QL
(/kōol/)

Matthew Piccolella: mjp2220 (Manager)
Anshul Gupta: akg2155 (System Architect)
Evan Tarrh: ert2123 (Tester)
Gary Lin: gml2153 (Language Guru)
Mayank Mahajan: mm4399 (System Integrator)

COMS4115 Project Proposal

September 30th, 2015

1 Introduction

Data is exploding. A CSC study predicts that there will be 44 times as much data in 2020 as there was in 2009. With this comes an increasing demand for the ability to understand and manipulate data. In attempting to manipulate and view data, the way the data is stored is often a barrier.

Two very common ways to store data involve relational and non-relational databases. In relational databases, data is stored inside of tables, where each table has rows and columns, where rows represent each instance of the object and columns represent a particular attribute that is being stored. Relational databases include things like MySQL and Postgres, and are accessed using SQL, a database query language.

Non-relational databases do not use the tabular relations used in relational databases, instead opting for key-value or document-based storage. In this way, objects can often be more flexible, but can be at times harder to query as a result of the lack of regularization. MongoDB is one of the most common NoSQL databases, and uses the JSON file format to store its data.

However, working with both of these sets of data at the same time can pose a challenge, and this situation comes up quite frequently in data analysis. Currently, if you wish to view and manipulate data that is split across SQL and NoSQL database formats (for example, SQL and JSON), you will have to write your queries separately. This is a large barrier for quick and simple data processing, and we would like to solve it.

We propose QL (/kōol/). QL will allow you to load data of several different types (SQL and JSON) and perform quick, simple queries that are storage-agnostic. Our language will allow you to worry less about the different ways your data is stored and more about what you hope to gather from the data. Using collections that aggregate different data types, operators that allow for quick and simple querying and manipulation, and output that prints separately for each data store, QL makes data analysis across different data sets simple.

2 Language Features

Primitive Data Types:

- *int* - number type that holds signed integer values.
- *double* - number type that holds signed double values.
- *string* - sequence of characters that have a representation in the ASCII table. Represented with double quotes and escaped with a backslash.
- *bool* - data type that holds the value of true or false.

Complex Data Types:

- $JsonTable$ - represents a connection to a JSON database. Each JsonTable will reference a single JSON database and queries for each database will be performed on JsonTable objects. JsonTables do not have the database saved in memory, instead think of it as a pointer to a database. Each query requires the JsonTable to access the database and perform the query.
- $SqlTable$ - represents a connection to a SQL database. Similar to the JsonTable, a SqlTable will reference a single SQL database and queries for a respective database will be performed on that particular SqlTable object. The SqlTable is also a pointer to a database and does not have the database contents saved locally.
- $Collection$ - hold JsonTable and SqlTable objects. Collections are enumerable, allowing you to iterate through the different database objects to perform uniform tasks to each one. This allows for personalized table organization; for instance, a zoo can have a collection of mammal tables, amphibian tables, and reptile tables.
- $Json$ - data type to store JavaScript Object Notation, or JSON objects. This is something that the user will never touch and will be used under the hood. JSON will be used to hold data that can be accessed from JsonTables.
- $Tuple$ - data type that consists of primitives that are comma separated in parentheses. Tuples will also be used under the hood and users will not interact with them. They will be used to hold data that can be accessed from SqlTables.

- *List* - holds Json data, Tuple data, or other lists, but the content of the List must be homogenous (must hold either just Json or just Tuple values or just other lists). Each type of List will be used to store query results from JsonTable and SqlTable, respectively. Ordered and enumerable, allowing you to iterate over it.

Operators:
Note: For collections containing JsonTable and SqlTable, the following operators are separately applied on each JsonTable and SqlTable independently.

- + (concatenation)
  - Behaves regularly for primitive data types.
  - Also adds files to a collection.
- - (difference)
  - Behaves regularly for primitive data types.
  - Also removes files from a collection based on object reference.
- [] (key lookup)
  - Used to find the value of a key in a table.
  - Can also be nested for JSON lookups.
  - e.g. $A[\text{"}name\text{"}]$ returns the value (if it exists) of the attribute called name in the datatype.
- >>
  - Writes the collection defined in the left operand to disk.
  - File name: right operand.
- ?
  - Evaluates a boolean logic statement against a collection.
  - Allows us to filter out all elements of the JsonTable and the SqlTable for which the boolean expression defined as the right operand of this operator evaluate to true.
- $>, <, >=, <=, !=, ==$ - normal logical comparisons for primitive data types.
- // - single line comments operate until the newline character.

3 Sample Programs

animals.json

```
[
    {
        "name": "dog",
        "age": 10,
        "height": {
            "feet": 5,
            "inches": 3
        }
    },
    {
        "name": "cow",
        "age": 3
    }
]
```

animals.sql

index	name	age	gender
0	cat	20	M
1	human	5	F

test.ql

```
Collection c
c += JsonTable("animals.json")
c += SqlTable("animals.sql")
results = c ? (["age"] >= 10)
results >> "output.txt"
c -= "animals.sql"
results2 = c ? (["age"] >= 10)
results2 ? (["age"] > 15) >> "output2.txt"
c ? (["age"] > 15) >> "output3.txt"

results4 = c ? (["height"]["feet"] == 5)
```

output.txt

```
[
    {
        "name" : "dog",
        "age" : 10,
        "height" : {
            "feet" : 5,
            "inches" : 3
        }
    }
]
```

```
index    name    age    gender
0        cat     20     M
```

*output*2.*txt*

```
index    name    age    gender
0        cat     20     M
```

*output*3.*txt*
Empty because *animals.sql* is removed from the collection

*test*2.*ql*

```
Collection c
JsonTable animals ("animals.json")
c += animals
results = c ? (true) // Get all of the data.
results["age"]++ // increments everything's age
results ? (["name"] == "dog")["height"]["inches"]++ // increments dog's height
results >> "output.txt"
```

output.txt

```
[
    {
        "name" : "dog",
        "age" : 11,
        "height" : {
            "feet" : 5,
            "inches" : 4
        }
    },
    {
        "name" : "cow",
        "age" : 4
    }
]
```