Checkpoint2

- **Overview**: Optimize your implementation to support specialized join algorithms and limited memory.
- **Deadline**: March 30
- **Grade**: 15% of Project Component
  - 5% Correctness
  - 5% Efficiency
  - 5% Code Review

This project is, in effect, a more rigorous form of Project 1. The requirements are identical: We give you a query and some data, you evaluate the query on the data and give us a response as quickly as possible.

First, this means that we'll be expecting a more feature-complete submission. Your code will be evaluated on more queries from TPC-H benchmark, which exercises a broader range of SQL features than the Project 1 test cases did.

Second, performance constraints will be tighter. The reference implementation for this project has been improved over that of Project 1, meaning that you'll be expected to perform more efficiently, and to handle data that does not fit into main memory.

Join Ordering

The order in which you join tables together is **incredibly important**, and can change the runtime of your query by **multiple orders of magnitude**. Picking between different join orderings is incredibly important! However, to do so, you will need statistics about the data, something that won't really be feasible until the next project. Instead, here's a present for those of you paying attention. The tables in each FROM clause are ordered so that you will get our recommended join order by building a *left-deep plan* going in-order of the relation list (something that many of you are doing already), and (for hybrid hash joins) using the left-hand-side relation to build your hash table.

Blocking Operators and Memory

Blocking operators (e.g., joins other than Merge Join, the Sort operator, etc…) are generally blocking because they need to materialize instances of a relation. For half of this project, you will not have enough memory available to materialize a full relation, to say nothing of join results. To successfully process these queries, you will need to implement

out-of core equivalents of these operators: At least one External Join (e.g., Block-Nested-Loop, Hash, or Sort/Merge Join) and an out-of-core Sort Algorithm (e.g., External Sort). For your reference, the evaluation machines have 2GB of memory. In phase 2, Java will be configured for 1**00 MB of heap space** (see the command line argument -Xmx). To work with such a small amount of heap space, **you will need to manually invoke Java's garbage collector** by calling `System.gc()`. How frequently you do this is up to you. The more you wait, the greater the chance that you'll run out of memory. The reference implementation calls it in the Two-Phase sort operator, every time it finishes flushing a file out to disk.

Query Rewriting

In Project 1, you were encouraged to parse SQL into a relational algebra tree. Project 2 is where that design choice begins to pay off. We've discussed expression equivalences in relational algebra, and identified several that are always good (e.g., pushing down selection operators). The reference implementation uses some simple recursion to identify patterns of expressions that can be optimized and rewrite them. For example, if I wanted to define a new HashJoin operator, I might go through and replace every qualifying Selection operator sitting on top of a CrossProduct operator with a HashJoin.

```
if(o instanceof Selection){
  Selection s = (Selection)o;
  if(s.getChild() instanceof CrossProduct){
    CrossProduct prod =
      (CrossProduct)s.getChild();
    Expression join_cond =
      // find a good join condition in
      // the predicate of s.
    Expression rest =
      // the remaining conditions
    return new Selection(
      rest,
      new HashJoin(
        join_cond,
        prod.getLHS(),
        prod.getRHS()
      )
    );
  }
}
return o;
```

The reference implementation has a function similar to this snippet of code, and applies the function to every node in the relational algebra tree.

Because selection can be decomposed, you may find it useful to have a piece of code that can split AndExpressions into a list of conjunctive terms:

```
List<Expression> splitAndClauses(Expression e)
{
  List<Expression> ret =
     new ArrayList<Expression>();
  if(e instanceof AndExpression){
    AndExpression a = (AndExpression)e;
    ret.addAll(
      splitAndClauses(a.getLeftExpression())
    );
    ret.addAll(
      splitAndClauses(a.getRightExpression())
    );
  } else {
    ret.add(e);
  }
}
```

Interface

Your code will be evaluated in exactly the same way as Project 1. Your code will be presented with a 1GB (SF 1) TPC-H dataset. Grading will proceed in two phases. In the first phase, you will have an unlimited amount of memory, but very tight time constraints. In the second phase, you will have slightly looser time constraints, but will be limited to 100 MB of memory, and presented with either a 1GB or a 200 MB (SF 0.2) dataset.

As before, your code will be invoked with the data directory and the relevant SQL files. An additional parameter will be used in Phase 2:

- `--swap directory`: A swap directory that you're allowed to write to. No other directories are guaranteed to be available or writeable. If the –swap parameter is not present, you should not swap (i.e., the data size is small enough to be handled entirely in-memory)

```
java -cp build:jsqlparser.jar
```

```
-Xmx100m      # Heap limit (Phase 2 only)

edu.buffalo.cse562.Main

--data [data]

--swap [swap]

[sqlfile1] [sqlfile2] ...
```

This example uses the following directories and files:

- `[data]`: Table data stored in '|' separated files. As before, table names match the names provided in the matching CREATE TABLE with the .dat suffix.
- `[swap]`: A temporary directory for an individual run. This directory will be emptied after every trial.
- `[sqlfileX]`: A file containing CREATE TABLE and SELECT statements, defining the schema of the dataset and the query to process

Grading

Your code will be subjected to a sequence of test cases and evaluated on speed and correctness. Note that unlike Project 1, you will neither receive a warning about, nor partial credit for out-of-order query results if the outermost query includes an ORDER BY clause.

Phase 1 (big queries) will be graded on a TPC-H SF 1 dataset (1 GB of raw text data). Phase 2 (limited memory) will be graded on either a TPC-H SF 1 or SF 0.2 (200 MB of raw text data) dataset as listed in the chart below. Grades are assigned based on per-query thresholds:

- **0/10 (F)**: Your submission does not compile, does not produce correct output, or fails in some other way. Resubmission is highly encouraged.
- **5/10 (C)**: Your submission runs the test query faster than the C threshold (listed below for each query), and produces the correct output.

- **7.5/10 (B)**: Your submission runs the test query faster than the B threshold (listed below for each query), and produces the correct output.
- **10/10 (A)**: Your submission runs the test query faster than the A threshold (listed below for each query), and produces the correct output.

TPC-H QUERY	PHASE 1 RUNTIMES		PHASE 2 RUNTIMES	PHASE 2 SCALING FACTOR
	45 s	**A**	1 min	
	67.5 s	**B**	2 min	
1	90 s	**C**	3 min	SF = 0.2
	45 s	**A**	40 s	
	90 s	**B**	80 s	
3	120 s	**C**	120 s	SF = 0.2
	45 s	**A**	70 s	
	90 s	**B**	140 s	
5	120 s	**C**	210 s	SF = 0.2
	45 s	**A**	2 min	
	67.5 s	**B**	4 min	
10	90 s	**C**	6 min	SF = 1
	45 s	**A**	1.5 min	
	67.5 s	**B**	3 min	
12	90 s	**C**	4.5 min	SF = 1