CSE 177 / EECS 277 – DATABASE SYSTEMS IMPLEMENTATION

Project 1: Database Catalog
Due date: Tuesday, February 18 (in the lab)

This project requires the implementation of a **database catalog** or **metadata**. The catalog contains data about the objects in the database, e.g., tables, attributes, views, indexes, triggers, etc. The catalog is used by many components of a database system. The query parser checks that all the tables/attributes in a query exist in the database and that the attributes are used correctly in expressions, e.g., you do not divide a string by an integer. This is called semantics analysis. The query optimizer extracts statistics on tables and attributes from the catalog in order to compute the optimal query execution plan. The statistics are created/updated during a statistics maintenance operation executed at certain time intervals.

To keep matters simple, the catalog in this project contains only data on **tables** and their corresponding **attributes**. For each table, the **name**, the **number of tuples**, and the **location of the file containing the data** are stored. For each attribute, the **name**, **type**, and **number of distinct values** are stored.

`Catalog` Class

The code (folder `code` in GitHub) includes class `Catalog`. The interface is given in `headers/Catalog.h`. Catalog reads all the data from a SQLite database file at startup. All the subsequent operations are executed on the memory-resident data structures. When the system is stopped, the catalog content is materialized to disk in the same SQLite database, so that, when the database is started again, the new content is preserved. Saving to the database can also be triggered at other time instants.

Class `Attribute` declared in `headers/Schema.h` and implemented in `source/Schema.cc` is a container for the elements of a table attribute. These are read from the SQLite database. When a table is created, only the name and type of attributes have to be specified. The number of tuples and of distinct elements in each attribute is updated directly in the SQLite database. We will provide these values later in the project.

Class `Schema` declared in `headers/Schema.h` and implemented in `source/Schema.cc` is a container for all the `Attributes` in a table. There is a `Schema` object for every table in the database. The order of the attributes in the schema requires care. It has to be identical to the order in which attributes are stored in the physical record representation. It is recommended to store the position of an attribute in the database and, when the `Catalog` is read, create the schema following the order of the attributes in the database.

`headers/Config.h` includes global constants and definitions used across the entire project.

`headers/Swap.h` includes two macros for swapping classes that implement the assignment operator (=) and `STL` objects, respectively. `Swap` moves the content from one object to another, in each direction.

Also included in the project code, a series of generic (template) data structures such as:

- `Vector` (`headers/Vector.h`, `headers/Vector.cc`)
- `List` (`headers/List.h`, `headers/List.cc`)
- `Map` (`headers/Map.h`, `headers/Map.cc`)
- `Swapify` (`headers/Swapify.h`, `headers/Swapify.cc`)
- `Keyify` (`headers/Keyify.h`, `headers/Keyify.cc`)

may be helpful for implementing the functionality of the catalog.
`Swapify` and `Keyify` classes are templates to create objects that can be used by the generic containers from any primitive data type. A few examples are provided in the header files. `List` requires `Swapify` (it actually requires a `Swap` method), while `Map` requires `Keyify`. To be precise, `Map` requires `IsEqual` and `LessThan`.

These container data structures are fully-implemented and ready to use. Folder `tests` includes sample code on how to use these data structures in your code. `tests/test-vector.cc` contains sample code for `Vector`. `tests/test-list.cc` contains sample code for `List`. `tests/test-map.cc` contains sample code for `Map`.

`tests/test-sqlite.cc` contains sample code on how to interact with a SQLite database from C/C++. The test implements a complete database application that creates tables in a SQLite database, populates the tables with data, queries the tables, and, finally, drops the tables. This application is similar to how a database is used in any other programming language, including Python, JavaScript, etc. This is what was the end-product of the CSE 111 project. You can compile and execute this test as described below. The Catalog application you are required to develop in this project follows the same pattern as tests/test-sqlite.cc.

`project/test-phase-1.cc` contains sample code that uses the `Catalog` class. The code calls the methods from the `Catalog` interface API declared in `headers/Catalog.h`. Since none of the methods are implemented, the code in this test does not do anything. Only after you implement the methods in `source/Catalog.cc`, you will get output by running the test.

`makefile` contains the compilation definitions. To compile the code, type `make test-phase-1.out` at the terminal in the project code directory. The executable `test-phase-1.out` is generated in a newly-created folder `execs`. The tests for the container data structures can be compiled similarly with `make test-vector.out`, `make test-list.out`, and `make test-map.out`, respectively. All the executables are created in `execs`. In order to run any of the executables, you have to type `./execs/test-***.out` in the command line, where `***` stands for the test you want to execute. In order to compile all the tests with a single command, run `make all` in the command line. `make clean` deletes all the object files (folder `object`) and executables (folder `execs`) resulted from the compilation process.

Requirements

- Create a SQLite database `catalog.sqlite`. Inside the database, create the necessary tables to store the catalog data on tables and attributes.
- Implement the interface of class `Catalog` as defined in `headers/Catalog.h`. Write your code in `source/Catalog.cc`. You are responsible for declaring all the data structures and interfacing with the SQLite database. Additionally, you can create new methods in class `Catalog`. However, the interface, i.e., the declared methods, of the class cannot be modified.
- Study the code in `project/test-phase-1.cc` and execute it with `./execs/test-phase-1.out`. Provide the required arguments and check the correctness of the output. This test checks most of the methods in the `Catalog` interface. As long as you pass the test, your `Catalog` should work fine. You are welcome to extend `project/test-phase-1.cc` with additional tests or to write a completely different test. If you add a new test file, make sure to include it in the `makefile`.
- Ideally, all your code has to be written in `source/Catalog.cc`. There is no need to do it differently, but if you plan so, discuss with the instructor/TA.
- Required packages: `libsqlite3-0`, `libsqlite3-dev`, `sqlite3`, `sqlite3-doc`.

Resources

- `http://www.sqlite.org/cintro.html`
- `http://www.tutorialspoint.com/sqlite/sqlite_c_cpp.htm`
- `http://www.tpc.org/tpch/default.asp`