Mike Calder, Nathan Longnecker
CS542 - Database Management Systems

Project 3 - Query Execution Engine

We implemented a simple query execution engine. The primary interface is the query planner class, which exposes functionality allowing users to select, project, and join relations. This implementation represents each relation in comma separated value (csv) format. This allows users to store or view their tables data using Excel and run queries against it without needing to convert to a different format. The assignment does not call for the ability to update the data, so we did not find it necessary to take extra steps to follow the ACID paradigm.

Using our query execution engine

We wrote our query execution engine in Java. The program exposes the following interface:

```
void select(Reader in, Writer out, int compareOn, Conditional1<String> conditional);
void select(Reader in, Writer out, int compareOne, int compareTwo, Conditional2<String> conditional);
void project(Reader in, Writer out, List<Integer> keep);
void join(Reader in1, Reader in2, Writer out, int index1, int index2, Conditional2<String> condtional);
void executeQuery();
```

This interface is visible in our code in the class QueryPlanner. For details on what each function and parameter is for, see the documentation in the query planner class.

To use the program, the user should first convert their data to csv format. Then they should create a main method that creates readers to read in the data from their csv data files, and pass those readers to the in parameter of the first operation. Operations can be connected using connected PipedReaders and PipedWriters (as shown in our sample main). To output the final result, either create a writer that writes to standard out (shown in our sample main) or that writes to a file. Once all the readers and writers are set up, call the executeQuery method on the query planner to run it. The query planner will spawn separate threads for each operation and pipeline each tuple to the next operation as soon as its ready.

Assumptions made

We assumed that the user had their data in CSV format. The query engine could be extended to support other data formats relatively easily by adding an interpreter to the initial reader (and/or final writer) to convert it to or from CSV, which is the format used when passing data between each step through the PipedReaders and PipedWriters.

We assumed that the relations were not indexed for simplicity. A more complex query execution engine could be written that makes use of indices if they exist.

We assumed that this query execution engine did not need to be able to update, insert, or delete data, although it could easily be extended to do so.

We assumed that one of the two tables would fit into memory. The join operation we implemented only supports a single pass join, but a two-pass join operation could be added by adding an operation class for two-pass joins.

How it works

Reader and Writer Generalization
Each operation supported by our query execution engine is sufficiently generalized so it can be reused in multiple queries or multiple times in the same query. Our sample main gives one example of how the query execution engine could be used, but other main methods could be written that could use a different query plan (such as with multiple joins and selections). Each operation uses a Reader and a Writer. This allows the user to specify where the data for each operation comes from and where the result goes. The user can use BufferedReader and FileReaders to pipe data directly from a file into the first operation, and can create PipedReaders and PipedWriters to pipe data between operations, and use a OutputStreamWriter to write the final data directly to standard out (Or a FileWriter to write it to a file). This allows each operation to act completely independently and not care about where the data it is processing comes from or where it is going, so the operations can be reorganized in any way to construct any sort of query that uses project, select and join.

Condition Generalization
The user can also take advantage of lambda expressions to create queries with different types of joins like full outer joins using the same join operation class the equijoin used in the sample main method. The only thing that would need to be modified is the lambda expression passed to the join operation and the list of attributes used by the lambda expression. Similarly, the select operation is generalized to allow any java lambda expression when deciding which tuples to include in the output.

Pipelining
The application uses multiple threads to allow it to take advantage of pipelining. Each operation has its own thread. Each thread blocks until there is data available from its reader so the thread can process each tuple as soon as it is ready.

Testing
We have created a few JUnit 4 tests to test constructing and running each of the operation classes. Running the final query on the world database data set served as our stress test for this application. These tests are visible in the test folder of our github repository.