Assignment 3: Table Statistics and Plan Costing

In this assignment you will complete the following features:

- Complete the implementation of statistics-collection from table data.
- Complete the plan-costing computations for various plan-nodes.
- Compute the selectivity of various predicates that can appear within execution plans.
- Perform some simple experiments with your implementations.

These tasks are described in the following sections.

There are also a few opportunities for extra credit described at the end of this assignment, if you want to supplement your grade.

Plan Costing Overview

Plan costing is a remarkably imprecise operation. Given a particular query execution plan, we must somehow make an estimate of how costly it will be to evaluate that plan, given statistics that are hopefully up to date, and a general understanding of how our plan nodes work in various cases. In addition, there can be several measures for how to represent the "cost" of a plan, such as the total number of rows produced at each plan node, the total time to produce the results, the time to produce the first result, the CPU/memory/disk usage of the plan, and so forth.

NanoDB has a very basic way of representing both plan costs and table statistics, but you will see that even with these simple representations, plan costing can be quite involved. Furthermore, we can only make costing estimates in very limited situations; in many other situations, we must simply use default guesses because there's no way to know.

Nonetheless, as imprecise as the whole effort is, it still gives our database the ability to choose a plan that is likely to be "better" than other plans. This allows us to generate multiple equivalent plans and then select the "best" one out of the bunch.

Statistics Collection

Plan costing is nearly impossible to perform effectively unless we have some basic table statistics describing the data that queries will run against. Therefore, we must first ensure that we have useful statistics before we even attempt to perform any plan costing.

Every table in NanoDB stores statistics describing that table's contents. These statistics are stored in the header page (page 0) of each table-file, immediately after the table's schema. These statistics are not updated continuously; that would be too costly. NanoDB currently requires the user to manually invoke the statistics-update operation. Additionally, the initial statistics created when a table is empty are, obviously, empty.

Users can update a table's statistics using the `ANALYZE` command. This command takes one or more table names as arguments, and performs stats collection on those tables:

```
CMD> ANALYZE cities, states, stores, employees;
Analyzing table CITIES
Analyzing table STATES
Analyzing table STORES
Analyzing table EMPLOYEES
Analysis complete.
CMD>
```

This analysis process scans each table file, collecting useful statistics from the table and then storing the results into the header page. When a table is opened in preparation to execute a query, the table's statistics are automatically loaded from the header page.

The statistics we will collect are:

- The total number of tuples in the table file
- The average size of a tuple, in bytes
- The total number of data pages in the table file (for NanoDB heap files, this is just the total number of pages minus one, since only the header page is not a data page)

In addition, for each column, we will collect:

- The total number of distinct values in that column, excluding `NULL` values
- The total number of `NULL` values
- The minimum and maximum value that appears in the column, but only if there are non-`NULL` values, and if the column's type is suitable for computing inequality-based selectivity estimates

Note that we only want to store the minimum and maximum values for a column if we can *easily* compute inequality-based selectivity estimates (such as `T.A > 5`) for the column. Generally we do this by computing the ratio (*high* – *low*) / (*maxVal* – *minVal*); depending on whether we are performing >, ≥, <, or ≤, we will use *maxVal* or *minVal* for the *high* or *low* value, and the actual comparison-value for the other value.

Although we could certainly find the max and min value for a `VARCHAR` column, it's substantially more difficult to perform the above computation with strings, so we will simply not gather min/max statistics for string columns. (In addition, the max or min string value may be very large, and it could use up substantial space in the header page, with no real benefit.)

You should also note that we will always find the total number of distinct values for a column, regardless of whether we will find the min and max values, because we need the number of distinct values to estimate the selectivity of = and ≠ conditions.

Completing the Stats-Collection Functionality

In NanoDB, all tuple file formats may provide statistics-collection capabilities through the `TupleFile.analyze()` interface-method. This is the method that NanoDB ultimately calls when the user issues the `ANALYZE` command. The (empty) implementation for this method is in the `HeapTupleFile` class; **this is where you will implement statistics-collection for heap files.**

In a database, you should always try to do as much work as possible with as little IO as possible. Specifically, when collecting statistics, your implementation should collect all stats in one pass over the table file. This is really not difficult to do; statistics collection can be implemented roughly like this:

```
for each data block in the tuple file:
  update stats based on the data block
  (e.g. use tuple_data_end – tuple_data_start to update a "total bytes in all tuples" value)

  for each tuple in the current data block:
    update stats based on the tuple (e.g. increment the tuple-count)
    for each column in the current tuple:
      update column-level stats
```

Note: While the `TupleFile` interface has `getFirstTuple()` and `getNextTuple()` methods, it will be faster if you access the file's `DBPages` and their contents more directly, without using these methods. Therefore, you should avoid using those functions in your implementation.

