Report for Project 3: Implementing a Query Engine

I. Introduction and Assumptions:

1. Assume the types of left value and right value in join condition are the same, and the data in Value (when Condition.bRhsIsAttr is false) will not be NULL (we do not support NULL for TypeVarChar as condition for now). These validations are supposed to be done before invoking query engine.

2. Assume all joins with an attribute from right table in the given condition (i.e., Condition. bRhsIsAttr is true) are INNER JOIN. That is to say, under this condition for each tuple from left table, if no tuple in right table matches the given condition, no output will be produced.

3. Assume HashJoin is only for equi-join, and in HashJoin at most one tuple in right (smaller) table matches the given condition. In this sense, HashJoin joins only once for each tuple from left (larger) table with tuples in current bucket from right (smaller) table, which have been hashed to in-memory hash_table.

4. Memory leak, if exists, can be quite apparent during nested-loop joins, in particular Tuple NL-Join. There were some points in our code causing memory leak, but they have been now fixed and according to current tests memory usage is under control.

II. Design

1. Attributes from input Iterator(s) are cached as private fields for later use as long as one class (Filter, Project, or Joins) is initialized (i.e., in its constructor).

2. For NLJoin and INLJoin, one tuple from left table and its corresponding attribute value for join condition are cached as private fields, so Joins do not need to read them again until there is no tuple available in right table (i.e., the scan of right table reaches the end). The type of join attribute is also cached during this process.

3. Hash functions for HashJoin:

   a. H(key): `unsigned getPartitionHash(char* value, AttrType type, unsigned M)`
   Partitions all tuples into buckets during partition phase.

   b. h(key): `unsigned getJoinHash(char* value, AttrType type, unsigned M)`
   Partitions tuples from one bucket into in-memory hash_table during join phase. It is similar to the partition hash function H(key), but first produces result in size of 2*M, and then (value mod M) is returned as hash result.

- Hash for TypeInt: uses key value as seed to invoke srand(seed), and operates on rand() to generate hash number.

- Hash for TypeReal: leverages multiplication method[1] to generate value for composing hash number.
- Hash for TypeVarChar: leverages BKDR (Brian Kernighan and Dennis Ritchie) hash function[2] to generate value for composing hash number.

III.Implementation

1. Filter

1. `RC Filter::getNextTuple(void *data)`
Get next tuple from input iterator given the input condition as a filter.

2. `void Filter::getAttributes(vector<Attribute> &attrs) const`
Return the attribute of the output tuple of the filter iterator.

2. Project

In order to ensure the sequence of fields in output tuple as same as the sequence of the given attribute names (`const vector<string> &attrNames`) for projection, we define a `struct` named Buff (see definition below) breaking down each tuple to fields and storing them in Buffs. Then these Buffs are shuffled according to the sequence of given attribute names.

```
struct Buff {
 AttrLength length;        // length of the data
 void       *data;         // container of one field
};
```

1. `RC Project::getNextTuple(void *data)`
Gets one tuple from input Iterator and shuffles it according to given attribute names for projection.

2. `void Project::getAttributes(vector<Attribute> &attrs) const`
Retrieves Attributes from input Iterator and shuffles it according to given attribute names for projection.

3. NLJoin

Implements Tuple Nested-Loop Join. See below for more details.

1. `RC NLJoin::getNextTuple(void *data)`
Gets one tuple from left iterator, and then gets one tuple from right iterator; performs condition validation, and produces one output tuple if the condition is satisfied. If no tuple is

[1] http://lcm.csa.iisc.ernet.in/dsa/node44.html
[2] *The C Programming Language*, Brian Kernighan and Dennis Ritchie

available in right iterator, reset scan iterator and perform above steps for the next tuple in left iterator. Repeat it until there is no more tuple available in left iterator.

2. `void NLJoin::getAttributes(vector<Attribute> &attrs) const`
Directly concatenates attributes from left and right input Iterators.

4. INLJoin

Implements Tuple Nested-Loop Join given the inner iterator is an index iterator.

1. `RC INLJoin::getNextTuple(void *data)`
Gets one tuple from left iterator, and then gets another tuple from right index iterator; performs condition validation, and produces one output tuple by concatenating the previous two tuples if the condition is satisfied. If there is no tuple available in right iterator, resets the right index iterator and repeat the above steps from the next tuple in left iterator. Return QE_EOF if there is no tuple available in left iterator.

2. `void INLJoin::getAttributes(vector<Attribute> &attrs) const`
Directly concatenates attributes from left and right input Iterators.

5. HashJoin

Implements GRACE Hash Join, including partition phase and join phase. Remember, assume left table is lager and right table is the smaller one. In addition, assume `numPages` is at least greater than 1, and the usage of buffer is:

1. Partition phase: (`numPages` - 1) pages to partition, and 1 page to read data.
2. Join phase: (`numPages` - 1) pages to hash the current bucket of right (smaller) table, and 1 page to read tuple from the corresponding bucket of left (larger) table.

Assume the caller of the HashJoin function provides the space for output (namely, `void* data`) during the join phase.

For partition phase, corresponding partition (or bucket) file is created for each bucket of either left or right table, named as HJ_LEFT_PAR_X and HJ_RIGHT_PAR_X, respectively (X represents bucket number). In the constructor, partitions tuples from both left and right input Iterators, and stores them in partition files. Hash function H(key) is utilized to partition all tuples, whereas h(key) is utilized during join phase to distribute tuples from one bucket.

A `struct` named Bucket (see below for definition) is defined to store data for one bucket. A vector of Buckets with H(key) as index stores partitioned tuples. As long as the length of data in each Bucket approaches PF_PAGE_SIZE (i.e., appending another tuple will make the length of data exceed PF_PAGE_SIZE), data are written to partition files with the first 4 bytes storing the length of data on current page. Length and data fields of corresponding bucket in memory are reset to receive following data. At the end of partition phase, flushes all buckets whose data fields are not NULL to corresponding partition files. Currently, all partition files are created in advance to simplify the process of partitioning.

See below for more details about join phase.

```
struct Bucket {
 unsigned length;   // length of current data
 void     *data;    // container of tuples distributed to this bucket
};
```

```
1. RC HashJoin::getNextTuple(void* data)
```

Reads one bucket of right (smaller) table, and hashes tuples in the bucket to a hash_table. Before reading data from a partition file, its total page number will be examined. Only those files with data will be read; otherwise, the file will be deleted. Bucket `struct` is re-used during the join phase. A vector of Buckets with h(key) as index represents the hash_table. Removes partition files after the join for those buckets has been processed.

After one bucket of right table is hashed in memory, reads one tuple from corresponding bucket (i.e., with the same hash number from H(key)) of left table. Probes the hash_table in-memory to see if they satisfy the given condition. If yes, produces output tuple. Otherwise, reads the following tuple from the bucket of left table and probes the hash_table again until reaches the end of current partition file. Current bucket number (`bktNumPtr`), current page number in the partition file of left table (`leftBktPageNum`), and the offset of data for current tuple on current page (`leftBktPageOffset`) are stored as private fields to locate next tuple from left table if exists.

Repeats above steps until all buckets of right (smaller) table have been processed.

```
2. void HashJoin::getAttributes(vector<Attribute> &attrs) const
```

Directly concatenates attributes from left and right input Iterators.