query-optimization.md

CS-339 Lab 3: Query Optimization

Assigned: Thursday, February 17, 2022

Due: Tuesday, March 1, 2022 11:59PM CST

In this lab, you will implement a query optimizer on top of SimpleDB. The main tasks include implementing a selectivity estimation framework and a cost-based optimizer. You have freedom as to exactly what you implement, but we recommend using something similar to the Selinger cost-based optimizer discussed in class.

The remainder of this document describes what is involved in adding optimizer support and provides a basic outline of how you might add this support to your database.

As with the previous lab, we recommend that you start as early as possible.

1. Getting started

Download the starter code from Canvas. If you set up an IDE for Lab 2, you may wish to replace the contents of your lab directory with the starter code.

1.1. Implementation hints

We suggest exercises along this document to guide your implementation, but you may find that a different order makes more sense for you. As before, we will grade your assignment by looking at your code and verifying that you have passed the test for the ant targets `test` and `systemtest`. See Section 3.4 for a complete discussion of grading and the tests you will need to pass.

Here's a rough outline of one way you might proceed with this lab. More details on these steps are given in Section 2 below.

- Implement the methods in the `TableStats` class that allow it to estimate selectivities of filters and cost of scans, using histograms (skeleton provided for the `IntHistogram` class) or some other form of statistics of your devising.
- Implement the methods in the `JoinOptimizer` class that allow it to estimate the cost and selectivities of joins.
- Write the `orderJoins` method in `JoinOptimizer`. This method must produce an optimal ordering for a series of joins (likely using the Selinger algorithm), given statistics computed in the previous two steps.

2. Optimizer outline

Recall that the main idea of a cost-based optimizer is to:

- Use statistics about tables to estimate "costs" of different query plans. Typically, the cost of a plan is related to the cardinalities of (number of tuples produced by) intermediate joins and selections, as well as the selectivity of filter and join predicates.
- Use these statistics to order joins and selections in an optimal way, and to select the best implementation for join algorithms from amongst several alternatives.

In this lab, you will implement code to perform both of these functions.

Briefly, if you have a catalog file
`catalog.txt` describing your tables, you can run the parser by typing:

```
java -jar dist/simpledb.jar parser catalog.txt
```

When the Parser is invoked, it will compute statistics over all of the tables (using statistics code you provide). When a query is issued, the parser will convert the query into a logical plan representation and then call your query optimizer to generate an optimal plan.

2.1 Overall Optimizer Structure

Before getting started with the implementation, you need to understand the overall structure of the SimpleDB optimizer. The overall control flow of the SimpleDB modules of the parser and optimizer is shown in Figure 1.

