figure However, the scala type inference is fairly localised so would need some type parameterisations filled out, yielding a more accurate syntax below. Even uglier syntax This is fairly cluttered syntax and difficult to read. It would also have been complicated by a desire to implement property graph features. Furthermore, we would not be able to easily and safely replicate Neo4js ability to extract an arbitrary number of objects of different types from a path. 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. reword above; neat syntax

This is much more readable and concise. With this syntax, and the assumption that ActsIn relates Actors to Movies, Scalas typesystem can infer that coactor relates Actors to other Actors. There is also syntax for repetitions, intersections and unions. Examples in the appendix

2.3.2 Algebraic Datatypes

These DSL queries should correspond to an intermediate representation. I have picked a set of expression constructors. These fall into two kinds: FindPair and FindSingle, indicating whether they return pairs or individual values. Insert latex of the ADTs here

We also define a view of the database and schema, as well as findables

Import view + schema semantic definitions here; reword if necessary

2.3.3 Typing

In order to ensure the correctness of a given query, there is a type system which checks the correctness of queries. Latex for typing; Whre do I mention that this is encoded into scala?

2.3.4 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. import he latex for these;Proof? I have also proved in the appendix the correspondence of these two sets of semantics.

2.3.5 Commands

In addition to the query construction language, there are five commands which make use of the queries. define and semantics

2.4 Languages and Libraries Used

Scala An object-oriented, functional JVM languages interoperable with Java. I chose Scala for its familiarity, advanced type system, ability to use Java libraries, and ease of deployment across many systems. https://github.com/tperrigo/scala-type-system/wiki/Scala-Type-System-Overview

Scala Build Tool A package manager and build tool for Scala. https://www.scala-sbt.org/

PostgreSQL An open source, multi platform, performant SQL implementation. https://www.postgresql.org/

LMDB An open source, highly optimised key-value datastore. https://symas.com/lmdb/

Scalaz A library for scala providing typeclasses and syntax to aid advanced functional programming. https://github.com/scalaz/scalaz

JDBC Javas standard library support for SQL connections. Used to interact with a postgres database. http://www.oracle.com/technetwork/java/javase/jdbc/index.html

LMDBJava A JNI library allowing access to the LMDB datastore. https://github.com/lmdbjava/lmdbjava

Junit A unit testing library for JVM languages. https://junit.org/junit5/

SLF4J A logging library for JVM languages https://www.slf4j.org/

Spray-JSON A JSON library for Scala, used to generate benchmarking datasets. https://github.com/spray/spray-json

Python Used to generate test datasets. https://www.python.org/

2.5 Software Engineering

The system was designed to be as modular and re-implementable as possible.

Figure from Google drive: the architecture map

2.5.1 Front End

The front end mostly consists of interfaces and syntax for building queries and the schema for the database.

2.5.2 Middle End

The front-ends 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.

2.5.3 Back End

Finally, there are three interfaces that a backend system must implement: DBBackend, DBInstance and DBExecutor. These specify a root backend object that opens an instance which represents a database connection. Each DBInstance has an executor that allows us to execute commands

2.6 Scala Techniques

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

2.6.1 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. Type enrichment example from G Drive This allows us to add syntax to types where a small number of core methods have been defined.

2.6.2 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 a 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. Implicit parameters example 1 This is typically used for purposes such as passing around values that are typically defined once and used many times in a program, such as an ExecutionContext, or a logging framework instance.

Functions can also be implicit and take implicit values Implicit function example, caption: 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.

Implicit parameters example 2

2.6.3 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. example

We can then use the type enrichment feature to add methods to values of a type that is a member of the typeclass.

Monoid typeclass example. Caption: An example of combining the three techniques above to define a Monoid Typeclass, ways to construct instances of the monoid typeclass and a syntax object which enriches types for which moinoid properties can be proven.

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

3.1 Note on Purity and Concurrency

All of the backends aim to preserve the global immutability of the database. The immutable semantics of the database also mean that queries generally dont interfere with each other. This means that we can avoid keeping locks or creating large transactions. Hence, the backends dont require much work to maintain correct concurrency.

3.2 Functional Programming Techniques

3.2.1 Monadic Compilation

At several points in building a backend, it becomes necessary to transform one algebraic type to another. This is typically done 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 subexpressions 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.

State monad definition or link to it

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

3.2.2 Constrained Future Monad

Part of the goal of type safety is the recovery of error cases. Typically, this would be done 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. http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf In scala, this manifests as the built in Try monad and Scalazs \/ (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 Trys failure case is the unsealed Throwable trait, we dont really have a way to ensure we have handled all error cases at compile time, whereas Either has a parametrised error type, which can be a sealed type hierarchy, which is then checked by the scala compiler at runtime. For example, consider the simple interpreter below. All error cases are proved to be handled by the typesystem. Error interpreter example

Typically, were dealing 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 were 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 dont 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.

Hashrecovery typeclass

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 appendix we ensure all non-fatal error cases are contained in a type-safe way.

3.2.3 Operation Monad

As stated, Can I link to it? 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 =¿ ConstrainedFuture[E, (ViewId, A)]) in a similar way to how State monad chains together functions S =¿ (S, A). Each of the commands on a database yields an operation.

3.2.4 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 (though my functional style does not make heavy use of them anyway) 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 for collections such as sets. This leads to a dramatic increase in speed, especially for when large numbers of elements are added sequentially Source?. In these cases, the mutable state never leaks out of the functions that make use of the mutability.

