COMP 530 Assignment 7: Putting It All Together

1. The Task

Your task, very simply, is to put together all of the pieces we've built this semester together into a single, final product. Basically, when your program is given an SQL query, you will compile it using the A6 compiler, translate the output of the A6 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.

Note that in `A7.zip`, I've included my solution to A5, as well as the A6 skeleton code augmented with some additional features. For example this A6 code maintains a `MyDB_TableReaderWriter` and `MyDB_Table` object for each of the tables currently defined in the system (it also maintains a `MyDB_BPlusTreeReaderWriter` object for each B+-Tree file). It also has some functionality for loading those tables from text (you can try this; from MyDB the command line, create the supplier table, and then type "`LOAD supplier FROM supplier.tbl`"). Also note that it is possible to create a table using a B+-Tree, by simply adding "`AS BPLUSTREE ON some_attribute`" at the end of the `CREATE TABLE` statement. Note that I did not include my own type checking code in `A7.zip`, because all of the queries in the test suite are semantically correct.

2 Creating the TPC-H Data

For the remainder of A7, we'll be using the TPC-H benchmark database. You can either create this database yourself (see `http://www.tpc.org/tpch/`) or you can download a version that I've posted onto Dropbox, at:

`https://www.dropbox.com/sh/o5qytjw7qncglem/gtNx3EklIT`

Note that we are using the default "scale factor one" database, which is about one GB in size.

3 The Path to Finishing the Assignment

The first thing you'll need to do is to look over the code that I have provided. I have provided some data structures that can hold a relational algebra expression (or equivalently a logical query plan). These data structures are in the file `MyDB_LogicalOps.h` in the new `Execution` source code directory. At the highest level, your task is twofold.

First you need to be able to take a `SFWQuery` data structure and translate it into a tree of `LogicalOp` objects (note that you'll be modifying the code for both the `SFWQuery` class and the LogicalOp class; you are welcome to modify any code I've given in any way that you see fit). I have already provided some code that can take a `SFWQuery`

data structure that encodes a query of exactly two tables (with no aggregation) and can use it to generate a small `LogicalOp` tree.

Your task will be to extend that code so that it can accept any `SFWQuery` data structure and translate it into a tree of `LogicalOp` objects that can run efficiently. One thing that you'll need to consider is the problem of generating a high-quality `LogicalOp` tree. Some of the more complicated queries that you'll be asked to run have a large number of tables, and can produce a lot of intermediate data if you choose a bad plan. One easy heuristic is to simply produce what is known as a "left deep" relational algebra tree of the form `((((table1 join table2) join table3) join table4) join table5)` where `table1` is the smallest table and `table5` is the largest; push all of the selection and join predicates as far down as they can go. This may work for some of the queries, but it won't work for all of them. I am quite sure that if you want to get all of the test queries to work, you are going to have to write some sort of "real" optimizer. I would suggest that you consider implementing the exhaustive search algorithm from class. I think that this is the easiest solution. The query with the largest number of tables has only nine of them, so a reasonable exhaustive algorithm should be able to search through all of the options reasonably quickly, especially if you keep a table of all of the problems you've already solved (the key of this table would be the set of tables that were optimized over and the value would be the resulting set of statistics and total cost).

To make this is a bit easier, I have already implemented code that will allow you to cost a tree of `LogicalOp` objects; you can simply use my code without modification to do the costing. This code more or less implements the costing algorithms that we described in class though it does take a few shortcuts to simplify the problem. Also note that my code assumes that the table statistics are available in the catalog; unless you've actually loaded the various database tables with data the statistics will not be available, and it won't be possible to use my costing algorithms. So make sure to load up the tables so that you can get the statistics (once the stats are in the catalog, you can go and delete the files; you won't actually need them as you implement the optimizer).

The second thing that you need to be able to do once you can create a tree of `LogicalOp` objects is to execute your tree. This is actually a lot easier than actually building the tree, as you will simply call the appropriate `RelOp` objects to execute the various `LogicalOp` objects in the tree. Note that you'll need to do at least some rudimentary physical optimization such as choosing whether to do a sort merge join or a scan join. Make sure that your code cleans up any intermediate tables that have been created!!

One big suggestion that I'll make is that unless you are fairly confident in your coding abilities and how well you understand the system at this point I would not just start coding and trying to get everything to work at first (including all 12 test queries). I would start by writing code that is able to execute the two-table join queries that the source code distribution is already able to build a `LogicalOp` tree over. Get that to work and try out a bunch of two-table queries. Then extend that code and make it so hat you can also handle one-table queries as well as aggregation. At that point you will be up to a C

on the assignment (70%; see below). I think that most groups will be able to do this in about 6-8 hours of work in all (maybe less if you are really good!). Only after all that is working should you attempt to handle more complex queries and implement optimization.

4 Testing, Grading, and Turnin

In the end, when the user issues a query, your program should evaluate it, and then print at most the first 30 records from the output file to the screen, the number of records resulting from running the query, as well as the number of seconds taken to run the query.

Your grade will be based upon a turnin doc that you need to submit separately from your code. This needs to be in PDF (while we will grade your PDF doc, we'll also be running a few of the test queries using your database, just to make sure that everything works correctly).

Included with the assignment is a suite of twelve 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. Copy and paste the output of your system from each of the test queries into your doc. Skip any queries that you can't run. The first five queries that you can get to run correctly will give you 14% of the total credit on the assignment each. So if you can run five of them you will get 70%. Note that five of the twelve queries only reference one or two database tables (with and without aggregation) so you should be able to get those to run fairly easily. The next seven queries that you get to run correctly are worth 4% of the grade each. So if you can get all of those to run correctly you are up to 98%.

To get the last two percentage points you need to be able to use a B+Tree to speed query processing. Add a B+Tree index to the `lineitem` table (you'll be indexing the table using the `l_receiptdate` attribute) and then re-run query 1 and query 11. Query 1 should be considerably faster and query 11 should be a little bit faster after using the index.

There is also a chance for some extra credit. Note that the code I've provided prints out the cost of the logical query plan used. For each of the queries 7-12 that you are able to get within 15% of the optimal cost, we'll give you an extra 1.5% on the assignment, for up to an extra 7.5% total.

For turnin, you need to include the PDF file that includes all of the results you get from running those 12 queries (plus the additional runs with the B+Tree if you did that). Make sure it has both team member names. This should be turned in along with all of your source code as a zip archive. Please turn in two different files: one PDF, and one zip.

And also: if you work with a partner, **only turn in one submission**. Otherwise, we'll possibly grade your submission twice, and end up getting quite annoyed.

5 A Note on the Due Date

Note that the assignment is due on the last day of the semester. This is a busy time for everyone; if you need more time, remember that the last day to ask for an extension is one week before the due dae.

6 A Note on Space

Many people will want to run this on Clear. If this is the case, you will likely run out of space (that is, running these queries will cause you to exceed your quota, because the database tables are large, and some of the joins will produce a lot of data). If you would like to run on Clear, you should post a note to Piazza asking for a higher quota; I can then send in a request to have your quota increased for the next couple of weeks. But if you want to do this, make sure not to wait until the last second!