CSE 562: Project 2 (4 pages)

Due Date: Anytime before the end of Exam Period: Wednesday May 15 (also see the extra credit assignment below)
Groups: You may work in groups of 2-3 people.
Submission Instructions: Compress (as tar.gz or zip) a single directory containing:

- All of the your Java Source files for the project, including those provided as part of the project itself (they may be organized in subdirectories).
- A file named TEAM containing each team member's UBIT, one per line.

Have one group member upload your file to
`timberlake.cse.buffalo.edu`, then log in and run the command:

```
/util/bin/submit_cse562 [filename]
```

Introduction

For this project, you will be using the TPC-H benchmarking workload. Let's start by getting that. Go to

`http://www.tpc.org/tpch/`

The TPCH-H benchmark is a standard benchmarking system for DBMSes, world-round. It's generally held to be the gold-standard for testing (analogous to BYTEMark for CPUs). You can download the latest version (2.15) from the column on the right side of the above URL.

You'll need the Specification document (PDF/DOC/DOCX), as well as the DBGen tool.

Compiling DBGen is a little tricky. You'll need gcc (or equivalent), and make. Instructions for Linix/UNIX machines (including OS X) are as follows:

- Make a copy of the file `makefile.suite`, name it `makefile`.
- Open up `makefile` and find the line that reads `CHANGE NAME OF ANSI COMPILER HERE`
- Modify the line that starts `CC =`
  Add `gcc` (or the name of your compiler) to the end
- Modify the line that starts `DATABASE =`
  Add `DB2` to the end

- Modify the line that starts `MACHINE =`
  Add your platform name from the list above. I've found that OS X works using the settings for `LINUX`.
- Modify the line that starts `WORKLOAD =`
  Add `TPCH` to the end
- If you're running on OSX, you'll need to make 2 quick changes to source files. Open the files `bm_utils.c` and `varsub.c` and comment out the line `#include <malloc.h>` (lines 71 and 44 respectively).
- Save the file, `cd` into the dbgen directory, and type `make`.

DBGen should now be compiled: Typing `./dbgen --help` should give you a list of command options. Play with it a bit. The one command you'll need for this project is:

```
./dbgen -s [size]
```

Here, `[size]` is the number of GB to make the raw database files. Let's start small: `./dbgen -s 0.1`. DBGen will run for a little whileand then you should have several files labeled `[filename].tbl` in the current directory.

These files correspond to the set of tables defined in the TPCH Specification. These tables are depicted in an ER-ish form in a diagram on page 13 (in the PDF). Additional detail, including full schema information, is provided starting on page 14.

Let's have a look at a few of them. Open up `orders.tbl`, which contains data for the Orders relation[1]. The first few lines should look something like this:

```
1|3691|O|194029.55|1996-01-02|5-LOW|Clerk#000000951|0|nstructions sleep furiously among |
2|7801|O|60951.63|1996-12-01|1-URGENT|Clerk#000000880|0| foxes.  pending accounts at the pending, silent asymptot|
3|12332|F|247296.05|1993-10-14|5-LOW|Clerk#000000955|0|sly final accounts boost.  carefully regular ideas cajole carefully.  depos|
```

The pipe character '`|`' separates each field. Each datatype used by TPC-H can be maps to one four formats: integer, float, string, and date. Dates may be represented as integers using the following trick. SQL implements dates in the form '`YYYY-MM-DD`'. You can represent dates as the integer formed by concatenating these values. For example, the date `1996-01-02` can be represented as the integer 19960102. This representation preserves comparisons (i.e., $19960102 > 19950102$), but does not generally support addition (i.e., $19990228 + 1 \neq 19960301$). Because of this, **date values will be hardcoded transformed to their corresponding integer values in queries**.

[1] The name of this table is actually somewhat special. Why is this table, and no other table named in the plural form? (2 bonus points on the project if you can figure it out – put your **MAX ONE SENTENCE** answer in a file named "ORDERS")

Part 1: Integrate with TPC-H (20 points)

TPC-H defines a total of 22 benchmark query templates (starting on page 29 of the specification). Each template query contains one or more variables (specified in brackets, for example `[DELTA]`, for Q1). At the bottom of the query definition is a specification for how these variables are assigned values.

For the purposes of this test, we will use a subset of the queries (as many of the queries require SQL features that we have not yet implemented). In the `test` directory you will find SQL for TPC-H benchmark queries: **Q1, Q3, Q5, Q6, Q10, Q19** (ignore the query files with names ending in LIMIT for now).

Recall the `USING` clause of the old `CREATE TABLE` syntax. Originally, we only supported one data format: `CSV`. Now we need to add support for TPC-H. You'll note that the new test queries have a new value for their using clause: `USING TPCH('[tablename]')`, where `[tablename]` is one of the 8 TPC-H tables.

