

Project 03: Table Statistics and Plan Costing

Project Overview

In this project, you will lay the foundation for join optimization by implementing statistics collection, plan-costing, and selectivity estimation.

Before Starting

As with the previous project, we've distributed some updates and additional files to your repository. If you don't have the file `doc/03-statistics-design.md`, make sure that your local changes are committed, then `git pull --rebase` (or `merge`).

Plan Costing

As discussed in class, plan costing is rather imprecise. We want to be able to estimate how long it will take to evaluate a plan, given statistics that may or may not be up to date, without actually executing the query. While there are many ways to represent and compute plan costs and table statistics, NanoDB takes a fairly basic (yet effective) approach.

“Stores” Database

To verify statistics and plan-costing, we would like to have a larger dataset that includes multiple tables. To accomplish this, we use the schema for “stores”, in `schema/stores`. The `make-stores.sql` file specifies the schema; while the other two files contain two different datasets. **Throughout this guide, we will assume the `stores-28K.sql` file.**

You can use the `\source` command to load these files in the NanoDB prompt:

```
CMD> \source schemas/stores/make-stores.sql
Table employees already doesn't exist; skipping drop-table.
Table stores already doesn't exist; skipping drop-table.
Table cities already doesn't exist; skipping drop-table.
Table states already doesn't exist; skipping drop-table.
Created table: states
Created table: cities
Created table: stores
Created table: employees
CMD> \source schemas/stores/stores-28K.sql
```

The second operation may take a while because of key-constraint enforcement; if it is too slow for you, you can turn off key-constraint enforcement before loading with this command:

```
CMD> set property 'nanodb.enforceKeyConstraints' = false;
```

Statistics Collection

Plan costing is nearly impossible to perform unless we have some basic table statistics describing the data that queries will run against. Every tuple-file in NanoDB stores statistics describing the file's contents. These statistics are stored in the header page of each tuple file, immediately after the table's schema. These are not updated continuously, as that would be too costly.

NanoDB currently requires the user to manually invoke the statistics-update operation with 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.
```

The analysis scans each table, and updates the statistics stored in the header page. We will collect:

- 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

Additionally, 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 values that appear in the column
 - Only if there are non-NULL values, and only if the column's type is suitable for inequality-based estimates (i.e. the ratio makes sense to compute)

Task 1: Statistics Collection

In NanoDB, all tuple file formats implement the `analyze()` method, which is invoked through the ANALYZE command. Complete the implementation of the `HeapTupleFile#analyze()` method.

Your implementation *must* collect all stats in one pass over the table file, implemented roughly like so:

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

To track column-level stats, you will want to have an array of [ColumnStatsCollectors](#) (one for each column in the tuple file), each of which can later be converted into a [ColumnStats](#).

The stats will ultimately be saved as a [TableStats](#) under the [HeapTupleFile#stats](#) field. To serialize the stats to disk, call `heapFileManager.saveMetadata(this)`.

While the `TupleFile` interface has `getFirstTuple()` and `getNextTuple()` methods, it will be faster to access `DBPages` and their contents directly. (We're inside the class and know that we're iterating over all tuples in sequence.) Remember that tuple slots may be empty!

Testing Statistics-Collection

The ANALYZE command doesn't print out the details, but NanoDB also has a "SHOW TABLE t STATS" command. Note that for some of the integer stats, -1 indicates NULL or "unknown", but for object values (like VARCHARs), the stats column will contain a null if the value is unknown.

The reference solution outputs the following statistics:

```
CMD> SHOW TABLE states STATS;
Statistics for table states:
  51 tuples, 1 data pages, avg tuple size is 15.7 bytes
  Column state_id: 51 unique values, 0 null values, min = 1, max = 51
  Column state_name: 51 unique values, 0 null values, min = null, max = null
CMD> SHOW TABLE cities STATS;
Statistics for table cities:
  254 tuples, 1 data pages, avg tuple size is 23.8 bytes
  Column city_id: 254 unique values, 0 null values, min = 1, max = 254
  Column city_name: 244 unique values, 0 null values, min = null, max = null
  Column population: 254 unique values, 0 null values, min = 100135, max = 8143197
  Column state_id: 44 unique values, 0 null values, min = 1, max = 44
CMD> SHOW TABLE stores STATS;
Statistics for table stores:
  2000 tuples, 4 data pages, avg tuple size is 13.0 bytes
  Column store_id: 2000 unique values, 0 null values, min = 1, max = 2000
  Column city_id: 254 unique values, 0 null values, min = 1, max = 254
  Column property_costs: 882 unique values, 0 null values, min = 0, max = 999000
CMD> SHOW TABLE employees STATS;
Statistics for table employees:
  24932 tuples, 118 data pages, avg tuple size is 36.4 bytes
  Column emp_id: 24932 unique values, 0 null values, min = 1, max = 24932
  Column last_name: 4958 unique values, 0 null values, min = null, max = null
  Column first_name: 4933 unique values, 0 null values, min = null, max = null
  Column home_loc_id: 254 unique values, 0 null values, min = 1, max = 254
  Column work_loc_id: 254 unique values, 0 null values, min = 1, max = 254
  Column salary: 46 unique values, 0 null values, min = 35000, max = 80000
  Column manager_id: 11754 unique values, 4259 null values, min = 1, max = 24924
```

Estimating Selectivity

As discussed in class, selectivity estimates allow us to guess how many rows a plan node will produce. If we know the number of rows that are input to a node, and we know the predicate's selectivity, we can multiply to estimate the number of rows produced. In NanoDB, the [SelectivityEstimator](#) class is used for making all selectivity estimates.

Task 2: SelectivityEstimator

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:

- $P_1 \wedge P_2 \wedge \dots$
- $P_1 \vee P_2 \vee \dots$
- $\neg P$
- $\text{COLUMN} = \text{VALUE}$ and $\text{COLUMN} \neq \text{VALUE}$, for all column types
- $\text{COLUMN} \geq \text{VALUE}$ and $\text{COLUMN} < \text{VALUE}$, for all column types that support the ratio computation
- $\text{COLUMN} \leq \text{VALUE}$ and $\text{COLUMN} > \text{VALUE}$, for all column types that support the ratio computation
- $\text{COLUMN}_A = \text{COLUMN}_B$ and $\text{COLUMN}_A \neq \text{COLUMN}_B$, for all column types

If a predicate doesn't fall into one of these categories, use the `DEFAULT_SELECTIVITY` constant (0.25) as your estimate.

Some notes:

- The necessary [ColumnStats](#) may not be available! For example, the `ANALYZE` command may not have been run. In these cases, use the default estimate.
- The conditions are grouped in pairs because they are negations; and the starter code structure reinforces this. Don't duplicate code unnecessarily.
- Why did we not consider $\text{VALUE} = \text{COLUMN}$? [CompareOperator](#) has a `normalize()` method, which ensures that the column is always on the left. This is already called for you.
- The selectivity of $>$, \geq , $<$, \leq is only supported for column types for which the ratio computation makes sense; in particular, `VARCHARs` are not supported. (See [computeRatio\(\)](#) and [typeSupportsCompareEstimates\(\)](#)).

Estimating Plan Node Column Statistics

If a predicate is applied within a plan node, the column statistics of the node's outputs may also be affected. NanoDB has a class for updating these called [StatisticsUpdater](#). A `PlanNode` can request updated stats for its predicate by calling the `updateStats` function, which handles only two kinds of predicates:

- COLUMN op VALUE , where `op` is one of $\{=, \neq, >, \geq, <, \leq\}$
- $P_1 \wedge P_2 \wedge \dots$ ([conjunctive selective predicate](#))
 - Each P_N is in the COLUMN op VALUE form.

The actual statistic update happens in the `updateCompareStats()` method, which works on each individual P_n , and updates the [ColumnStats](#) object for the relevant column.

Task 3: Applying Predicates to Statistics

Implement the `updateCompareStats()` method. **When it makes sense to do so**, update the values in the provided `stat` object based on the predicate – it may not always be possible to make a reasonable update or estimate.

Plan Costing and Selectivity Estimates

NanoDB represents plan costs with the `PlanCost` class, which contains some of the statistics discussed in class (number of tuples, worst-case disk IOs, etc.) Of note is the `cpuCost` measure, which estimates the computational cost of evaluating a plan, using some imaginary units. For example, let's say that the CPU cost of processing one tuple in a plan-node is 1.0. Although a select plan-node may produce 10 tuples, if it has to look at 1000 tuples, then the CPU cost generated is 1000.0. Unlike row counts, the CPU costs should accumulate up the tree. For example, the above select-plan node's total cost is 1000 plus the CPU cost of its subplan.

Plan Nodes

Every plan node has a `prepare()` method that computes three pieces of information, stored in protected fields of `PlanNode`:

- `schema` (output schema of the node)
- `cost` (estimated cost of executing the node)
- `stats` (column-level statistics on the output tuples)

The schema is already computed for you.

Task 4: Plan Node Costing

Update the `prepare()` method to compute the cost and stats in the following three plan nodes:

- `SimpleFilterNode` (select applied to a subplan)
- `FileScanNode` (select applied to a table file on disk)
- `NestedLoopJoinNode` (theta-join applied to two subplans)

Some notes:

- Use the `SelectivityEstimator` to estimate the selectivity of a predicate where relevant.
- Use "best case" estimates; for now, assume that NanoDB always has all the memory it needs.
- Your join I/O calculation should be true to your implementation.
- You can refer to existing implementations in `RenameNode`, `SortNode`, `ProjectNode`. Note that these work properly even when a child plan has no cost. You don't need to account for this, because you're adding the only missing ones (so the cost will always be available).
- In class, we discussed how join-costing can account for primary or candidate keys. For your initial implementation, use the non-key-based approach.
- For a join node, the provided call to `prepareSchemaStats()` ensures that stats are passed through. Don't worry about any updates beyond that.

Manually Testing Plan Costing

Make sure to ANALYZE your tables before running these commands.

Basic Plans

As mentioned in Project 2, NanoDB has an EXPLAIN command to inspect what it generates for query plans. Once you have completed this week's project, you should see something like this:

```
CMD> EXPLAIN SELECT * FROM cities;
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 we instead tested the query

```
EXPLAIN SELECT * FROM cities WHERE population > 5000;
```

we should see the **same plan costs**, since the smallest city population in the table is 100,135.

You can see the costs change if we increase the threshold:

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

Note how many rows the database expects the last query to produce (our estimate is 99.3 tuples) – but how many does it actually produce? Compare it to the following query:

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

which will *produce* the exact same results, but has a very different costing estimate. This is a fundamental limitation of the simple statistics we track in NanoDB. As mentioned in class, a histogram is one way of producing more accurate estimates.

Joins

You can also test 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;
```

Our 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're just our particular implementation's estimates. (Alternate solution CPU cost: 1,527,776.3 units.)

For now, our planner can't reorganize the query to put the predicates in optimal locations in the query plan, though we can do so manually:

```
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, for an incredible savings of 56000 (~5%) “CPU units”! (Alternate solution CPU cost: 1,414,105.3 units, a savings of 113,671 or ~7.4%.)

Finally, here’s a slower query for you to experiment 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 around 23 tuples, and our reference implementation estimates a CPU cost of 45,214,024.0. The actual tuple-count is 7.

It will be very slow if you try to run it. See how fast you can get it by rewriting it!

Column Statistics

The EXPLAIN command doesn’t show the column-stats for each plan node because this would make the command output far too dense. You can still do some basic testing by creating nested subqueries in the FROM clause:

This plan will report a specific cost and estimated number of tuples. The city_id column is a primary key with uniform distribution, so the estimate should be pretty accurate.

```
EXPLAIN SELECT * FROM cities WHERE city_id > 20 AND city_id < 200;
```

(The estimates for this query may be slightly different from the below queries, for boring, nitty reasons.)

You can force your planner to create multiple select nodes with subqueries:

```
EXPLAIN SELECT * FROM (
  SELECT * FROM cities WHERE city_id > 20) t
WHERE city_id < 200;
```

Then, you can reverse the application of the predicates and see if the results are still the same:

```
EXPLAIN SELECT * FROM (
  SELECT * FROM cities WHERE city_id < 200) t
WHERE city_id > 20;
```

If the plan-nodes are properly updating their output statistics, you should see extremely close or even identical costing estimates for these kinds of queries.

Extra Credit

Candidate Keys

In class, we discussed how joins can use primary (or candidate) key information to produce better estimates. However, while NanoDB schemas can report what candidate keys hold ([Schema#candidateKeys\(\)](#)), none of the plan nodes propagate this information to output schemas.

Extra Credit: Candidate Key Propagation and Costing

Update the following plan nodes to report candidate keys on their output:

- SimpleFilterNode
- FileScanNode
- SortNode
- RenameNode
- ProjectNode
- GroupAggregateNode
- ThetaJoinNode

Then, update the NestedLoopJoinNode to use candidate key information in its costing estimate. Additionally, the FileScanNode and SimpleFilterNode estimates can be refined as well.

Column Statistics

NanoDB's column statistic updates are somewhat limited. We can provide estimates in other scenarios.

Extra Credit: More Column Statistic Predicates

Add support for these disjunctive selection predicates:

- `COLUMN IN (VALUE-LIST)` (see the [InValuesOperator](#) class). The COLUMN should not appear in other conjuncts.
- `COLUMN = V1 ∨ COLUMN = V2 ∨ ...`
 - Each of the column references is the same column name; all comparisons are `=`, and the column does not appear in other parts of the predicate.
 - Additionally, support the case where one of the conjuncts of a conjunctive selection is a nested expression of this form.

Design Document

Design Doc

Answer the questions in the file `doc/03-statistics-design.md` in the document itself.

Submitting

Once you are finished, make sure your code is pushed to GitLab. Then, submit the commit hash you would like to be reviewed on Gradescope.

For more details, see the Project Submission instructions on the course website.