A handful of helper classes are provided to make statistics collection simpler (all of these classes are in the `edu.caltech.nanodb.queryeval` package):

- The `TableStats` class holds all the statistics for the tuple file. Among other things, it holds a list of `ColumnStats` objects, each one holding the statistics for the corresponding column.
- To help you compute the column statistics, there is a `ColumnStatsCollector` class that can be fed the values from a given column, and it will record the minimum and maximum values, the set of distinct values, and the number of `NULL` values. Then, it can be used to generate a `ColumnStats` object that holds the necessary details.

  In your implementation of `HeapTupleFile.analyze()`, you will want to create an array of `ColumnStatsCollector` objects, one for each column in the tuple file. As you traverse the tuples in the file, you can loop over each tuple's columns, passing each column's value into the corresponding `ColumnStatsCollector` object. When you have traversed all columns, you can generate a list of `ColumnStats` objects for the table statistics.

Don't forget that a tuple-slot can be empty if the corresponding tuple has been deleted! Make sure to use the `HeaderPage` and `DataPage` helper classes in implementing this class; they should make it very straightforward. Don't duplicate the same functionality already provided there, or you will lose points.

Finally, when you have completed generating the table statistics, you can save the new statistics as follows:

1) Create a new `TableStats` object with the new statistics you have computed
2) Store your `TableStats` object into the `HeapTupleFile.stats` field on the tuple-file object
3) Call `heapFileManager.saveMetadata()`, passing in the tuple-file object (`this`). The `heapFileManager` field is set to the `HeapTupleFileManager` that loaded the heap file that you are analyzing in your function.

The above function will update the header page with the serialized version of the current schema and statistics. Once you have done this, the stats will be saved to the tuple file.

Testing Statistics-Collection

The `ANALYZE` command doesn't print out the details when it is run on a table, so you can't really tell if it is working correctly by itself. However, NanoDB also has a "`SHOW TABLE` *t* `STATS`" command, which will output the statistics for the specified table. You can compare your results to the contents of the SQL files you loaded. Note that for some of the integer stats, -1 indicates "`NULL`" or "unknown", but for object-values, the stats will contain a `null` reference if the value is unknown.

Plan Costing and Selectivity Estimates

Plan costs in NanoDB are represented with a few simple measures. There are the standard ones you would expect, such as the number of tuples produced by each plan-node, the worst-case number of disk IOs that will be performed, and so forth. These values are managed in the `PlanCost` class (`edu.caltech.nanodb.queryeval` package).

There is also a "CPU cost" measure, which is a simple estimate of the computational cost of evaluating a plan, using some imaginary units. For example, we might say that the CPU cost of processing one tuple in a plan-node has a CPU cost of 1.0. A select plan-node might only produce 10 tuples, but if it has to look at 1000 tuples then the CPU cost generated by that node will be 1000.[1]

Also, unlike row-counts, these CPU costs should accumulate up the tree: if we have two equivalent plans, and our row-estimates are incredibly accurate so that we think the two plans will produce the same number of rows, the CPU cost will still tell us which plan is likely to do more work to generate the same results. For example, if a select plan-node must consider 1000 tuples, its cost will be 1000 *plus* the CPU-cost of its sub-plan.

Plan-Node Details

Every plan-node has a `prepare()` method that computes three critical pieces of information for the node:

- The schema of the plan node, stored in the protected field `PlanNode.schema`
- The estimated cost of executing the plan node, stored in the protected field `PlanNode.cost`
- Column-level statistics on the tuples produced by the plan node, stored in the protected field `PlanNode.stats`

Since these fields are protected access, all subclasses can access and manipulate these values. Subclasses are expected to provide an implementation of `prepare()` to compute these details.

