CSE177/EECS277 – DATABASE SYSTEMS IMPLEMENTATION
Project 3: Single-Table Relational Algebra Operators
Due date: April 7 (TA's office hours)

This project requires the implementation of all the single-table relational algebra operators: Heap Scan, Selection, Projection, DuplicateRemoval, Sum, GroupBy, and WriteOut. With these operators, it is possible to run simple `SELECT-FROM-WHERE` SQL queries.

Heap Scan

A heap file stores the records of a table. The records are grouped into pages, which represent the I/O unit. Pages eliminate the requirement to access the file (disk) for every record. This improves the I/O bandwidth utilization. Records are stored in arbitrary order in a heap file. The only access path to a heap file is to read sequentially all the records, from beginning to end.

Class `DBFile` (`headers/DBFile.h` and `source/DBFile.cc`) contains the interface of a heap file. It is straightforward:

- `Create` creates a new heap file. `FileType` has to be `Heap`. This is done only once, when a SQL table is created.
- `Open` gives access to the heap file. The name is taken from the catalog, for every table.
- `Close` closes the file.
- `MoveFirst` resets the file pointer to the beginning of the file, i.e., the first record.
- `GetNext` returns the next record in the file. The file pointer is moved to the following record.
- `AppendRecord` appends the record passed as parameter to the end of the file. This is the only method to add records to a heap file.
- `Load` extracts records with a known schema from a text file passed as parameter. Essentially, it converts the data from text to binary. Method `ExtractNextRecord` from class `Record` does all the work for a given schema.

You are required to implement all these methods. The good news is that we already provide you with all the functionality to access paged files on disk. Files `headers/File.h` and `source/File.cc` contain two classes. Class `File` provides all the necessary functionality to access the pages from a file. Class `Page` stores records on pages. Files `headers/Record.h` and `source/Record.cc` contain the class `Record` that implements all the necessary functionality to handle a table record. The methods from the heap file `DBFile` have only to invoke the methods of these two classes correctly.

The `Scan` relational operator is set up by the query compiler. At runtime, method `GetNext` is invoked repeatedly. `Scan` reads the records from the heap file sequentially and passes them to the operator invoking `GetNext`. You have to invoke the methods of class `DBFile` for this.

Selection

Implement the `GetNext` method from the `Select` relational operator. Call `GetNext` for the producer operator. For every returned record, apply the selection predicate and, if the record satisfies the predicate, pass it to the invoking operator. `GetNext` returns when a record satisfying the predicate is found or no records exist anymore. Method `Run` from class `CNF` checks if the record satisfies the predicate, passed as a record of constants.

Projection

Implement the `GetNext` method from the `Project` relational operator. Call `GetNext` for the producer operator. For every returned record, apply the projection and return the trimmed record. Class `Record` has a `Project` method that does the job for you.

DuplicateRemoval

Implement the `GetNext` method from the `DuplicateRemoval` relational operator. You have to use a set-like data structure. Whenever a record is generated by the child operator, check to see if it appears in the set data structure. If not, return it to the caller operator and add it to the set. If it appears, ask for another record from the child operator. Repeat the process until a record can be produced or no more records exist. The most complicated part is to compare two records in order to find if they are identical or not. Class `OrderMaker` already implements this functionality in method `Run`. It is important to remember that the `DuplicateRemoval` operator appears at the top of the tree, above `Project`, and below `WriteOut`.

Sum

Implement the `GetNext` method from the `Sum` relational operator. Apply the `Function` to every record produced by the child operator. Keep a running sum that is continuously updated with the result of `Function`. When all the records are processed, create the result record containing only the sum and pass it to the parent operator. Method `Apply` from `Function` does all the work.

GroupBy

Implement the `GetNext` method from the `GroupBy` relational operator. This is a combination of the `DuplicateRemoval` and `Sum` operators. Replace the set-like data structure in `DuplicatRemoval` with a `Map` having as key the grouping attributes and as value the running sum. `OrderMaker` over the grouping attributes allows you to run comparisons between records. For every record produced by the child operator, check to see if it appears in the map. If yes, compute the function on the aggregate attributes and add the result to the running sum. If no, add the new grouping attributes to the map and initialize the running sum with the result of `Function` applied to the record. Remember that records are produced from `GroupBy` only after all the records in the child operator are processed. The order in which you generate the records is not important. However, remember that a single record is returned at-a-time. The sum aggregate appears in the first position of the created record, followed by the grouping attributes.

WriteOut

Implement the `GetNext` method from the `WriteOut` relational operator. Call `GetNext` for the producer operator and write the returned record in `outFile`. Class `Record` has a `print` method for this. `outFile` is set to an arbitrary file.

Requirements

- Load the data from the text file into your database, i.e., create a heap file for every table. The TPC-H data in text format are available in the `data` folder. File `code/project/phase-3-data-loader.cc` contains the driver code. The program reads the schema corresponding to the table from the catalog. Then, it creates the heap file and opens it. Finally, it invokes method `Load` with the schema and the

text file. `Load` invokes `ExtractNextRecord` for all the lines in the text file and appends the resulting binary records to the heap file. Finally, the heap file is closed.

- Execute the simple queries we provide you in folder `queries/phase-3`. For this, you have to implement method `ExecuteQuery` from class `QueryExecutionTree`. The method simply calls `GetNext` for the root node until no more records are generated. File `code/project/test-phase-3.cc` is the driver to perform the queries.
- For correctness and performance analysis, compare the results you obtain with the results generated by some other database server, e.g., `SQLite`.