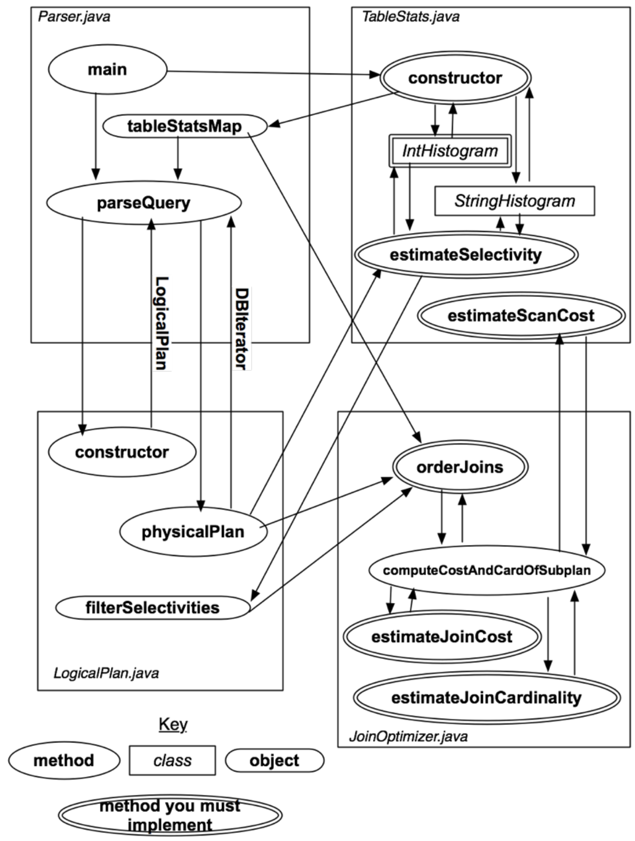

Figure 1: Diagram illustrating classes, methods, and objects used in the parser

The key at the bottom explains the symbols; you will implement the components with double-borders. The classes and methods will be explained in more detail in the text that follows (you may wish to refer back to this diagram), but the basic operation is as follows:

1. `Parser.java` constructs a set of table statistics (stored in the `statsMap` container) when it is initialized. It then waits for a query to be input, and calls the method `parseQuery` on that query.
2. `parseQuery` first constructs a `LogicalPlan` that represents the parsed query. `parseQuery` then calls the method `physicalPlan` on the `LogicalPlan` instance it has constructed. The `physicalPlan` method returns a `DBIterator` object that can be used to actually run the query.

In the exercises to come, you will implement the methods that help `physicalPlan` devise an optimal plan.

2.2. Statistics Estimation

Accurately estimating plan cost is quite tricky. In this lab, we will focus only on the cost of sequences of joins and base table accesses. We won't worry about access method selection (since we only have one access method, table scans) or the costs of additional operators (like aggregates).

You are only required to consider left-deep plans for this lab. See Section 2.3 for a description of additional "bonus" optimizer features you might implement, including an approach for handling bushy plans.

2.2.1 Overall Plan Cost

We will write join plans of the form `p=t1 join t2 join ... tn`, which signifies a left deep join where t1 is the left-most join (deepest in the tree). Given a plan like `p`, its cost can be expressed as:

```
scancost(t1) + scancost(t2) + joincost(t1 join t2) +
scancost(t3) + joincost((t1 join t2) join t3) +
...
```

Here, `scancost(t1)` is the I/O cost of scanning table t1, `joincost(t1,t2)` is the CPU cost to join t1 to t2. To make I/O and CPU cost comparable, typically a constant scaling factor is used, e.g.:

```
cost(predicate application) = 1
cost(pageScan) = SCALING_FACTOR x cost(predicate application)
```

For this lab, you can ignore the effects of caching (e.g., assume that every access to a table incurs the full cost of a scan) -- again, this is something you may add as an optional bonus extension to your lab in Section 2.3. Therefore, `scancost(t1)` is simply the number of pages in `t1 x SCALING_FACTOR`.

2.2.2 Join Cost

When using nested loops joins, recall that the cost of a join between two tables t1 and t2 (where t1 is the outer) is simply:

```
joincost(t1 join t2) = scancost(t1) + ntups(t1) x scancost(t2) //IO cost
                       + ntups(t1) x ntups(t2)  //CPU cost
```

Here, `ntups(t1)` is the number of tuples in table t1.

2.2.3 Filter Selectivity

`ntups` can be directly computed for a base table by scanning that table. Estimating `ntups` for a table with one or more selection predicates over it can be trickier -- this is the *filter selectivity estimation* problem. Here's one approach that you might use, based on computing a histogram over the values in the table:

- Compute the minimum and maximum values for every attribute in the table (by scanning it once).
- Construct a histogram for every attribute in the table. A simple approach is to use a fixed number of buckets *NumB*, with each bucket representing the number of records in a fixed range of the domain of the attribute of the histogram. For example, if a field *f* ranges from 1 to 100, and there are 10 buckets, then bucket 1 might contain the count of the number of records between 1 and 10, bucket 2 a count of the number of records between 11 and 20, and so on.
- Scan the table again, selecting out all of fields of all of the tuples and using them to populate the counts of the buckets in each histogram.
- To estimate the selectivity of an equality expression, *f=const*, compute the bucket that contains value *const*. Suppose the width (range of values) of the bucket is *w*, the height (number of tuples) is *h*, and the number of tuples in the table is *ntups*. Then, assuming values are uniformly distributed throughout the bucket, the selectivity of the expression is roughly *(h / w) / ntups*, since *(h/w)* represents the expected number of tuples in the bin with value *const*.
- To estimate the selectivity of a range expression *f>const*, compute the bucket *b* that *const* is in, with width *w_b* and height *h_b*. Then, *b* contains a fraction *b_f = h_b / ntups* of the total tuples. Assuming tuples are uniformly distributed throughout *b*, the fraction *b_part* of *b* that is *> const* is *(b_right - const) / w_b*, where *b_right* is the right endpoint of *b*'s bucket. Thus, bucket *b* contributes *(b_f x b_part)* selectivity to the predicate. In addition, buckets *b+1...NumB-1* contribute all of their selectivity (which can be computed using a formula similar to *b_f* above). Summing the selectivity contributions of all the buckets will yield the overall selectivity of the expression. Figure 2 illustrates this process.
- Selectivity of expressions involving *less than* can be performed similar to the greater than case, looking at buckets down to 0.

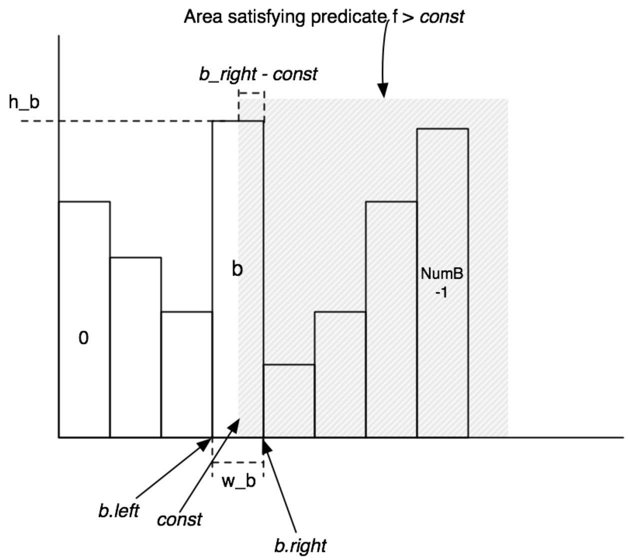

Figure 2: Diagram illustrating the histograms you will implement in Lab 5

In the next two exercises, you will code to perform selectivity estimation of joins and filters.

Exercise 1: IntHistogram.java

You will need to implement some way to record table statistics for selectivity estimation. We have provided a skeleton class, `IntHistogram` that will do this. Our intent is that you calculate histograms using the bucket-based method described above, but you are free to use some other method so long as it provides reasonable selectivity estimates.

We have provided a class `StringHistogram` that uses `IntHistogram` to compute selecitivites for String predicates. You may modify `StringHistogram` if you want to implement a better estimator, though you should not need to in order to complete this lab.

After completing this exercise, you should be able to pass the `IntHistogramTest` unit test (you are not required to pass this test if you choose not to implement histogram-based selectivity estimation).

Exercise 2: TableStats.java

The class `TableStats` contains methods that compute the number of tuples and pages in a table and that estimate the selectivity of predicates over the fields of that table. The query parser we have created creates one instance of `TableStats` per table, and passes these structures into your query optimizer (which you will need in later exercises).

You should fill in the following methods and classes in `TableStats` :

- Implement the `TableStats` constructor: Once you have implemented a method for tracking statistics such as histograms, you should implement the `TableStats` constructor, adding code to scan the table (possibly multiple times) to build the statistics you need.
- Implement `estimateSelectivity(int field, Predicate.Op op, Field constant)` : Using your statistics (e.g., an `IntHistogram` or `StringHistogram` depending on the type of the field), estimate the selectivity of predicate `field op constant` on the table.
- Implement `estimateScanCost()` : This method estimates the cost of sequentially scanning the file, given that the cost to read a page is `costPerPageIO` . You can assume that there are no seeks and that no pages are in the buffer pool. This method may use costs or sizes you computed in the constructor.
- Implement `estimateTableCardinality(double selectivityFactor)` : This method returns the number of tuples in the relation, given that a predicate with selectivity selectivityFactor is applied. This method may use costs or sizes you computed in the constructor.

