Immortals

Challenge Problem 1 – DataBase Schema Migration

# Overview

Many applications make use of relational databases for storing and retrieving data. Our SA application uses such a database for storing location history, as well as for storage of points of interest and map data. The server also stores locations, but for all the clients. Relational databases have a *schema*, which describes the layout and content of the data within the database. As applications evolve, and the usage changes, sometimes the original layout of the database can become inefficient for the kinds of queries that are being run: it could be that much more data than was originally intended is being stored, it could be that the database was designed for a slightly different use than what the application is currently doing.

What often happens in these situations is that a human database expert will analyze the performance of the queries against the current data sets and will propose a new database layout, or *normalization*, that improves the performance of the queries the applications are actually using. Once this layout change is put into place, all the SQL queries that an application makes must be migrated to the new database schema.

The scenarios for this challenge problem address this last issue: migrating the SQL queries to the new database schema. The key technology for this is OSU’s CASTOR relational learning algorithm. We will also utilize our discovery tools to find where the application is making SQL queries. Success will allow the application to function when making queries against a database with the new schema.

# Test Data

The CASTOR algorithm requires some baseline data from the old database, as well as expected results from a set of SQL queries. It then requires the same actual data to be loaded into the new schema. CASTOR can use the expected results to find new SQL queries against the new schema such that the expected results for each query are returned.

**Example:** Table 1 shows an original schema and its evolved schema, for a database about students and faculties in a university. Consider the following query that retrieves the years of students who are in the phase ‘*qualification*‘ over the original schema in Table 1.

Year (y) ← inPhase(x, ‘qualification’), yearsInProgram(x, y).

(**SQL**: Select year from inPhase, yearsInProgram where inPhase.stud = yearsInProgram.stud and inPhase.phase = ‘qualification’ )

The evolved schema normalizes the table in a particular way. CASTOR will produce an SQL statement that is equivalent to the old query:

Year(y) ← student(x, ‘qualification’ ,y).

|  |  |
| --- | --- |
| **Original Schema** | **Alternative Schema** |
| student(stud) | student(stud,phase,years) |
| inPhase(stud,phase) | professor(prof,position) |
| yearsInProgram(stud,years) | publication(title,person) |
| professor(prof) | taughtBy(crs,prof,term) |
| hasPosition(prof,position) |  |
| publication(title,person) |  |
| taughtBy(crs,prof,term) |  |

Table : Original schema and evolved schema. The original schema has the data broken into 7 tables. The evolved schema has 4 tables.

(**SQL**: Select year from student where student.phase = ‘qualification’ )

Our application will have a number of queries, and we will provide at least one baseline dataset, along with the ‘expected value’ of those queries (i.e., what the query should return against a given dataset). We will provide a ‘new schema’ generator described below to support testing with various ‘evolved schemas’.

# Test Parameters

Testing of this capability requires that some known data be loaded into the new schema. As such, we believe that it will be necessary to provide a tool that will allow generation of a wide variety of normalized databases. Our intent is to start with a database that has X columns. The Test Harness will choose a number of tables to normalize into, and choose which columns will be in which table. For a baseline schema where the columns were:

Id, Time, Latitude, Longitude, Altitude, task, speed, course

A potential input from the TH might look like:

Number of tables: 3

Table 1: {time} / Table 2: {lat, lon, alt} / Table 3: {task, speed, course}

Another permutation might be:

Number of tables: 3

Table 1: {alt, task} / Table 2: { lon, speed} / Table 3: {lat, course}

(note that as a practical matter, the primary key field, in this case ‘id’ will be added to all tables in the ‘evolved’ schema. This isn’t necessary for CASTOR, but ensures that the normalization is valid)

# Test Procedure

The Test Harness will provide the number of tables to normalize to, as well as which columns to put in each table. The Test Adapter will then generate the appropriate evolved database and populate it with the sample data. At this point we have all the input needed for CASTOR, so the DAS will be invoked and CASTOR will determine a replacement SQL query for each query used in the application.

The testing strategy used in Phase 1 will work normally for this challenge problem. Baseline A would be the original queries on the original schema, and should never fail. Baseline B will be expected to fail on most queries, but it is possible that certain permutations of the database schema might not affect very simple queries. However, given that there are a number of queries the application uses, as long as there is a query that depends on every column, then any re-arranging of the schema will cause at least one of the baseline queries to fail. Challenge Stage is fairly straightforward: we can run the new queries against the new database and test to see that it returned expected results.

# Interface to the Test Harness (API)

In the example below, there are two tables defined, on having columns 1 and 2, and the other table contains column3.

|  |
| --- |
|  |
| // API to normalize the database  POST http://brass-ta/action/perturbDatabase  TEST\_ACTION:    {"TIME" : TIME\_ENCODING,     "ARGUMENTS" : {"tables": [  { "columns": [columnName1, columnName2]},  { "columns": [columnName3,…]}  …  ]  }  }  ACTION\_RESULT:    {"TIME" : TIME\_ENCODING,     "RESULT" : {"nQueriesExecuted" : Integer, "successfulQueries" : Integer}    } |
|  |

*An interaction diagram like the following would also be helpful in our discussions.*



# Intent Specification and Evaluation Metrics

The intent in this challenge problem is fairly straightforward: does the query return the expected results. While in the real world the purpose of doing a normalization would be to speed up queries, it might make sense to measure how long the query takes. However, since the testing infrastructure we are proposing allows arbitrary schema rearrangements that are not tuned for particular queries, it is not necessarily the case that the queries themselves are faster. Thus the only measure of ‘intent’ that really makes sense is one of validating that particular queries return expected results.

Note that CASTOR requires access to what are essentially the intent tests (e.g. known answers to a given query). So there is a bit of a question as to whether we should provide a different set of tests for the TA to run than what CASTOR runs itself. To make that work however, we’d need a different data set in the new schema. This implies that the “schema generator” tool would potentially emit multiple databases, each with the new schema but with unique datasets.