COMP 530 Assignment 6: Implementation of Relational Ops

Due Wednesday, April 20th at 11:55 PM.

1. The Task

Your task, very simply, is to provide implementations for the four relational operations in `RelOps/headers` (the implementation of the fifth operation, `ScanJoin`, is provided for you as a template). At a high level, the five operations are:

1. `ScanJoin`. This is a join to be used when one of the input tables is small enough that all of its pages can be stored in the buffer pool, and the smaller table's contents indexed using a hash table. Then, the larger table is scanned and joined with the indexed table.

2. `SortMergeJoin`. This operation should perform a full, sort-merge join of the two input relations. You should use the sorting code as a starting point for implementing this operation. In the ideal case, you will have a three pass algorithm: sort the two input tables into runs (one read pass and one write pass) and then in the third pass you do the merge/join.

3. `BPlusTreeSelection`. This performs a relational selection over data stored in a B+-Tree, and makes use of the B+-Tree to perform the selection.

4. `RegularSelection`. This is the simplest operation, as it just scans the input table and writes the output.

5. `Aggregate`. This performs aggregation and grouping over an input table. You should implement a hash-based method to do this. As a hint: what you want to do when you evaluate the aggregation is to create a schema that can store all of the required aggregate and grouping attributes. Then you can store records having that schema on a bunch of anonymous, pinned pages. To index the records (via a pointer to each record, like what I did in the `ScanJoin`) you'll need to get the address of the records when you write them on the pages. What I would suggest that you do is to append records to these page using the `appendAndReturnLocation` method. This returns a pointer to the location where the record was written, so that you can use that pointer to find the record later. To do this, store the poster in an appropriate C++ hash structure (sort of like what I did in the `ScanJoin` implementation). If you need to update the aggregate, you can use `fromBinary ()` on the record to reconstitute it, then change the value as needed (using a pre-compiled `func` object) and then use `toBinary ()` to write it back again. You might worry that since records are packed into a page, you'll end up writing over the next record when you write back, but this is actually not a worry, since when you compute an aggregate, you only update `double`s and `int`s, which are fixed length. Thus the record cannot

change in size.

The interfaces for these operations are a bit intricate, but they are not that different from one another, so it behooves you to carefully look that the `ScanJoin` code that I've provided you with to figure out what is going on.

One of the interesting things about this assignment is that it will pull together various parts of the system: database tables, the buffer manager, the schema infrastructure, the expression evaluation engine, and more. It is a lot to take in… the overall code base is now more than 8,000 lines! That's another reason to look carefully at the `ScanJoin`, because you can use it as a basis for your implementation of the other operations.

2 Testing And Grading

Again, we're using `Qunit`. Again, I have written a simple `Qunit` test to show you how to run the `ScanJoin` operation.

You'll need to create additional test cases of your own to check your implementation of the other operations.

When you turn in your code, and it's time for us to grade, we'll run our private test suite. You'll be graded on your code's success in passing all of the test cases, though we revere the right to browse through your code and take off additional points if it appears you are missing some functionality or have somehow hacked something in a sketchy sort of way. You won't be graded on style and comments. However, I strongly encourage you to take this opportunity to put your best software engineering practices to use.

3 Project Difficulty

This assignment is more difficult than the sorted file and B+-Tree assignments, but easier than the buffer manager, in my opinion. One nice thing is that the classes you are writing are totally independent of one another. `Selection`, for example, is quite easy.

4. Turnin

Simply zip up all of your source code and then turn it in on OwlSpace (make sure to archive into the zip format, and not some other archiving format. If you choose to use something else, we'll take off a few points!). Please name your archive `A6.zip`. Please do not change the original directory structure, except for perhaps adding some new files. The root should be a directory called A6, with two subdirectories Build and Main. And so on.

And remember, **to get *any* credit on A6, your code must compile and run on Clear**. That way, we have a common environment for grading and we don't have to spend time getting your code to compile.

Finally, and this is important: **include a README file in the root of your project with any important information**, including the names of the one or two people who worked on the project.

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