You may wish to modify the constructor of `TableStats.java` to, for example, compute histograms over the fields as described above for purposes of selectivity estimation.

After completing these tasks you should be able to pass the unit tests in `TableStatsTest` .

2.2.4 Join Cardinality

Finally, observe that the cost for the join plan `p` above includes expressions of the form `joincost((t1 join t2) join t3)` . To evaluate this expression, you need some way to estimate the size (`ntups`) of `t1 join t2` . This *join cardinality estimation* problem is harder than the filter selectivity estimation problem. In this lab, you aren't required to do anything fancy for this, though one of the optional excercises in Section 2.4 includes a histogram-based method for join selectivity estimation.

While implementing your simple solution, you should keep in mind the following:

- For equality joins, when one of the attributes is a primary key, the number of tuples produced by the join cannot be larger than the cardinality of the non-primary key attribute.
- For equality joins when there is no primary key, it's hard to say much about what the size of the output is -- it could be the size of the product of the cardinalities of the tables (if both tables have the same value for all tuples) -- or it could be 0. It's fine to make up a simple heuristic (say, the size of the larger of the two tables).
- For range scans, it is similarly hard to say anything accurate about sizes. The size of the output should be proportional to the sizes of the inputs. It is fine to assume that a fixed fraction of the cross-product is emitted by range scans (say, 30%). In general, the cost of a range join should be larger than the cost of a non-primary key equality join of two tables of the same size.

Exercise 3: Join Cost Estimation

The class `JoinOptimizer.java` includes all of the methods for ordering and computing costs of joins. In this exercise, you will write the methods for estimating the selectivity and cost of a join, specifically:

- Implement `estimateJoinCost(LogicalJoinNode j, int card1, int card2, double cost1, double cost2)` : This method estimates the cost of join j, given that the left input is of cardinality card1, the right input of cardinality card2, that the cost to scan the left input is cost1, and that the cost to access the right input is card2. You can assume the join is an NL join, and apply the formula mentioned earlier.
- Implement `estimateJoinCardinality(LogicalJoinNode j, int card1, int card2, boolean t1pkey, boolean t2pkey)` : This method estimates the number of tuples output by join j, given that the left input is size card1, the right input is size card2, and the flags t1pkey and t2pkey that indicate whether the left and right (respectively) field is unique (a primary key).

After implementing these methods, you should be able to pass the unit tests `estimateJoinCostTest` and `estimateJoinCardinality` in `JoinOptimizerTest.java` .

2.3 Join Ordering

Now that you have implemented methods for estimating costs, you will implement the Selinger optimizer. For these methods, joins are expressed as a list of join nodes (e.g., predicates over two tables) as opposed to a list of relations to join as described in class.

Translating the algorithm given in lecture to the join node list form mentioned above, an outline in pseudocode would be:

```
j = set of join nodes
for (i in 1...|j|):
    for s in {all length i subsets of j}
      bestPlan = {}
      for s' in {all length d-1 subsets of s}
```

```
6.           subplan = optjoin(s')
7.           plan = best way to join (s-s') to subplan
8.           if (cost(plan) < cost(bestPlan))
9.              bestPlan = plan
10.      optjoin(s) = bestPlan
11. return optjoin(j)
```

To help you implement this algorithm, we have provided several classes and methods to assist you. First, the method `enumerateSubsets(Vector v, int size)` in `JoinOptimizer.java` will return a set of all of the subsets of `v` of size `size`. This method is not particularly efficient; you can earn extra credit by implementing a more efficient enumerator.

