COP 6726 Assignment 3: Relational Operations

Due March 23, 2021 11:59PM

In this assignment, your task is to implement a set of relational operations, specifically:

`SelectPipe`, `SelectFile`, `Project`, `Join`, `DuplicateRemoval`, `Sum`,

`GroupBy,` and `WriteOut`. These operations will all be encapsulated within classes that are derived from the following, virtual base class:

```
class RelationalOp {

public:

    //blocks the caller until the particular relational operator
    //has run to completion

virtual void WaitUntilDone () = 0;

    //tells how much internal memory the operation can use virtual
void Use_n_Pages (int n) = 0;


};
```

With a couple of exceptions, operations always get their data from input pipes and put the result of the operation into an output pipe. When someone wants to use one of the relational operators, they just create an instance of the operator that they want. Then they call the `Run` operation on the operator that they are using (`Run` is implemented by each derived class; see below). The `Run` operation sets up the operator by causing the operator to create any internal data structures it needs, and then `Run` it spawns a thread that is internal to the relational operation and actually does the work. Once the thread has been created and is ready to go, `Run` returns and the operation does its work in a non-blocking fashion. After the operation has been started up, the caller can call `WaitUntilDone`, which will block until the operation finishes and the thread inside of the operation has been destroyed. An operation knows that it finishes when it has finished processing all of the tuples that came through its input pipe (or pipes). Before an operation finishes, it should always shut down its output pipe.

Simple enough, right? Now, one at a time we consider each of the operations that are derived from this base class.

```
class SelectPipe : public RelationalOp {

public:
```

```
     void Run (Pipe &inPipe, Pipe &outPipe, CNF &selOp, Record
&literal);

}
```

`SelectPipe` takes two pipes as input: an input pipe and an output pipe. It also takes a `CNF`. It simply applies that `CNF` to every tuple that comes through the pipe, and every tuple that is accepted is stuffed into the output pipe.

```
class SelectFile : public RelationalOp {

public:

     void Run (DBFile &inFile, Pipe &outPipe, CNF &selOp, Record
&literal);

}
```

`SelectFile` takes a `DBFile` and a pipe as input. You can assume that this file is all set up; it has been opened and is ready to go. It also takes a `CNF`. It then performs a

scan of the underlying file, and for every tuple accepted by the `CNF`, it stuffs the tuple into the pipe as output. The `DBFile` should *not* be closed by the `SelectFile` class; that is the job of the caller.

```
class Project : public RelationalOp {

public:

     void Run (Pipe &inPipe, Pipe &outPipe, int *keepMe, int
numAttsInput, int numAttsOutput); }
```

`Project` takes an input pipe and an output pipe as input. It also takes an array of integers `keepMe` as well as the number of attributes for the records coming through the input pipe and the number of attributes to keep from those input records. The array of integers tells `Project` which attributes to keep from the input records, and which order to put them in. So, for example, say that the array `keepMe` had the values [3, 5, 7, 1]. This means that `Project` should take the third attribute from every input record and treat it as the first attribute of those records that it puts into the output pipe. `Project` should take the fifth attribute from every input record and treat it as the second attribute of every record that it puts into the output pipe. The seventh input attribute becomes the third. And so on.

```
class Join : public RelationalOp {

public:

     void Run (Pipe &inPipeL, Pipe &inPipeR, Pipe &outPipe, CNF &selOp,
             Record &literal);
```

```
}
```

`Join` takes two input pipes, an output pipe, and a `CNF`, and joins all of the records from the two pipes according to that `CNF`. Join should use a `BigQ` to store all of the tuples coming from the left input pipe, and a second `BigQ` for the right input pipe, and then perform a merge in order to join the two input pipes. You'll create the `OrderMaker`s for the two `BigQ`'s using the `CNF` (the function `GetSortOrders` will be used to create the `OrderMakers`). If you can't get an appropriate pair of `OrderMakers` because the `CNF` can't be implemented using a sort-merge join (due to the fact it does not have an equality check) then your `Join` operation should default to a block-nested loops join.

```
class DuplicateRemoval : public RelationalOp {

public:

    void Run (Pipe &inPipe, Pipe &outPipe, Schema &mySchema);

}
```

`DuplicateRemoval` takes an input pipe, an output pipe, as well as the schema for the tuples coming through the input pipe, and does a duplicate removal. That is, everything that comes through the output pipe will be distinct. It will use the `BigQ` class to do the duplicate removal. The `OrderMaker` that will be used by the `BigQ` (which you'll need to write some code to create) will simply list all of the attributes from the input tuples.

```
class Sum : public RelationalOp {

public:

    void Run (Pipe &inPipe, Pipe &outPipe, Function &computeMe);

}
```

`Sum` computes the `SUM` SQL aggregate function over the input pipe, and puts a single tuple into the output pipe that has the sum. The function over each tuple (for example:

`(l_extendedprice*(1-l_discount))` in the case of the TPC-H schema) that is summed is stored in an instance of the `Function` class that is also passed to `Sum` as an argument (the `Function` class is included in the project tar file).

```
class GroupBy : public RelationalOp {

public:

    void Run (Pipe &inPipe, Pipe &outPipe, OrderMaker &groupAtts,
Function &computeMe);

}
```

`GroupBy` is a lot like `Sum`, except that it does grouping, and then puts one sum into the output pipe for each group. Every tuple put into the output pipe has a sum as the first attribute, followed by the values for each of the grouping attributes as the remainder of the attributes. The grouping is specified using an instance of the `OrderMaker` class that is passed in. The sum to compute is given in an instance of the `Function` class.

```
class WriteOut : public RelationalOp {

public:

    void Run (Pipe &inPipe, FILE *outFile, Schema &mySchema);

}
```

`WriteOut` accepts an input pipe, a schema, and a `FILE*`, and uses the schema to write text version of the output records to the file.

What To Turn In:

1. Turn in your submission via Canvas. Turn in your code, output1.txt, and your report in a zip file called *firstNameLastName1_firstNameLastName2_p3.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 a3test 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. Include the following:

   a) All code needed to compile test.cc and a2test.cc
   b) Create 4 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

The script generates a file called output1.txt. Turn in output1.txt along with your code. 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) A screen shot of output1.txt generated from your code.
   d) Screen shots of GTest results that match results generated by your code.
   e) 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 from runTestCases.sh

+30 report