3.3 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.

3.3.1 Schema Hierarchy

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

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 are implemented by effectively defining marshalling functions from the type to and from tuples of Storable (read: primitive) types.

To go in the appendix: The definition of a schema object subclass

3.3.2 DBObjects

To store objects in the database under various backends, we need to have a type erased (at compile time, but not runtime) 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. DBObject and cell definitions

3.3.3 Unerasure

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

3.3.4 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.

Relation definition

3.3.5 SchemaDescription

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

3.3.6 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.

3.4 Query ADT

This feels like it needs rewriting The ADTs described in the preparation section link are implemented in scala each by a pair of ADTs: a typed and type-erased equivalent for each. The typed ADTs are parameterised by the object types which they lookup. Due to this typing and scalas type inference, I have encoded the type of the typed ADTs such that the scala compiler checks the types and does inference for us, according to the type rules of the query language. Appendix: Type definitions of ADT 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) rule, and the (Find) rule, as we cannot predict the contents of the SchemaDescription at compile time without dependent types. These 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 runtime 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.

3.5 Commands

As specified in the previous chapter, each backend needs to implement five commands:

- find(S)
- findPairs(P)
- ShortestPath(start, end, P)
- allShortestPaths(start, P)
- insert(relations).

Each command should return an Operation of the correct type.

3.6 DSL

The DSL mostly consists of syntactic sugar to make queries easier to read. It is implemented using the type enrichment pattern. Both Relatation and FindPair implement the trait (interface) FindSingleAble, so we can use type enrichment to write new DSL opertors. The main thing to note in the DSL is the use of arrows to chain relations together. (See RelationSyntax.scala in the appendix for examples of DSL). Put in appendix

3.7 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 backends can make use of them regardless of the underlying types. These algorithms are found in core.utils.algorithms

3.7.1 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. (Diagram showing expanding fringe)

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. diagram

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. This reachable set is diffed (define diff) with the accumulator to find the newly reached nodes. These are now added to the fringe queue and the accumulator. When the fringe is empty, we return the accumulator.

3.7.2 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). 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. Diagram

Upto The next algorithm is to compute Upto. This is again done 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 the memo, while the accumulators are unioned with the fringe of the appropriate root with each iteration. Diagram

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

3.7.3 Pathfinding

3.7.4 Joins

3.8 Views and Commits

3.9 Memory Backend

3.9.1 Table Structure

3.9.2 Reads

3.9.3 Left Optimisation

3.9.4 Writes

3.9.5 Storage

3.9.6 Mutability

3.9.7 Pathfinding and fixed point traversal

3.10 PostgreSQL backend

3.10.1 Table Structure

Control Tables

Schema defined Tables

3.10.2 Query Structure

3.10.3 Monadic Compilation

3.10.4 Writes

3.10.5 Mutability

3.10.6 Pathfinding and Fixed Point Traversal

3.10.7 Object Storage

3.11 LMDB Backends

3.11.1 Common

LMDB API

Terminology

Transactions

JVM API

Storage and Keys

Table Structure

Writes

3.11.2 Original LMDB Implementation

3.11.3 Batched

Read Batching

Pre-Caching

FindFrom

3.11.4 Common Subexpression Elimination

The Memoisation Problem

Retrievers

Monadic Compilation

3.11.5 Complex Common Subexpression Elimination

Index Building

Exactly

Upto Optimisation

Chapter 4

Evaluation

4.1 Unit Tests

The correctness of each backend is tested using a suite of 45 unit tests which verify adherence to 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, Ive written a test case to target that bug, ensuring that it is not leaked into production again.

4.2 Performance Tests

4.2.1 Introduction

In order to evaluate the effectiveness of the various optimisations to the LMDB backends described above, it was necessary to run performance tests over wide range of queries. The various LMDB backends were tested against each other as a control and particularly against the SQL backend as a standard to beat. The in-memory backend was omitted as initial tests indicated that it runs with approximately the same algorithmic characteristics as the Original LMDB implementation. The LMDB implementation was able to do indexing faster than Scalas tree based hash maps, which are used by the in-memory implementation. Further more, the original LMDB implementation is typically around 3x slower than the batched version for most jobs, so we also omit it for most tests, since most of the algorithms it uses are the same.

4.2.2 Hardware

Tests were run on the oslo machine belonging to Timothy Jones group. Its specifications are shown below. All of the backends ran off of an SSD. Oslo Specs

4.2.3 Datasets

IMDB

UFC

4.2.4 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 test instances must implement. Each Test must have a setup method and a test method which is run on a DBInstance with the test method indexed by a TestIndex. The test specification must give a maximum index and a mask to avoid running inappropriate backends (For example, those that might take too long on a large test.) The benchmarks that I have run test only the read speed. The time taken to construct the database is not included.

4.2.5 Results

Overall Picture

Pathfinding

Redundancy

Conjunctions and Disjunctions

Tests that involve repetitions

Exactly Test

Exactly Pairs

UptoTest

UptoLarge

JoinSpeed

4.3 Semantics Proofs

4.3.1 Denotational == Operational

4.3.2 Typesafety

4.3.3 Join as a Monoid

Chapter 5

Conclusion

I hope that this rough guide to writing a dissertation is LaTeX has been helpful and saved you time.