Second, we have provided the method:

```
private CostCard computeCostAndCardOfSubplan(HashMap<String, TableStats> stats,
                                             HashMap<String, Double> filterSelectivities,
                                             LogicalJoinNode joinToRemove,
                                             Set<LogicalJoinNode> joinSet,
                                             double bestCostSoFar,
                                             PlanCache pc)
```

Given a subset of joins (`joinSet`), and a join to remove from this set (`joinToRemove`), this method computes the best way to join `joinToRemove` to `joinSet - {joinToRemove}`. It returns this best method in a `CostCard` object, which includes the cost, cardinality, and best join ordering (as a vector). `computeCostAndCardOfSubplan` may return null, if no plan can be found (because, for example, there is no left-deep join that is possible), or if the cost of all plans is greater than the `bestCostSoFar` argument. The method uses a cache of previous joins called `pc` (`optjoin` in the psuedocode above) to quickly lookup the fastest way to join `joinSet - {joinToRemove}`. The other arguments (`stats` and `filterSelectivities`) are passed into the `orderJoins` method that you must implement as a part of Exercise 4, and are explained below. This method essentially performs lines 6--8 of the psuedocode described earlier.

Third, we have provided the method:

```
private void printJoins(Vector<LogicalJoinNode> js,
                        PlanCache pc,
                        HashMap<String, TableStats> stats,
                        HashMap<String, Double> selectivities)
```

This method can be used to display a graphical representation of a join plan (when the "explain" flag is set via the "-explain" option to the optimizer, for example).

Fourth, we have provided a class `PlanCache` that can be used to cache the best way to join a subset of the joins considered so far in your implementation of Selinger (an instance of this class is needed to use `computeCostAndCardOfSubplan`).

Exercise 4: Join Ordering

In `JoinOptimizer.java` , implement the method:

```
Vector<LogicalJoinNode> orderJoins(HashMap<String, TableStats> stats,
                 HashMap<String, Double> filterSelectivities,
                 boolean explain)
```

This method should operate on the `joins` class member, returning a new Vector that specifies the order in which joins should be done. Item 0 of this vector indicates the left-most, bottom-most join in a left-deep plan. Adjacent joins in the returned vector should share at least one field to ensure the plan is left-deep. Here `stats` is an object that lets you find the `TableStats` for a given table name that appears in the `FROM` list of the query. `filterSelectivities` allows you to find the selectivity of any predicates over a table; it is guaranteed to have one entry per table name in the `FROM` list. Finally, `explain` specifies that you should output a representation of the join order for informational purposes.

You may wish to use the helper methods and classes described above to assist in your implementation. Roughly, your implementation should follow the psuedocode above, looping through subset sizes, subsets, and sub-plans of subsets, calling `computeCostAndCardOfSubplan` and building a `PlanCache` object that stores the minimal-cost way to perform each subset join.

After implementing this method, you should be able to pass all the unit tests in `JoinOptimizerTest`. You should also pass the system test `QueryTest`.

2.4 Extra Credit

In this section, we describe several optional excercises that you may implement for extra credit. These are less well defined than the previous exercises but give you a chance to show off your mastery of query optimization!

Bonus Exercises. Each of these bonuses is worth up to 5% extra credit:

- *Add code to perform more advanced join cardinality estimation*. Rather than using simple heuristics to estimate join cardinality, devise a more sophisticated algorithm.
- One option is to use joint histograms between every pair of attributes *a* and *b* in every pair of tables *t1* and *t2*. The idea is to create buckets of *a*, and for each bucket *A* of *a*, create a histogram of *b* values that co-occur with *a* values in *A*.
- Another way to estimate the cardinality of a join is to assume that each value in the smaller table has a matching value in the larger table. Then the formula for the join selectivity would be: 1/(*Max*(*num-distinct*(t1, column1), *num-distinct*(t2, column2))). Here, column1 and column2 are the join attributes. The cardinality of the join is then the product of the cardinalities of *t1* and *t2* times the selectivity.
- *Improved subset iterator*. Our implementation of `enumerateSubsets` is quite inefficient, because it creates a large number of Java objects on each invocation. A better approach would be to implement an iterator that, for example, returns a `BitSet` that specifies the elements in the `joins` vector that should be accessed on each iteration. In this bonus exercise, you would improve the performance of `enumerateSubsets` so that your system could perform query optimization on plans with 20 or more joins (currently such plans takes minutes or hours to compute).
- *A cost model that accounts for caching*. The methods to estimate scan and join cost do not account for caching in the buffer pool. You should extend the cost model to account for caching effects. This is tricky because multiple joins are running simultaneously due to the iterator model, and so it may be hard to predict how much memory each will have access to using the simple buffer pool we have implemented in previous labs.
- *Improved join algorithms and algorithm selection*. Our current cost estimation and join operator selection algorithms (see `instantiateJoin()` in `JoinOptimizer.java`) only consider nested loops joins. Extend these methods to use one or more additional join algorithms (for example, some form of in memory hashing using a `HashMap`).
- *Bushy plans*. Improve the provided `orderJoins()` and other helper methods to generate bushy joins. Our query plan generation and visualization algorithms are perfectly capable of handling bushy plans; for example, if `orderJoins()` returns the vector (t1 join t2 ; t3 join t4 ; t2 join t3), this will correspond to a bushy plan with the (t2 join t3) node at the top.