You will need to complete this method for three of the plan-node types, computing the cost of each kind of node. (The schema and statistics are already computed for you.) The nodes whose implementation you must complete are:

- `SimpleFilterNode` – a select applied to a subplan
- `FileScanNode` – a select applied to a table file stored on disk
- `NestedLoopJoinNode` – a theta-join applied to two subplans; the join may be an inner or an outer join

The cost is represented as a `PlanCost` object (in the `edu.caltech.nanodb.queryeval` package). You should look at this class to see all values that your implementation must estimate. Use "best case" estimates; for now, you can assume that NanoDB always has all the memory it needs.

You can also look at examples of the `prepare()` method already implemented, in these classes: `RenameNode`, `SortNode`, `ProjectNode`. Be aware that these implementations are written to operate properly even when a child plan's cost is not available. You do not need you mimic this; you can assume that the child plans' costs are always available, because once you are done they always should be available.

Estimating Selectivity

Selectivity estimates are essential for guessing how many rows a plan-node will produce. Besides our table statistics, we must also be able to guess the *selectivity* of a predicate. A predicate's "selectivity" is simply the probability that a row from the input will satisfy the predicate; if we know the number of rows that come into a node, and we know the selectivity of a node's predicate, we simply multiply the two together to estimate the number of rows produced.

---

[1] We might want to be more intelligent and scale the CPU cost based on the size of the predicate, but we will keep it simple for now.

In NanoDB, the `SelectivityEstimator` class (`queryeval` package) is used for making all selectivity estimates. **You must complete the implementation of this class to compute the selectivity of predicates that appear in query plans.** Specifically, you must support these kinds of predicates:

- P1 AND P2 AND …
- P1 OR P2 OR …
- NOT P
- COLUMN = VALUE and COLUMN ≠ VALUE, for all column-types
- COLUMN ≥ VALUE and COLUMN < VALUE, for column-types that support the ratio-computation discussed earlier
- COLUMN ≤ VALUE and COLUMN > VALUE, for column-types that support the ratio-computation discussed earlier
- COLUMN_A = COLUMN_B and COLUMN_A ≠ COLUMN_B, for all column-types

If a predicate doesn't fall into one of these categories, use a default selectivity estimate of 25%.

Some important caveats about the last four kinds of predicates:

- Be aware that the necessary statistics are <u>not</u> always available! For example, if someone hasn't run `ANALYZE` on a table in a query. In those cases, use the default selectivity estimate.
- These conditions are grouped in pairs for a reason – you should implement the first estimate, and then implement the second one in terms of the first one. For example, we can assume that P(A > 5) = 1.0 – P(A ≤ 5), or that P(A ≠ B) = 1.0 – P(A = B).
- NanoDB includes a `normalize()` method on comparison expressions, such that when a column and a value are being compared, the column will always appear on the left. This means we don't have to support VALUE *op* COLUMN, only COLUMN *op* VALUE.
- Finally, recall that we will only estimate the selectivity of >/≥/</≤ when the column-type easily supports it; we will not support this estimate for strings, for example.

You will see many detailed comments in the `SelectivityEstimator` class; you really only have to focus on the costing equations, and how to extract the statistics you will need. (Read the Javadocs for the various classes you will need to use.) Many other parts are provided for you, including the `computeRatio()` method that is able to take a column's min and max values, and compute the ratio (*high* – *low*) / (*maxVal* – *minVal*) mentioned earlier. This function can work with any numeric type.

Testing Plan Costing

After you have completed your plan-costing code, you will want to ensure that it works properly. You don't have to write any test classes this week (but if you do, there will be extra credit for it!). A simple schema is provided for you on the course website; you can load `make-stores.sql`, and then load `stores-28K.sql`.[2] Obviously, make sure to `ANALYZE` these tables before trying any queries, or else you won't have statistics for the costing computations to use.

<u>Note</u>: You will need to put the answers for this section into your design document, so you might as well record them.

As mentioned last week, NanoDB has an `EXPLAIN` command you can use to see what it generates for different query plans. For example, you can try this:

```
EXPLAIN SELECT * FROM cities;
```

[2] Note that the `states` table contains 51 rows since it also includes the District of Columbia.

Once you have completed the costing implementation, you should see something like this (your costs might be different, depending on your assumptions):

```
Explain Plan:
    FileScan[table:  CITIES] cost=[tuples=254.0, tupSize=23.8, cpuCost=254.0, blockIOs=1]

Estimated 254.0 tuples with average size 23.787401
Estimated number of block IOs:  1
```

If you then try: `EXPLAIN SELECT * FROM cities WHERE population > 5000;`

This should print out the same plan costs, since the smallest city-population in that table is 100135.

