Samuel R. Gerard

CS 457

SQL database

Python Implementation Version 2 (PA_3 Updated):

How To Use This Program (PA_3 Update):

1. This Program was tested on the university linux environment and shouldn't have any dependency issues
2. To execute the program just do a simple: **$python3 main.py**
3. The program comes with a **PA3_test.py file** that acts as an imitation to the PA3_test.sql file provided with the assignment prompt. The **program will prompt you with a message if you would like to use that PA3_test.py file**. If you use the test file, all commands just like in PA3_test.sql will be inputted and executed and you will receive the same output as if you typed everything in. Otherwise, you're free to type in all the input commands by hand
4. When you end the program, whatever databases were created and allocated will be saved in it's own file in the directory along with a bunch of .tbl files of the tables of the database(s) within it.

How does the program store data?

The program uses a cross between basic python list, tuples, and dictionaries to store the data. In the db_abstract.py file, there is a class known as _db_abstract() which is the base class that acts as a database for holding all of it's tables and other important metadata. Within the database class is a table tuple that stores all tables of the database with a key name and an object

reference to the database. As for the handling for the databases, just like in SQL, only a single database can be used at a time when managing it's tables. A class called db_context is used for handling a single database object and a time and storing available databases for use at current runtime. Additionally, a single variable known as 'db_runtime_context' is an instance of that db_context class and is initialized in the argument parsing module known as argument.py. That single object is used to keep track of the database currently used by the user. This objects has a bunch of other functionalities that deal with managing the database

How are general SQL functionalities done?

The general idea is that for each function (E.G: create, drop, and select) each of these keyword functionalities has their own class with helper functions to execute the designated request the user calls. As an example, say the user calls for a new database to be created ('create database tbl_1'); python interprets command line inputs as strings so the string needs to get parsed. When the program sees that it is a create argument, the argument fields 'database' and 'tbl_1' get passed to be made into a create_argument() object from CreateModule.py. When the class is established, the creation argument gets executed. When the database is finished being created, a prompt is given that the creation is successful, otherwise it returns a failed prompt.

Tuple modification (updating of tables)

To handle data modification, we first parse the UPDATE statement into three parts: the table name, a dictionary that maps column names to values, and a conditional statement. To update the table when a WHERE clause is supplied, we iterate through the rows stored in the table and if the condition matches for that row, we use the dictionary to change the values of the columns. If no WHERE clause is specified, all rows in the table are updated with the new values.

Deleting rows from tables

When a delete command is called, it is parsed into three components: the table name, a list of attributes and their names, and the conditional. A call to the equivalent table is and and the corresponding rows are pulled from the list. Those rows then get deleted, if there is no condition present, the whole row gets deleted.

Queries

When a select command is called, the same parsing function that was used in DELETE is called to handle the WHERE clause. The Head of the table is then displayed along with the corresponding data from the WHERE clause.

Insertions

When an insert command is called, the Insert command along with it's passed arguments get parsed into two things: the name of the table and the values that shall be inserted. The a new row is appended into the table objects schema.

Aliasing

Aliasing To implement aliasing functionality in select statements, we created a dictionary which mapped alias names to table names when we parsed the select statement. When determining which tables to select from, we reference the alias map if an alias was present in the select statement.

Inner Joins

To implement inner joins, we implemented parsing for both explicit inner joins (using the "inner join" keywords) as well as implicit inner joins (using a where conditional). The algorithm operates the same as the pseudocode outlined in the assignment document. For each tuple in the left hand table, iterate through the tuples in the second table to find the matching value in the

specified attribute, when found, both tuples are added to a new temporary table which is then printed.

Outer Joins

To implement outer joins, we kept track of what type of join was specified in the SELECT statement when we parsed it using special constants. Once we did that, we grabbed the tables that were being joined from our database object (see above), and then merged the tables into one large temporary table using a nested loop outer join algorithm. Once we had the merged table, we gathered the data from it as we would in a normal select statement without joins and displayed that to the user