You have now completed this lab. Good work!

3. Logistics

You must submit your code (see below) as well as a short (2 pages, maximum) writeup describing your approach. This writeup should:

- Describe any design decisions you made, including methods for selectivity estimation, join ordering, as well as any of the bonus exercises you chose to implement and how you implemented them (for each bonus exercise you may submit up to 1 additional page).
- Discuss and justify any changes you made to the API.
- Describe any missing or incomplete elements of your code.
- Describe how long you spent on the lab, and whether there was anything you found particularly difficult or confusing.

3.1. Collaboration

This lab should be manageable for a single person, but if you prefer to work with a partner, this is also OK. Larger groups are not allowed. Please indicate clearly who you worked with, if anyone, ub write-up.

3.2. Submitting your assignment

To submit your code, please create a `cs339-lab3.zip` tarball (such that, unzipped, it creates a `cs339-lab3/src/simpledb` directory with your code) and submit it on Canvas.

You may submit your code multiple times; we will use the latest version you submit that arrives before the deadline (before 11:59 PM on the due date). Place the write-up in a file called lab3-writeup.txt, which has been created for you in the top level of your simple-db-hw directory.

3.3. Submitting a bug

SimpleDB is a relatively complex piece of code. It is very possible you are going to find bugs, inconsistencies, and bad, outdated, or incorrect documentation, etc.

We ask you, therefore, to do this lab with an adventurous mindset. Don't get mad if something is not clear, or even wrong; rather, try to figure it out yourself or send us a friendly email.

Please submit (friendly!) bug reports to jennie@northwestern.edu. When you do, please try to include:

- A description of the bug.
- A `.java` file we can drop in the `test/simpledb` directory, compile, and run.
- A `.txt` file with the data that reproduces the bug. We should be able to convert it to a `.dat` file using `HeapFileEncoder`.

You can also post on the class page on Piazza if you feel you have run into a bug.

3.4 Grading

75% of your grade will be based on whether or not your code passes the test suite we will run over it. These tests will be a superset of the tests we have provided. Before handing in your code, you should make sure it produces no errors (passes all of the tests) from both `ant test` and `ant systemtest`.

Important: before testing, we will replace your `build.xml`, `HeapFileEncoder.java`, `BPlusTreeFileEncoder.java`, and the entire contents of the `test/` directory with our version of these files! This means you cannot change the format of `.dat` files! You should therefore be careful changing our APIs. This also means you need to test whether your code compiles with our test programs. In other words, we download your Canvas submissions, replace the files mentioned above, compile it, and then grade it. It will look roughly like this:

```
[replace build.xml, HeapFileEncoder.java, BPlusTreeFileEncoder.java and test]
$ ant test
$ ant systemtest
[additional tests]
```

If any of these commands fail, we'll be unhappy, and, therefore, so will your grade.

An additional 25% of your grade will be based on the quality of your writeup and our subjective evaluation of your code.

We've had a lot of fun designing this assignment, and we hope you enjoy hacking on it!