COMP 430 Assignment 4.2

Overview

In this assignment, your task is to finish up your single-user database system that you started as A4.1. I've talked about this in class a bit, but basically, when your program is given an SQL query, you will compile and type check it using the A4.1 compiler, translate the output of the A4.1 compiler into relational algebra, optimize the relational algebra, and then execute the relational algebra, print the results, and then clean up any garbage files that you have created.

I have supplied you with code to actually execute individual relational algebra operations, so you don't need to implement them. Your job is to implement an optimizer, as well as some software that uses the individual operations I'll supply you with to run an entire query plan.

Supplied Code

In addition to the A4.1 code, in A4.2.jar I have given you three new Java classes that implement the relational algebra operations you need: `Selection`, `Join`, and `Grouping`, as well as a few helper classes, and an extensive example of how to actually use these classes (in `Runner.java`). You'll notice that, strictly speaking, these three operations are not pure relational algebra because they all have the ability to do things like run complicated projections. But it will be clear how to use them after you look over the code a bit.

One thing that is important for you to understand (and was discussed in class) is that the way that these Java classes work is to "write" C++ code, which is then compiled and executed to actually perform the operation. This means that to actually be able to run everything, you need a C++ compiler installed on your machine. `Runner.java` and these three classes have been tested on Linux and Mac using the `g++` compiler (note: `g++` and the other standard developer tools are not shipped with your Mac; you'll need to download and install all of them from Apple).

Unfortunately, you'll expectedly have to change things slightly to get everything to work on Windows or with a different C++ compiler. For example, look at lines 94 and 95 in `Selection.java`, where a command is built up and then used to execute the C++ compiler, via the java `Process.exec` method. If you use Windows and/or you use a different C++ compiler, you'll likely need to change these two lines so they work with your environment and your compiler, because your compiler may take slightly different arguments. You'll also need to change the corresponding lines in `Grouping.java` and `Join.java`.

If you don't want to deal with all of the that, you might want to seriously consider using Linux rather than Windows. There are good tools out there like `cygwin` that will allow you to run something a lot like Linux from within Windows; `cygwin` in particular has the `g++` compiler bundled as part of the distribution (see `http://en.wikipedia.org/wiki/Cygwin`).

Creating the TPC-H Data and Running Runner.java

For the remainder of A4, we'll be using the TPC-H benchmark database. You'll need an instance of this database to actually try out `Runner.java`. To create this database (assuming that you have a C compiler installed, such as `gcc`) you can use the "`dbgen`" program, whose source code I have provided for you as part of `A4.2.zip` (as an alternative, you can go to the TPC-H website and download the very latest version). Go into the `tpch-dbgen` subdirectory, and then (assuming Mac or Linux) type "`make`". A bunch of stuff will happen, with the important result being the creation of the `dbgen` executable. By running the executable, you can create the eight TPC-H tables. If you just run this executable with no args, you'll create the default "scale factor one" database, which is about one GB in size (you can easily create larger or smaller ones for testing if you'd like, by giving `dbgen` appropriate arguments).

If you want to use this data with `Runner.java`, copy the resulting `.tbl` files (which are in plain text, with one record per line) one directory up, so they are sitting with the Java files. Then compile and run `Runner.java`. `Runner.java` will run a selection `over orders`, then a join over `customer` and `orders`, and then a grouping over orders. The result of each operation is placed in to a text file with the same format as the various `.tbl` files, so (in theory) you could run more relational algebra operations over the output of those first operations, which is exactly what you'll be doing eventually. Also note that all of the temporary little C++ files that `Runner.java` creates as it "writes" its C++ code will be put into the `cppDir` directory. Check them out to get a better idea of what is going on.

Your Optimizer

As described above, your task is to optimize and run the output of the parser, where the "running" is accompanied using the operations I've given you.

The first thing you'll need to do is to design some data structures that can hold a relational algebra expression. Then you'll write code to take the output of the parser and create a simple relational algebra expression with a single selection predicate sitting on top of a bunch of joins (cross products, actually) over of all of the input tables below the selection. You should just ignore projections and grouping/aggregation at this point.

Once you've got that, you might want to work on just being able to run a simple relational algebra expression (using `Selection.java` and `Grouping.java`) that has no joins, with no optimization. This will get you at least a low passing grade on A4.2, since several of the queries I'll give you don't even have any joins.

One you've actually been able to run something and you have a few points guaranteed on the assignment, you should work on optimizing the relational algebra. In particular, you should implement the following three relational algebra transformations on your data structure:

1) A transformation that pushes a selection predicate down past a join (breaking it up if needed).

2) A transformation that takes three joins of the form `(A join (B join C))` and transforms them into `(C join (A join B))`, and which handles all of the attached selection predicates appropriately.

3) A transformation that takes `(A join (B join C))` and transforms it into `(B join (A join C)),` and which also handles all of the attached selection predicates appropriately.

Once you've got that, you should write some code that takes a relational algebra expression and "costs" it, by estimating the number of tuples created by each relational algebra operation. You can then use this along with your transformations to implement a simple (yet effective) greedy query optimizer. Take the current plan, and try each of the three transformations described above on all of the spots in the plan where they are applicable. Cost the result of each transformation. Then choose the best resulting plan, and replace the current plan. Repeat as long as you need to until you get no change. At that point, you have your optimized relational algebra expression.

Now you've got to run it. So you need to design and implement some code that uses the three operations I've given you to run the query plan. As you execute the plan, you should just project away attributes as soon as you are sure you won't need them anymore, and run the grouping and aggregation as the last operations at the top of the plan (if needed).

Thats it!

Well, not really. There are some other things you can do to get fancy. For example, rather than maintaining just the best plan, you can maintain the top k best plans; this might get you an overall better result. You can try to implement early evaluation of output expressions, so that (for example) if the user is asking for `(table1.a + table1.b)` as the output, you evaluate this early on and then project away the two input attributes, which means that you have to carry around less data. You can implement grouping on multiple attributes, which `Grouping.java` supports but our parser does not.

Testing and Turnin

In the end, when the user issues a query, your program should evaluate it, print (at most) the first 30 output records to the screen, and then print the time that was taken for evaluation. When the user exits the program, the total time for evaluation of all of the queries that were run in the session should be printed to the screen.

I will supply you (soon!) with a suite of ten test queries. When you are ready to prepare your turnin doc, you should fire up your program and run those queries (in order) in single session. Skip any ones that you can't run. To get 60% of the credit on A4.2, you need to be able to run three of those correctly. That should be relatively easy, since three of the queries don't even have a join and can be run without even having a functioning optimizer. After that, running each of the remaining six queries gets you an additional 5%, for a total of 95%.

Once you get all ten of the queries running, the last 5% of your grade comes from the total running time needed for all queries in the session. This is why your program should print out the total running time for all queries furring the session. The person with the fastest running time gets 105%. After that, your score goes down (95% is the min score you can get) depending on how much slower you were than the fastest. This may seem a bit unfair (that you can get everything working but not get 100%) but remember that the purpose of this assignment is optimization, and the one way that I have to grade the quality of your optimizer is how fast the queries run.

To make things as fair as possible when comparing execution times, before you run the suite of queries, you should first run `Runner.java`, unmodified, on the same hardware you'll be using to test the ten queries. The output will say how long the three operations run by `Runner.java` took. Record the output at the top of your turnin document. When computing how fast your final code is, I'll divide the running time by the number of milliseconds taken by `Runner.java`. So that way if you have a slow machine, you are not penalized.

Print out your results as a hard copy and give it to me by the due date (same as A4.1). Also, submit a soft copy of your turnin document, as well as all of your source code.