Joshua Dahl - CS457
Programming Assignment 4 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 a constant referring to the directory the program was launched in, 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. Likewise, a Table's metadata consists of its name, path, a vector of Columns, and a vector of Tuples. Columns represent one column of data in the Table and store the column's name and datatype. Tuples represent one row of data and are responsible for storing the actual data. Data is represented as a pointer to its associated column and a variant (tagged union) of 64-bit int, double, bool, string, and an additional state representing null. We determine how to serialize and deserialize the data based on the data type of the column the data belongs to with an extra boolean in front to determine if the data is null. This is an optimization so that we only need to store a single boolean value in the file for null data.

For any Table manipulation, we load a copy of the Table into memory and then save a serialized version back to disk once we are finished. This ensures that even if an operation fails part way through, none of the data on the disk will be corrupted. Before we load the data into memory we check if a **<Tablename>.table.lock** file exists; if this file exists it will store the thread identifier of the process that locked the table. We check if that process is the same as this process and if not the operation fails. If the lock file doesn't exist (and there is currently a Transaction) the process creates it, adding its thread identifier to the file.

The program maintains a pointer to the current Transaction, or a null pointer if there is not currently a Transaction. A Transaction in the program manages a map of table paths to temporary table paths. These temporary paths append the current process's thread identifier to the name of the table. Whenever a table manipulation tries to load a table, it first checks if the current Transaction has that table in its map and if so it loads the temporary table instead. Whenever a table manipulation tries to save a Table, it checks if there is an active Transaction and if so it creates a new temporary table (adding that new table to the Transaction's map) and saves the data to that temporary Table. When a Transaction is committed, all of the temporary Tables that the Transaction touched override their respective original Tables. When a Transaction is aborted all of the temporary Tables that the Transaction touched are discarded. Either way, any locks associated with the tables in the Transaction are released. If there is not currently a Transaction, the whole process of writing to a temporary table is bypassed and manipulations modify the Table directly.

To insert data into the Table, we simply append a new tuple to the back of the Table's vector of tuples before saving it back to disk; we have a helper method on Tables that create a new Tuple with the Column pointers properly set and all of the data nullified. To modify data in the Table we filter out all of the Tuples that don't match the user's provided "where conditions", and then simply change the appropriate bits of data before saving it back to disk. A "where condition" is represented by a Column name, an enumeration storing all the possible comparison types, and a value to be compared. Deletion is similar to modification, except the matching Tuples are removed before saving the Table back to disk.

Selection is more complicated; first, we create a new temporary Table composed by loading the first table, then successively applying a Cartesian Product of the temporary table with additional tables the user requested to join. After each join, we add an extra column to the table containing the index of the data. Once all of the tables have been joined, we create a new table with only the Tuples matching the user's "where conditions." If the user requested a Left Outer Join we then scan through to determine the table indices we have included and invert that set to add all of the left tuples we didn't select paired with matching null right data. We then remove Columns (and their associated data) from the temporary table to implement projection, making sure to remove the extra index columns we added to facilitate Outer Joins. Finally, the temporary Table is displayed to the user.

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 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 a 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 an Action. 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 Action pointer. An Action holds information about what to target (Database or Table and its name) and what kind of manipulation to perform on the target. Additionally, there are extended Transaction types for Table creation, alteration, data deletion, data modification, data insertion, and data query which contain additional information necessary for those operations; most notable being a vector of “where conditions.”

**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 .. # If this step fails try again with: CXX=g++ cmake ..
make
```

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

```
./pa4 # In the newly created build directory
```

There are currently no command line options. Simply start entering SQL in the provided prompt.
The “.exit” command can be used to close the application.

A demo (demo.mp4) is included, it shows the program running with operations multiplexed between the two processes. During the entire demo the folder representing the database is open in the top left where all of the changes being made can be observed.

**NOTE: My DBMS follows the capitalization rules of standard SQL, thus identifiers (Tables, Columns, etc…) are case sensitive while other keywords are not.**