Yeamin Chowdhury
Project 4
CS 457
April 24, 2023

Programming Assignment 4

Design

This program is designed to create databases exclusively in the home directory where the program resides. Tables can only be made within a database, not outside. The SQL USE query must be used to create a database and its tables. Without specifying a database, creating or querying tables is impossible. By employing the USE command, the current working directory is set as the chosen database, allowing file creation within it through CREATE commands. Although users can have multiple directories (databases) from the home directory, nesting databases is prohibited. However, multiple tables can be created within each database. Keep in mind that tables cannot be created without first selecting a database. Users can add data according to the table's defined schema using INSERT commands. The SELECT command offers four different functionalities: displaying everything from the selected table; performing inner joins on two tables with the default join command; performing inner joins on two tables with the inner join command and performing left outer joins. These are the functionalities that are pre-existing and from project 3.

Additionally, for Project 4, transactions are implemented for multiple users. In this program, especially, a table can not be updated if any other user has already started a transaction using “Begin transaction;”, and needs to wait until the other user commits the transaction by using “Commit;”. This was implemented using a locking mechanism, so different users can not write on the table at the same time; the transaction will be aborted. Finally, the program can be exited using the ".exit" command.

Functions

main()

The primary function anticipates user input consisting of commands accompanied by arguments. It can accommodate up to three lines of valid input unless a line ends with a semicolon. The input process terminates if a semicolon appears at the end of a line. Otherwise, it asks until the 3rd line of commands to pass the commands for conditional checks of functionalities. The contents of these three lines are consolidated into a single string, which is then passed to the parsing stage. The string is converted to uppercase and assessed to determine if the input is '.EXIT'. If this condition is met, the program terminates; otherwise, the string is forwarded to the command() function for command execution.

beginTransaction():

The beginTransaction function serves a crucial role within a database context by initiating a transaction while implementing a locking mechanism. This mechanism is based on the existence of a file named "locked" in the current working directory, which signals whether a transaction is underway or not.

As a global variable, the transaction variable can be accessed or modified anywhere in the script since it is externally defined. The "locked" file is then checked for existence. When a "locked" file is present, indicating an ongoing transaction, the function sets the transaction variable to False so that the "Begin Transaction;" command is used again before updating the table. However, if the "locked" file is absent, signifying no active transaction, the function sets transaction to True to initiate a new transaction. It then creates a "locked" file by opening it with write permissions. Once the "locked" file is successfully created, it is immediately closed, and the function prints "Transaction starts." to the console, indicating the successful initiation of a new transaction. It uses the fileinput library to implement the locking mechanism.

commit():

The commit function's purpose is to complete a transaction that was started earlier with the beginTransaction function. It uses a simple locking mechanism, where the existence of a file named "locked" indicates an ongoing transaction. This function also relies on a set of global variables - transaction, table_name_global, data_global, counter_global, and table_header_global.

Firstly, the function checks whether data_global, table_header_global, or table_name_global are empty. If any of these are empty, it signifies that there's no active transaction to commit, and the function prints the message "No active transaction." before returning. Next, it checks if the "locked" file exists in the current working directory. If the file does exist, this signifies that a transaction is ongoing. In this case, the function first sets the transaction to False, indicating the end of the transaction. The "locked" file is then removed using the os.remove function, unlocking the transaction. The function then prints the message "Transaction committed." to signify the successful completion of the transaction. The write_to_file function is called with table_name_global, data_global, counter_global, and table_header_global as arguments. This function is used to write the updated data from the transaction to the respective file (table).

After writing to the file, table_header_global, data_global, and counter_global are reset to their initial states. This is done to ensure that any data from the previous transaction doesn't interfere with future transactions.

write_to_file(table_name, two_d_list, count, first_element):

The function write_to_file is used to write the updated data from a transaction to a specific table (file). The function takes four parameters: table_name, two_d_list, count, and first_element. ‘table_name’ is the name of the table (or file) where the data will be written, ‘two_d_list’ is a two-dimensional list containing the data that will be written to the file, ‘count’ is the number of records that have been modified in the transaction, ‘first_element’ contains the header information of the table. Then, it prints the modified data to the file with the specified formatting of the database table.

update_table(table_name, set_name, attribute_name, operator, operator_value, new_value):

This function, like the insert_into and delete_from functions, necessitates the correct table_name and prior utilization of USE commands. An auxiliary function, **"update_operation(operator, two_d_list, attribute_index, set_index, operator_value, new_value)"**, is employed to evaluate the suitable operator and update values corresponding to the specified condition. If the conditional portion of the commands contains incorrect data types, an error message will be displayed and the function will return to the message and return to main. If transaction variable equals to False, that means another transaction has already started on the table and aborts the

transaction. Otherwise, it will set the table name, header of the table, and the data to the global variables to be used by write_to_file(table_name, two_d_list, count, first_element) to commit the updates.

create_database(database_name):

This function takes a database name as its parameter and creates a folder as the database. The database is created in the home directory(where the program is located) and can not be created anywhere else. Nested folders are not allowed.
If the database with the entered database_name already exists, then it will print an error message.

