Assignment 4: Query Optimizer

Project Description

In this lab, you will implement a query optimizer on top of BuzzDB. 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.

Program Specification

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(estimate_join_cardinality, estimate_join_cost) in the JoinOptimizer class that allow it to estimate the cost and selectivities of joins. Write the order_joins 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.

Task #1 - 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. You are only required to consider left-deep plans for this lab.

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). Therefore, `scancost(t1)` is simply the number of pages in `t1 x SCALING_FACTOR`.

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.

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 range [illegible] 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 below illustrates this process.
- Selectivity of expressions involving less than can be performed similar to the greater than case, looking at buckets down to 0.

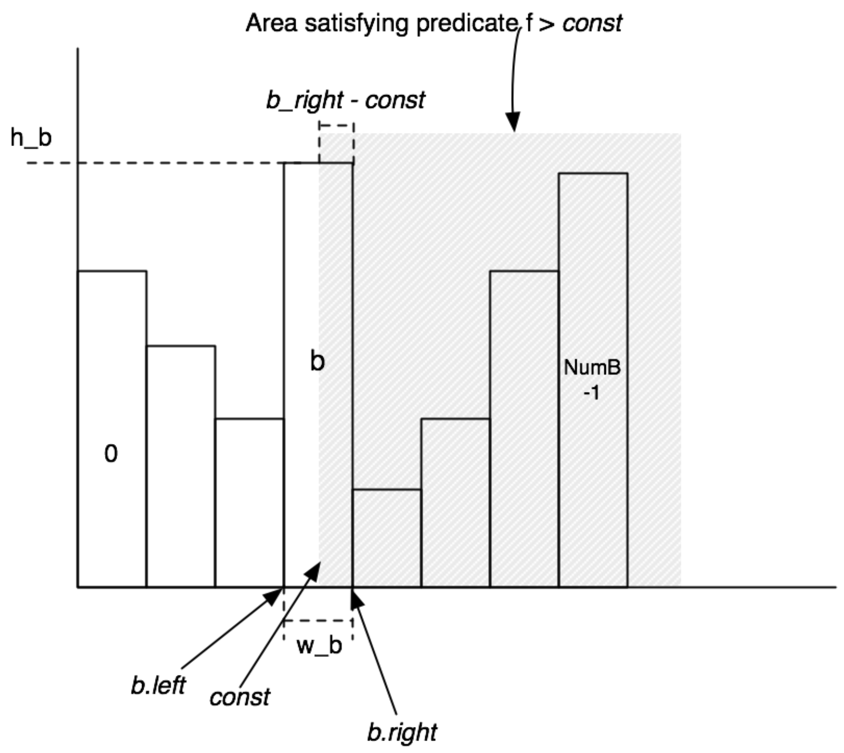

Diagram illustrating the histograms you will implement in Lab 4

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.

Exercise 1: 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.

Exercise 2: The class TableStats contains methods that compute the number of tuples in a table and that estimate the selectivity of predicates over the fields of that table.

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 estimate_selectivity(int64_t field, PredicateType op, int64_t constant): Using your statistics, estimate the selectivity of predicate field op constant on the table.
- Implement estimate_scan_cost(): This method estimates the cost of sequentially scanning the file, given that the cost to read a p `cost_per_page_IO`. You can assume that there are no seeks and that no pages are in the buffer pool. This method may use cos you computed in the constructor.
- Implement estimate_table_cardinality(double selectivity_factor): This method returns the number of tuples in the relation, given that a

predicate with selectivity selectivity_factor is applied. This method may use costs or sizes you computed in the constructor.

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.

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 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 `estimate_join_cost(LogicalJoinNode j, uint64_t card1, uint64_t card2, double cost1, double cost2, std::map<std::string, TableStats>& stats)`: 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 `estimate_join_cardinality(LogicalJoinNode j, uint64_t card1, uint64_t card2, bool t1pkey, bool t2pkey, std::map<std::string, TableStats>& stats)`: 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).

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).

Join Ordering Pseudocode

```
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}
           subplan = optjoin(s')
           plan = best way to join (s-s') to subplan
           if (cost(plan) < cost(bestPlan))
              bestPlan = plan
     optjoin(s) = bestPlan
return optjoin(j)
```

To help you implement this algorithm, we have provided several classes and methods to assist you. First, the method `enumerate_subsets(vector<LogicalJoinNode> v, int size)` in JoinOptimizer will return a set of all of the subsets of v of size size.

Second, we have provided the method:

```
compute_cost_and_card_of_subplan(
         std::map<std::string, TableStats> stats,
         std::map<std::string, double> filter_selectivities,
         LogicalJoinNode join_to_remove, std::set<LogicalJoinNode> join_set,
         double best_cost_so_far, PlanCache pc, CostCard& cc)
```

Given a subset of joins (join_set), and a join to remove from this set (join_to_remove), this method computes the best way to join join_to_remove to join_set - {join_to_remove}. It poulates this best method in a `CostCard object (cc)`, which includes the cost, cardinality, and best join ordering (as a vector). compute_cost_and_card_of_subplan may return false, if no plan can be found (because, for example and best join ordering (as a vector). compute_cost_and_card_of_subplan may return false, 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 best_cost_so_far argument. The method uses a cache of previous joins called pc (optjoin in the psuedocode above) to quickly lookup the fastest way to join join_set - {join_to_remove}. The other arguments (stats and filter_selectivities) are passed into the order_joins 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.

Exercise 4: Join Ordering In JoinOptimizer, implement the method:

```
std::vector<LogicalJoinNode> JoinOptimizer::order_joins(
            std::map<std::string, TableStats> stats,
            std::map<std::string, double> filter_selectivities)
```

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. `filter_selectivities` allows you to find the selectivity of any predicates over a table;

Prerequisites

You need to follow the instructions mentioned in the setup document.

Download the handout shared via Canvas.

Instructions for execution

```
[vm] $ cd buzzdb-query-optimizer
[vm] $ mkdir build
[vm] $ cd build
[vm] $ cmake -DCMAKE_BUILD_TYPE=Release ..
[vm] $ make
[vm] $ ctest
```

We treat compiler warnings as errors. Your project will fail to build if there are any compiler warnings.

General Instructions

Testing for correctness involves more than just seeing if a few test cases produce the correct output. There are certain types of errors (memory errors and memory leaks) that usually surface after the system has been running for a longer period of time. You should use *valgrind* to isolate such errors. Command to run *valgrind* can be found here.

You will get a listing of memory errors in your program. If you have programmed in Java you should keep in mind that C++ does not have automatic garbage collection, so each new must ultimately be matched by a corresponding delete. Otherwise all the memory in the system might be used up. Valgrind can be used to detect such memory leaks as well. More information about valgrind can be found at: http://www.valgrind.org/docs/manual/index.html.

Submitting Your Assignment

```
bash submit.sh <name>
```

You will be submitting your assignment on Gradescope. You are expected to run `submit.sh` and submit the generated zip to the autograder.

You can use `REPORT.md` to describe the following design and program criteria (**optional**). In case you don't complete all the testcases, we will award you partial points based on the report.

Grading

The maximum score on this assignment is 140. If you get 140 on the autograder that is your score. If you get score less than 140 we will award partial points based on the report.