You can see the costs change if you try queries like this:

```
EXPLAIN SELECT * FROM cities WHERE population > 1000000;
EXPLAIN SELECT * FROM cities WHERE population > 5000000;
```

Take note of how many rows the database expects this last query will produce. (My estimate is 99.3 tuples.) *However, how many tuples does the query actually produce?*

Now, if you run the following query, notice that you will get the exact same results, but if you `EXPLAIN` it, the costing estimate is *very* different:

```
SELECT * FROM cities WHERE population > 8000000;
```

This is the fundamental limitation of the simple statistics that we track in NanoDB. Clearly, recording a histogram for different columns would produce much more accurate estimates.

Costing Joins

You can also try your costing estimates on more complex queries involving join operations. For example:

```
EXPLAIN SELECT store_id FROM stores, cities
WHERE stores.city_id = cities.city_id AND cities.population > 1000000;
```

(My analyzer predicts 1776.2 tuples for this query, and a CPU cost of 1,019,776.3 units. Don't feel like you have to match these numbers exactly; they are just my analyzer's estimates.)

Of course, the planner isn't quite intelligent enough to position the query predicates in optimal locations in the query plan, although we can manually rewrite the query to put predicates in better positions:

```
EXPLAIN SELECT store_id
FROM stores JOIN
     (SELECT city_id FROM cities
      WHERE population > 1000000) AS big_cities
     ON stores.city_id = big_cities.city_id;
```

(The rewritten query is still estimated to produce 1776.2 tuples, but the CPU cost is now 962,940.8 units. Woo, 56000 "CPU units" less!)

Finally, here is a slower query for you to tinker with:

```
SELECT store_id, property_costs
FROM stores, cities, states
WHERE stores.city_id = cities.city_id AND cities.state_id = states.state_id AND
      state_name = 'Oregon' AND property_costs > 500000;
```

The estimated tuple-count on this one is ~23 tuples, and the solution estimates a CPU cost of 45,214,024.0. The actual tuple-count is 7.

See how fast you can get it by rewriting it! (I was able to take it from 35 seconds down to 0.5 seconds on my laptop, with a final CPU cost of 299,724.4.)

Submitting Your Assignment

When you are finished with the assignment, tag it with a `hw3` tag as usual, and then push all of your changes to your team repository. (Remember, if you need to submit an updated version later, update the tag to `hw3-2`, etc. Don't try to reuse the same tag multiple times; it will only mess everything up.)

Finally, complete the design document, including the Git commit-hash of the checkin you want us to grade, and submit this file on Moodle.

Extra Credit

This week's assignment is a bit easier than the previous two assignments. So, if you want to earn some extra credit, here are some options for you to pursue.

- The assignment doesn't require you to modify the statistics describing a plan-node's output, but this is a straightforward thing to add, as long as you constrain the kinds of situations you support. You should modify the selection plan-nodes to produce statistics based on the selection predicate. Try supporting predicates of the form: P1 AND P2 AND …, where P1, P2, etc. are of these forms:

    - COLUMN *op* VALUE
    - COLUMN IN (VALUE-LIST) (there is a specific operator for this kind of comparison)
    - COLUMN = VALUE1 OR COLUMN = VALUE2 OR … (where all column-references are the same column name; in other words, the column is b)

    This should sufficiently constrain the problem that it will be feasible to implement. Don't feel compelled to support other forms of predicates, unless you see something very easy to support.

    You will receive maximal points for supporting all of the above forms with a clean architecture, and with corresponding unit tests. Partial support, messy implementation details, or lack of testing will reduce the bonus correspondingly.

- You may have noticed that there is no testing for the table-analysis and plan-costing code in Assignment 3. Implement some unit-tests to exercise and verify these components. The better and more useful tests you add, the more extra credit points you will get. (If they are very good, we will likely incorporate them into the course codebase.)

- Similarly, the Assignment 2 testing is also limited. For example, there are few queries exercising various kinds of joins, nested queries in the `FROM` clause, queries requiring renaming of various tables, subqueries involving grouping and aggregation, and so forth. Add more tests! The more useful tests you add, the more extra credit points you will get.