Assignment Four, Part One: Statistical Estimation

Due April 6, by 11:59PM

Assignment four will be broken into two parts. In the first part, you are asked to implement from scratch a class called `Statistics` that will be used by your query optimizer to help it decide among the various query plans. In the second part of A4, you will then use the `Statistics` class to actually implement your optimizer.

The `Statistics` class will implement what essentially amounts to two different (but related) types of functionality. First, this class will store statistical information about your database's attributes and relations. In this sense, it is a container class. It will be able to serialize itself to a text file on disk, and re-read this information from disk at a later time.

Second, the `Statistics` class also has the ability to use this statistical information during the optimization process to "simulate" joins and selection operations over the relations it describes, in order to guess what the statistics would be after the application of such relational operations to the actual data stored in the database.

Aside from the requirement that you implement exactly the ten methods that are described in this assignment, you have complete freedom to implement all of this however you see fit. My only real suggestion is that all of the attribute and table information should be stored internally within some sort of hash structure or structures so that you can very quickly locate any attributes and/or relations as you are trying to perform lookups of the statistics that are associated with each. The reason for this that you will be doing a lot of lookups on attributes and relations, and you will make extensive use of a statistics object within the inner-most loop of the query optimizer. If your data structures are slow and you are always doing sequential search using string comparisons, there may a significant performance hit.

Container Operations

The following are the container-oriented operations having to do with storing simple data about the relations and attributes that the Statistics object will operate over. We begin with `AddRel`:

```
void AddRel (char *relName, int numTuples);
```

This operation adds another base relation into the structure. The parameter set tells the statistics object what the name and size of the new relation is (size is given in terms of the number of tuples). Next is `AddAtt`:

```
void AddAtt (char* relName, char *attName, int
             numDistincts);
```

This operation adds an attribute to one of the base relations in the structure. The parameter set tells the `Statistics` object what the name of the attribute is, what relation the attribute is attached to, and the number of distinct values that the relation has for that particular attribute. If `numDistincts` is initially passed in as a `–1`, then the number of distincts is assumed to be equal to the number of tuples in the associated relation.

Note that `AddRel` and `AddAtt` can both be called more than one time for the same relation or attribute. If this happens, then you simply update the number of tuples or number of distinct values for the specified attribute or relation.

Next we have `CopyRel`:

```
void CopyRel (char *oldName, char *newName);
```

This operation produces a copy of the relation (including all of its attributes and all of its statistics) and stores it under the new name.

The `Statistics` object also has the ability to write itself to a text file, and then also to read itself back from a text file. In the case where the object is asked to read itself from a file that does not exist, it should not give an error; instead, the resulting text file is simply empty. The operations that perform this reading and writing are:

```
void Read (char *fromWhere); void Write (char *fromWhere);
```

We also have a constructor, a copy constructor, and a destructor. Note that the copy constructor must perform a deep copy of all of the data structures that live internally within the `Statistics` object that is to be copied:

```
Statistics (Statistics &copyMe); Statistics ();

~Statistics ();
```

What-If Operations

Now, we get to the interesting operations. First, we have `Apply ()`:

```
void Apply(struct AndList *parseTree, char **relNames, int
numToJoin);
```

This operation takes a bit of explanation. Internally within the `Statistics` object, the various relations are partitioned into a set of subsets or partitions, where each and every relation is contained within exactly one subset (initially, each relation is in its very own singleton subset). When two or more relations are within the same subset, it means that they have been “joined” and they do not exist independently anymore. The `Apply` operation uses the statistics stored by the `Statistics` class to simulate a join of all of the relations listed in the `relNames` parameter. This join is performed using the predicates listed in the parameter `parseTree`.

Of course, the operation does not *actually* perform a join (actually performing a join will be the job of the various relational operations), but what it does is to figure out what might happen if all of the relations listed in `relNames` *were* joined, in terms of what it would do to the important statistics associated with the result of the join. To figure this out, the `Statistics` object estimates the number of tuples that would exist in the resulting relation, as well as the number of distinct values for each attribute in the resulting relation. How exactly it performs this estimation will be a topic of significant discussion in class. After this estimation is performed, all of the relations in `relNames` then become part of the same partition (or resulting joined relation) and no longer exist on their own.

Note that there are a few constraints on the parameters that are input to this function. For completeness, you should probably check for violations of these constraints, because when you write your optimizer using the `Statistics` class, it will be very useful to have good error checking.

First, `parseTree` can only list attributes that actually belong to the relations named in `relNames`. If any other attributes are listed, then you should probably catch this, print out an error message, and exit the program. Second, the relations in `relNames` must contain exactly the set of relations in one or more of the current partitions in the Statistics object. In other words, the join specified by the set of relations in `relNames` must make sense. For example, imagine that there are five relations: A, B, C, D, and E, and the three current subsets maintained by the `Statistics` objects are {A, B}, {C, D}, and {E} (meaning that A and B have been joined, and C and D have been joined, and E is still by itself). In this case, it makes no sense if `relNames` contains {A, B, C}, because this set contains a subset of one of the existing joins. However, `relNames` could contain {A, B, C, D}, or it could contain {A, B, E}, or it could contain {C, D, E}, or it could contain {A, B}, or any similar mixture of the current partitions. These are all valid, because they contain exactly those relations in one or more of the current partitions. Note that if it just contained {A, B}, then effectively we are simulating a selection.

Also note that if `parseTree` is empty (that is, null), then it is assumed that there is no selection predicate; this either has no effect on the `Statistics` object (in the case where `relNames` gives exactly those relations in an existing partition) or else it specifies a pure cross product in the case that `relNames` combines two or more partitions.

Finally, note that you will never be asked to write or to read from disk a `Statistics` object for which `Apply` has been called. That is, you will always write or read an object having only singleton relations.

The final operation that you'll need to implement is `Estimate`:

```
int Estimate(struct AndList *parseTree, char **relNames,
             int numToJoin);
```

This operation is exactly like `Apply`, except that it does not actually change the state of the `Statistics` object. Instead, it computes the number of tuples that would result from a join over the relations in `relNames`, and returns this to the caller.

That's it!

What To Turn In:

1. Turn in your submission via Canvas. Turn in your code, output41.txt, and your report in a zip file called *firstNameLastName1_firstNameLastName2_p41.zip*.

- Do not include .tbl or .bin files.
- Include code named a2test.cc needed to create .bin and supporting files from the .tbl files.
- Make sure your code runs with all files in directory a4-1test without any modifications.
- Don't change the directories in test.cat. Since part of the grading may be done by script, it's import to keep the file structure unchanged.
- Store your statistics in a file named Statistics.txt

Include the following:

a) All code needed to compile test.cc and a2test.cc
b) Create 2 GTests for any of the methods you wrote. Turn in your GTest code.
c) In a Bash shell, using the .bin files generated from tpch-dbgen with the option -s 1 (1GB files) run the test cases script with the following command:

```
./runTestCases.sh
```

Submit the file output41.txt generated after running the test cases script. If you wrote your code in Rust or are running on a Windows machine, change the script as needed.

2. Your report as a PDF file that includes:
   a) Group member names

   b) Include instructions on how to compile and run your code as well as a brief explanation of each method you wrote and how it works.

   c) Explain the format you used for Statistics.txt.

   d) A screen shot of output41.txt generated after running the script.

   e) Screen shots of GTest results that match results generated by your code.

   f) If you had a problem with the code or found a bug that you think would be useful for future classes to know about, list the bug you found and what you did to fix it.

Grading:

Graded out of 100 points.
+20 Gtests

+50 points for correct output for each query

+30 report