Query Universal Interface

Haak Saxberg and Jess Hester

November 5, 2011

1 Domain

Different database backends have different interfaces. This means that it takes a lot of work to transition from one database to another database; in the case of SQL-compliant interfaces, like MySQL and PostgreSQL, this work is manageable, since these backends have overlapping syntaxes and very simple representation models (rows with columns, stored in tables). However, if one needed to switch between non-SQL backends, the task is nontrivial. Two of the biggest non-relational databases (Google's AppEngine and Amazon's SimpleDB) have very different APIs; migrating an application between the two is full of headaches. A DSL that abstracts the interaction between an application and the database API would alleviate that headache, making migration of code a much smoother process.

2 Language Overview

2.1 Basic Computation

Converting instances of classes into database-intelligible formats and vice-versa. In a relational database scheme, this would mean making a table for each class, and columns for each attribute of the class (and being able to reverse that translation as necessary). In non-relational schemes, this could be very similar or very different; Amazon's S3 service uses the concept of "buckets" for data, while Google's AppEngine stores its data in a format much more like traditional tables.

2.2 Basic Data Structures

Classes themselves are the data structures we're interested in, since those are the things we want to store in the databases. In order to keep classes and their database representations separate, we'll probably have "Mapper" structures who do the lifting. If we have time, then we might also have Syntax Trees, which would allow us to parse (limited) "raw" queries written in SQL (or some other standard) into actionable code that can talk to the proper backend.

2.3 Basic Control Structures

Users of our language will be able to filter results by chaining filtering methods on the result of some query. If we have time, as we said before, they'll be able to write "raw" queries, which will grant them a greater amount of control over their interaction with the database. (This is a pipe dream)

2.4 Input and Output

Since this is an internal DSL, it acts on already extant classes and their instances; by introspecting a class, we can find its attributes. These can then be stored in the appropriate place, and retrieved as needed - storage and retrieval methods are thus the inputs of our DSL. Output of storage routines is either nothing if successful or an exception if something went wrong. In the retrieval routines, output is either an instance of the class if successful or an exception if unsuccessful.

2.5 Error Handling

As mentioned above, there could be a number of problems encountered; issues with retrieval could include no database found, no data in the database to retrieve, no data that matches the definition of the class

we're trying to retrieve to, or interrupted connections to the database. With storage, we also have to deal with non-existent or interrupted database connections, as well as appropriate locations not existing for the data of the instance being stored. The traditional way to handle these is to throw informative exceptions - then the user can catch them and perform recovery as they see fit.

2.6 Other DSLs in same domain?

Yes, but none that focus on universalizing interaction with *non-relational* databases; there are many that attempt to bridge the gaps between relational databases (ala SQLAlchemy or Django's own ORM), but there doesn't seem to be a large market for non-relational bridgers. At least, not a publicly available market.

3 Implementation Plan

We're leaning strongly towards Python as the host language; there are already low-level APIs for both of our focus databases available in Python, and we're more comfortable in that language than in Scala or Java. Python's decorator concepts also lend themselves to a "store this, not that" kind of model, and hopefully mean that we won't have to force users to redeclare their classes - they'd just decorate them with our decorators.
There are APIs available for Java as well, but since we haven't seen much of the way that Scala can 'pull out' to Java, we're hesitant to move in that direction.

4 Teamwork Plan

FUN VERSION:
I mean, we're both going to work on it. Neither of us trusts the other one enough to not do that.
Since this is a new area for both of us, we're probably going to use a kind of pair programming – always working in the same room, at the least – so that we can have both pairs of eyes watching for insidious logical errors. All planning and design is going to be done jointly, using the democratic system of voting. In the event of a tie, a gladiatorial battle will take place. The winner survives and gets to choose the design path. The more serious the design choice, the greater the spectacle (core language decisions will be decided by a naval battle hosted in Case's courtyard).

SERIOUS VERSION:
Language planning and design will always be done jointly. Pair programming will be employed to ensure equal division of labor. While one person is implementing, the other is documenting the process for use in the final documentation. This will ensure that we are both acquainted with all aspects of the language we're implementing, as well as allowing the documenter to check the progress of the implementer against the design plan as we go.