As written, the five queries each assume that the data files (the .tbl files created by dbgen) are all located in the current directory.

Your goal for this part of the project is to extend your SQL Parser with functionality to support the TPC-H queries.

Testing: Modify your `edu.buffalo.cse.sql.Sql` class and add a main() function that evaluates a SQL file provided on the command line. For example, if I run (from the command line):

```
java -cp build edu.buffalo.cse.sql.Sql test/TPCH_Q1.SQL
```

I should get a print out of all of the results of running Q1 on the .tbl files in the current directory.

You should use the utility class `sql.util.TableBuilder` to format your output. A sample invocation of TableBuilder may be found in the comments at the top of the file.

Part 2: Query Rewriting (40 points)

Now we will add a generic framework for doing optimization through simple, localized rewrites to the logical plan (PlanNode and its descendants). You must implement, at a minimum, the following two optimizations for this part:

- Push-Down Selects - As discussed in class, in the homework, on the midterm, and in Chapter 15 of the book, SelectionNode operators should be pushed down and through joins, aggregation predicates, and other operations wherever possible. Use the equivalencies in section 15.3 (and specifically 15.3.4) when designing your solution to this problem.

  For example, consider TPC-H Query 3. This query has at least one selection predicate on each of the 3 tables that appear in the query. After rewriting, your query plan should have these selection predicates directly around the table.

- Hybrid Hash Joins OR Sort-Merge Joins - You should implement a Hybrid Hash Join or a Sort-Merge join. You may use purely in-memory constructs for this part (i.e., you may use Java's HashTable to implement the HHJ Index). By rewriting the logical plan, you should identify joins that are candidates for a HHJ or SMJ and modify the logical plan to denote this. For the Sort-Merge Join, you may also want to implement a Sort operator – You may find that it is more efficient to implement this as an External Sort (The Buffer Pool might be of use here), but doing so is not required.

  Note that **all** joins in the TPC-H workload provided are equi-joins on a single attribute, and thus ideal candidates for both HHJ or SMJ. Many of the joins in the original validation workload (e.g., `TABLE04.SQL`) are also ideal candidates for testing this.

  Also note that you may need to change or create subclasses of classes in the sql.plan package to implement this portion of the project.

Each of these transformations may be implemented as localized rewrites on the logical plan. For example, to implement the Hybrid Hash Join rewrite, we could search for instances of the following pattern in the logical plan tree:

$$\sigma_{A=B}(Q_1 \times Q_2)$$

Every such instance could then be replaced by

$$Q_1 \bowtie_{HHJ:A=B} Q_2$$

This replacement should occur regardless of where the pattern occurs in the expression tree. For example, if we had the more complex query

$$\pi_{T.C}(\sigma_{R.A=S.B}(R \times (S \times T)) \times U)$$

Then, using $Q_1 := R$, and $Q_2 = (S \times T)$ we would come up with the rewritten expression

$$\pi_{T.C}(R \bowtie_{HHJ:R.A=S.B} (S \times T)) \times U)$$

To perform a rewrite like this, you will need to visit every node in the tree to see if it and its immediate descendants match the specified pattern. If they do, they can be rewritten. An additional utility class has been provided for your use in this part of the project: `sql.optimizer.PlanRewrite` that should simplify this process.

Concrete subclasses of PlanRewrite each instantiate an apply() method. For example, we could define a subclass of PlanRewrite called `PushDownSelects`. When rewrite(Q) is invoked on an instance of PushDownSelects, apply() will be called on every node of Q. apply() should return a rewritten version of Q, or the original version of Q if no rewrites are possible on Q.

For example we can implement hybrid hash join rewrite described above through an apply method that did the following (with `node` as its input).

- Is `node` a select node?
- If so, is it a conjunction of terms, any of which are an equality predicates ($A = B$)?
- If so, is its immediate child a join node (a nested-loop join to be specific)?
- If so, are any of the equality predicates mentioned earlier suitable candidates for a join predicate?
- If so, return a new SelectNode (without that predicate), and make its child a HybridHashJoin.
- If any of the above conditions are false, return `node` unchanged.