use_database(database_name)

This function takes a database name as its parameter and entered the folder as the database, only from the home directory. If the database does not exit with the specified name, it prints an error message. This function must be used before the creation, deletion, alteration, and selection of the tables.

create_table(table_name, variable_list)

This function takes a table_name as its parameter and creates a table with the specified name, otherwise, prints an error message if it already exists in the database. The USE commands with a database name must be used before creating a table because the creation of files outside of any databases is not allowed.
This function also takes a list of variable that was entered in the command to add schema in the table. The variables from the commands are parsed in the command function and passed to the create_table function. The variables are printed inside the table, joined by a “ | ”.

select_table(table_name)

This function accepts a table_name parameter and accesses a table with the given name in the database; if it cannot find the specified table, it displays an error message and exits.
Subsequently, it opens the file associated with the table name and reads its contents. The function then outputs the file contents to the terminal before closing the file.

insert_into(table_name, value_list)

This function accepts a table_name as a parameter and generates a table with the specified name, or displays an error message if the table is not present in the database. The USE command, accompanied by a database name, must be employed prior to selecting a table, as multiple tables may coexist within a single database. The number of parameters provided in the command line must correspond to the existing table attributes. If there is a mismatch, an error is printed and control returns to the main function. If the parameters match, the new set of parameters is appended to the table following the existing records.

select_join(t_dict, a_dict, operator)

This function takes t_dict parameter to take the table names with their alias and a_dict parameter to take the attribute names to take the attributes that will be compared. It also takes a operator, which is "=" for this program. Select_join performs inner joins based on the specified attributes and a condition, which in this case is equality. First, it checks if the current working directory matches the program's directory and verifies if the tables exist. It prints an error message if either of these conditions is not met. After reading the content of the tables, the function stores each line as a list of items. It then creates a cross product of the two lists, using the **cross_product** function. Then, it checks if the given attributes for comparison are valid. If it is not, prints error and returns to main. If it is valid, runs the comparison and find the indices of the rows that matches the condition with help of "**indices_of_records_that_match_condition**". With those indices, this function allows us to print or display the values that should be received after performing a join or inner join on two tables. The current version of this function, operates two commands: **join**(with where keyword**) and inner join.**

select_join_outer_left(t_dict, a_dict, operator)

This function takes t_dict parameter to take the table names with their alias and a_dict parameter to take the attribute names to take the attributes that will be compared. It also takes a operator, which is "=" for this program. Select_join performs inner joins based on the specified attributes and a condition, which in this case is equality. First, it checks if the current working directory

matches the program's directory and verifies if the tables exist. It prints an error message if either of these conditions is not met. After reading the content of the tables, the function stores each line as a list of items. It then creates a cross product of the two lists, using the **cross_product** function. Then, it checks if the given attributes for comparison are valid. If it is not, prints error and returns to main. If it is valid, runs the comparison and find the indices of the rows that matches the condition with help of "**indices_of_records_that_match_condition**". With those indices, this function allows us to print or display the values that should be received after performing a join or inner join on two tables. Ater performing the inner join, achieved rows performs union operation with the list that contains the values of the left table. It performs the union operation on just the length of each row of the left table. If the left table has only two elements, it will perform the union operation on with just first two elements of the list the that was returns from "**indices_of_records_that_match_condition**". After performing the union operation, the new result is stored in a new list. And prints them with join " | " and "\n" at the end of each row to show left outer join.

cross_product(list_1, list_2)

This function areturns a list containing the cross products of two lists. It is a helper function for select_join and select_join_outer_left function.

separate_attributes_from_types(attributes)

The first line of the a file contains the attributes with type. A list of variable with types are passed into this function to separate the variable names and return the names in a list to find the indices of the attributes that needed to be compared by the program.

indices_of_records_that_match_condition(cross_product, index_1, index_2, operator)

This function finds the indices of the rows of the cross products list to show the value for default joins or inner joins. It performs the comparison with the operator between the columns given by index_1 and index_2 It is a helper function for select_join and select_join_outer_left function.

Test Script

-- On P1:

CREATE DATABASE CS457_PA4;
Database CS457_PA4 created.

USE CS457_PA4;
Using database CS457_PA4.

create table Flights (seat int, status int);
Table FLIGHTS created.

insert into Flights values (22,0); -- seat 22 is available
1 new record inserted.

insert into Flights values (23,1); -- seat 23 is occupied
1 new record inserted.

begin transaction;
Transaction starts.

update flights set status = 1 where seat = 22;

-- On P2:

USE CS457_PA4;
Using database CS457_PA4.

select * from Flights;
SEAT INT | STATUS INT
22 | 0
23 | 1

begin transaction;
Transaction starts.

update flights set status = 1 where seat = 22;
Error: Table FLIGHTS is locked!
Transaction abort.

select * from Flights;
SEAT INT | STATUS INT
22 | 0
23 | 1
-- On P1:

commit;
Transaction committed.

select * from Flights;
SEAT INT | STATUS INT
22 | 1
23 | 1

-- On P2:

select * from Flights;
SEAT INT | STATUS INT
22 | 1
23 | 1