Computer Science Tripos – Part II – Project Proposal

A purely functional approach to graph queries on a database

Alexander Taylor

St John's College

Originator: Adapted from Eiko Yoneki's project `http://www.cl.cam.ac.uk/research/srg/netos/stud-projs/current/#QueryLayer`

Supervisor: Dr. Timothy Jones

Director of Studies: Dr. Robert Mullins

Project overseers: Dr. Thomas Sauerwald, Prof Jon Crowcroft

Introduction

Databases are ubiquitous in modern industrial programming. However, there are several properties of leading database technologies which hinder programming using them. Firstly, SQL databases are poorly suited to transitive operations over relations. Queries end up being unnecessarily bloated and slow. Secondly, database systems often come with a query language, such as SQL, or Cipher. Usage of the database then often involves constructing a query for the database language as a string and then submitting it to the database system, then interpreting the results, which are typed in the style of the underlying database, as objects in the native language. This leads to issues with security and type safety, along with limiting the capability of compile-time checks. Finally, the model of most databases appears to be that of unsafe, imperative code. This obscures semantics and hides data dependencies.

Project Description

This project is a proposed solution to these difficulties. Written in Scala, it shall provide a typesafe abstraction over an underlying SQL database to allow for easy, composable graph queries, allowing the application programmer to treat the database as a purely functional data structures. Furthermore, it shall return its results in a monadic, asynchronous container, which will allow the user to chain together the results of queries in an efficient, meaningful way.

Starting point

I shall be writing the module in Scala, using the ScalaZ (`https://github.com/scalaz/scalaz`), and ScalikeJDBC (`http://scalikejdbc.org/`) dependencies. These provide a library of useful purely functional definitions, such as type classes and monad transformers, and database access to scala, respectively.

Resources required

Software

I intend to track progress and back up work using a Github repository and google drive.

Hardware

- I will be using a pair of computers for development: desktop + laptop. They both have 16GB of RAM and a quad core intel processor and run windows 8.

- I will keep my work synchronised between both using a GitHub repository.
- I expect that this triple setup will provide enough redundancy in the event of hardware failure.
- For my evaluation, I expect to use my desktop which has an SSD

Project Structure

The project can be broken down into four linked parts:

- A system of views that allow us to treat the database as a purely functional data structure. A view is the state of the database at a given logical point in time. Any operation repeated on the same view gives the same result no matter the number of repetitions. Write operations that mutate a view instead create a new view with the changes applied.
- An execution system that runs internally represented queries against a particular view. This will utilise a simple internal bytecode like ADT to represent queries before they are executed.
- A set of return types, primarily a monadic, generic result container that combines monadic asynchrony (Queries take time to execute), error handling (Queries should be able to fail in type-safe ways), and nondeterminism (Queries may return a set of results). We also need a datatype, which is held by the result container, to represent paths through the graph. This should also permit various manipulations, such as folds.
- An integrated, type-safe DSL for constructing and composing queries. This should be algebraic in style, and allow users to build queries using valid scala code.

Success Criteria

The project will be a success if a user can write a composable query with the DSL and receive correct results with appropriate error checking

Extensions

- The immediate extension is to add write queries to the database, generating new immutable views in the database. This will also require garbage collection of unreachable views and commits
- Next is to add some optimisations to queries before they are executed.

Evaluation

There are several potential means of evaluation:

- Performance against pure SQL
  - A bare minimum is within a factor of 2 against well-written SQL over short queries
  - It's not feasible to directly outperform perfectly written SQL because the system effectively compiles to SQL to make queries.
  - But on the other hand, there should not be a significant performance difference, other than that caused by filtering the database entries for those that match a particular logical view.
- A proof of type safety/that errors can be bound within a sealed trait, allowing us to pattern match all possible error cases
- A set of semantics for queries
- If I manage to complete my primary extension, write queries, then I also intend to evaluate the performance of the garbage collector, and the database's disk footprint
- Evaluation of any optimisations I add to the project against the unoptimised project

Timetable

Michaelmas week 2

- Configure Github repository
- I will write up some motivating examples of what syntax I want in the DSL
- This will also require some research of how I want to use advanced features of the scala type system, such as implicits to implement syntax features

Michaelmas week 3

- I will draw up the main interfaces between the main classes in the project
  - Results container
  - In memory view objects
  - DSL query methods

Michaelmas week 3-4

- I will start looking at the low-level implementation of the project.
- This involves implementing a conversion of user-level case classes to database schema and building a views-and-commits table.

Michaelmas week 5-6

- I will start building the intermediate language ADT

Michaelmas 7-8

- Construction of DSL according to the desired syntax
- I will then implement the methods for converting DSL expressions into the intermediate language

Michaelmas Vacation

- 2 Weeks of buffering, in case I have got behind
- 2 Weeks spent implementing execution of the simpler read commands of the intermediate language against a view as SQL. This will require some research into building especially efficient SQL queries
- 2 weeks I will consolidate everything from my project log book along with any code written into a document, this will be a starting point for my dissertation draft. I will also work on my Progress Report for Lent Term.

Lent Week 1-2

- I will implement the remaining read commands of the intermediate language (such as pathfinding)

Lent Week 3

- Initial performance tests
- At this point, the initial success criteria will have been met

Lent Week 4-5

- Extension of the project to include monadic writes to the database

Lent Week 6

- Since the system will now be able to generate new views, I will need to implement a simple garbage collector to remove views and commits when the memory representations of them become unreachable

Lent Week 7-8

- Buffer weeks and further performance evaluation

Easter Vacation

- I shall consolidate from my project log book, along with any code written into a document, combining it with my draft from Michaelmas, and start drafting my dissertation.

Easter Week 1-3

- I shall iterate over drafting my dissertation and redrafting according to feedback from my supervisor and DoS