Testing: Extend your main function to accept the `-explain` flag. This flag should cause your main function to print out the relational algebra plan (using toString()) for the provided SQL both before and after optimization (instead of running the query). For example, running:

```
java -cp build edu.buffalo.cse.sql.Sql -explain test/TPCH_Q1.SQL
```

Should produce something like:

```
=== Before ===
AGGREGATE [returnflag: l.returnflag, linestatus: l.linestatus, sum_qty:
SUM(l.quantity), sum_base_price: SUM(l.extendedprice), sum_disc_price:
SUM((l.extendedprice * (1 - l.discount))), sum_charge: SUM(((l.extendedprice
* (1 - l.discount)) * (1 + l.tax))), avg_qty: AVG(l.quantity), avg_price:
AVG(l.extendedprice), avg_disc: AVG(l.discount), count_order: COUNT(1)] {
  SELECT[(l.shipdate <= 19981101)] {
    SCAN [lineitem(orderkey, partkey, suppkey, linenumber, quantity,
extendedprice, discount, tax, returnflag, linestatus, shipdate, commitdate,
receiptdate, shipinstruct, shipmode, comment)]
  }
}
=== After ===
AGGREGATE [returnflag: l.returnflag, linestatus: l.linestatus, sum_qty:
SUM(l.quantity), sum_base_price: SUM(l.extendedprice), sum_disc_price:
SUM((l.extendedprice * (1 - l.discount))), sum_charge: SUM(((l.extendedprice
* (1 - l.discount)) * (1 + l.tax))), avg_qty: AVG(l.quantity), avg_price:
AVG(l.extendedprice), avg_disc: AVG(l.discount), count_order: COUNT(1)] {
```

```
  INDEXSCAN [shipdate <= 19981101; lineitem(orderkey, partkey, suppkey,
linenumber, quantity, extendedprice, discount, tax, returnflag, linestatus,
shipdate, commitdate, receiptdate, shipinstruct, shipmode, comment)]
}
```

A visual inspection of the After query should show selection predicates pushed down and all joins replaced by Hybrid Hash Joins or Sort-Merge Joins. In addition to the TPC-H queries, query rewriting functionality will be tested on several of the original validation queries, and in particular the TABLEXX.SQL queries with equality predicates.

Part 3: Index Use (40 points)

For the final part of this assignment, you should apply query rewrites to take advantage of index scans. This means two things. First, you will need to select a set of indices to build for each relation. Extend your main function to accept the `-index` flag, which indicates that you should build indexes for all tables in the specified query. For example, running:

```
java -cp build edu.buffalo.cse.sql.Sql -index test/TPCH_Q1.SQL
```

should build indexes for the Lineitem relation (the only relation to appear in Q1. You should manually select an appropriate set of indexes to be built for each table. You may hardcode these values into the query processor, or identify a more generalizable way of doing so (you may modifying the query language and the TPC-H queries themselves to do so).

You should implement an INDEX SCAN operator and an Index-Nested-Loop join operator using your ISAMIndex and HashIndex classes, and modify your query optimizer to inject these operations where appropriate.

Testing: Using the `-explain` flag as above should show the INDEX SCAN and/or Index-Nested-Loop JOIN operators being used where appropriate.

Extra Credit: Sort/Limit (Up to 20 points)

20 bonus points for implementing a Sort and Limit operator, and successfully evaluating the `_LIMIT` version of each query that has one.

Extra Credit: Need for Speed (Up to 30 points)

Starting Wednesday April 10 at 11:59 PM, and every Sunday and Wednesday until exam week **starts**, I will run a competition between the most recent submissions from each group. Details of the competition workloads will be released shortly.

Scoring for the competition will be based on (1) Time to completion (TTC), and (2) Maximum memory (MM) consumed. Your team's overall score for each category will be based on the median value in each according the the following formula:

$$Score_i = \sqrt{\left(\frac{TTC_i}{Median(TTC)}\right)^2 + \left(\frac{MM_i}{Median(MM)}\right)^2}$$

Lower scores are better. Time to completion will be measured after indexes have been constructed.

The 5 teams with the lowest (best) overall score for each competition will receive 15 extra credit points on this assignment (to a maximum of 30).

(Optional) Project 4 (Up to 100 points)

Project 4 is now optional. For this extra credit assignment, you will (1) select a specialized feature to incorporate into your query processor, (2) develop a strategy for validating the performance of your additional feature, (3) test and document the performance of this feature. Examples of appropriate features include:

1. Single-Threaded Support for Updates with Incrementally Maintained Views.
2. Support for Streaming Selects (Window Joins and Semi-Joins)
3. Transactional Support for Updates
4. Translating from relational algebra operators to a non-java backend (e.g., Map/Reduce, LLVM, etc...).
5. A Disk-Resident Data Cube
6. Threading support for SELECTs (i.e., read-only queries) – A threadsafe buffer pool and scheduling.
7. Parallel execution of SELECT queries.
8. Given a query (or a set of queries), Select an appropriate set of indexes for this workload.
9. Suggest your own project!

Before attempting the optional project 4, you should consult with the professor and arrive at a suitable project description. Schedule a meeting at:

`http://doodle.com/okennedy`