Joshua Dahl CS457

Programming Assignment 1 Design Document

The basic design of the project is very similar to what was proposed in the project write-up. A folder is used to represent each database with a special **.metadata** file in that folder holding metadata about the database. Additionally, the folder holds a **<tablename>.table** file that specifies all of the metadata and data of each table belonging to the database. The metadata files are serialized to or from a binary stream which is written to or read from disk. A third-party library called SimpleBinaryStream was used to facilitate this process. Additionally, all of the file system manipulations are performed using C++'s std::filesystem library. As a final step in high-level management, we store a root database folder path, which is currently a constant referring to the current working directory, which defines where to look for existing databases and where to create new databases.

A database's metadata consists of its name, path, and a vector of table paths. Additionally, a map from table paths to table pointers is present and will be used to keep tables around so they don't need to be constantly loaded from the disk.

A table's metadata consists of its name, path, a vector of columns, and a vector of records. Columns represent one column of data in the table and store the column's name and datatype. Records represent one row of data and are responsible for storing the data. Currently, data is represented as a variant (tagged union) of 64 bit int, double, and string. We determine how to serialize and deserialize the data based on the datatype of the column the data belongs to.

Behind the scenes, the user is presented with an input prompt provided by the linenoise library,a cross-platform rewrite of GNU readline. Linenoise provides history input, meaning that users can use the up arrow key to quickly re-enter SQL statements they had previously entered (this is very useful when mistakes are made). Once the input has been received from the user, we discard any comment lines and check for special command lines. If what we receive is neither of those commands we pass the input off to the SQL parser.

The SQL parser was written using the lexy library which provides a C++ syntax for defining both combined lexer and parser. It is isolated from the rest of the program in its own translation unit and interfaced with using a single parse function which returns a (possibly null) pointer to a transaction. This isolation was put in place because lexy performs much of its work at compile-time, meaning that the current very simple SQL parser takes about 10 seconds to compile.

If the SQL parser fails to parse the provided SQL, it will provide an error message to the user and return a null transaction pointer. A transaction holds information about what to target (database or table and its name) and what action to perform on the target. Additionally, there are extended transaction types for table creation, alteration, and query which contain additional information necessary for those operations.

Build and Run Instructions

The project has been developed using cmake and git submodules, all of the files required to build the project should be in the distributed zip folder (with the notable exception of the boost libraries which can be easily installed via the package manager).

To actually build navigate to the root directory of the project (the one containing thirdparty and src) and run:

```
mkdir build
cd build
cmake ..
make
```

Once the project has been built it can be ran using:

```
./pa1 # In the newly created build directory
```

There are currently no command line options, simply start entering sql in the provided prompt, but the ".exit" command can be used to close the application.