Project
Database management system

Report

Patrick Tomasini Malatesta
18205940

COMP30640: Operating Systems 2018-2019

UCD School of Computer Science

Introduction

A database management system (DBMS) is a type of software which is commonly used to perform different tasks and manage data stored in a database.
In other words, a DBMS organizes your files to give you more control over your data.
A DBMS is a powerful tool that allows users to create, edit and update data in database files. Once created, the DBMS makes it possible to store and retrieve data from those database files. In order to guarantee user-friendly access, the user-database interaction is performed through query languages, the most popular being SQL.

The key features of a complete DBMS are:

- Concurrency: concurrent access to the same database by multiple users
- Security: security rules to determine access rights of users
- Backup and recovery: processes to back-up the data regularly and recover data if a problem occurs
- Integrity: database structure and rules improve the integrity of the data
- Data descriptions: a data dictionary provides a description of the data

In this project we will have a closer look to the concurrency feature of a basic DBMS, assuming that the other characteristics do not need to be implemented.

Requirements

The simple DBMS that we are implementing is composed of different pieces that, when combine together, will create a piece of software that allows users to create, modify end retrieve data in/from a database.

In our system each database will be represented by a folder which name is `$database` name. Within each database there can be any number of tables containing different information. Each table is represented by a comma separated file with `$table` name. In a comma separated file the information is represented with the comma acting as a column separator. Each file can have any number of columns and tuples (rows).

Our DBMS will be composed of three main parts:

- basic commands
- the server
- the client(s)

<u>Basic commands</u>

The basic commands which will allow the user(s) to perform different actions will need to be implemented within the server.

The commands that our DBMS needs to have are:

- create database: the user will create a new folder with a given name that will then store the tables
- create table: this command will allow the user to create, inside a database folder, a file with a given name and with a specified header (representing the number of columns)
- insert data: this feature will allow the user to insert a tuple (row) in a specific table file
- select data: using this command the user will be able to query the database in order to retrieved the desired information from a specific table (e.g. a particular column of data)

Each of these commands must also provide adequate error handling. In general, the user will have to be informed by an error in the following scenarios:

- the requested action is not in the correct form (e.g. too many/few parameters given)
- the folder/file that the user is trying to create already exists (e.g. database folder already exists)
- the format of the existing file does not match the request (e.g. if the user tries to insert a new tuple with *n* columns while the destination table has a different number of columns)

If the request is in the correct format and can be performed, the user will be notified with a confirmation message that everything went well.

<u>The server</u>

The server is what operates in the background and manages a set of databases.
The server will need to read the input from the clients and execute the requested command. This means that the server will need to always be 'listening' for instructions (this will need to be implemented through named pipes).
Error handling must be implemented as every request will have to follow a specific form and the user must be notified if a request has been sent in the wrong format.
The server will accept 5 requests:

- `create_database`
- `create_table`
- `insert`
- `select`
- `shutdown`: which will allow to exit correctly the server

As the server will be used by multiple users sending different requests at the same time, those commands will need to run in the background to allow concurrency.

The concurrency of the system is crucial in a DBMS and in order to allow many clients to access the system at the same time, we must make sure that the different requests won't interfere with each other (e.g. we can't allow more than

one user to create the same table inside the same database folder simultaneously).

The clients

The client(s) script(s) will send requests to the server which will be 'listening' at all times. Each client will need to have a unique identifier (client ID) representing a specific user.
As mentioned above, this also means the client must make sure that the request is in the correct format. If the request is correct the client will print the query (including the client ID) that will be sent to the server.
The clients and the server will need to be connected and in constant communication. This will be performed though named pipes where the server will have its own pipe (`server.pipe`) and each client will generate the relative pipe (`clientID.pipe`). With this approach we allow the client to send requests to the server and the server to send the outcome of the query back to the client.

We must also implement a way to efficiently exit the client(s) and the server. We should make sure that the client can send a shutdown request to the server as well as understanding an exit command that will quit the client.

Architecture/Design of the DBMS

In order to achieve the requirements presented in the previous section, the solution approach required several steps:

1- Creation of basic commands:
   the first step was to write the scripts `create_database.sh`, `create_table.sh`, `insert.sh` and `query.sh`.
   - `create_database.sh:` in this script we first check if the number of parameters entered by the user is correct, we then create a directory (which represents our DB) with the name given by the second argument (`$1`). We then print an error if the DB already exists or we confirm that the folder is created;
   - `create_table.sh:` similar to above, we first check if the number of parameters entered by the user are correct, we then check if the folder (DB) exists and if it doesn't we give an error, then we check if the table already exists and in that case we signal it to the user. If the request passes all the checks, we create a file representing a table (can be imagined as a `.csv` file) inside the DB folder and give a confirmation message;
   - `insert.sh:` also in this script we check for the correct number of parameters, the existence of the DB and the existence of the table. In addition to this, we have to make sure that the number of columns that we are trying to add, matches the columns in the existing table. In order

to do this the script counts the number of the columns in the existing table (if any) and the columns in the tuple the user wants to insert. If there is a difference, we give an error otherwise we append the tuple in the existing table;

- `query.sh:` this script does the usual error checking (number of parameters, the existence of the DB and the existence of the table). In addition, we need to check if the user is requesting data from a column that does not exist. This is done by the script counting the number of columns in the table and finding the min and max column number of the query. If the max column number requested is not in the table OR if the min column number requested is 0 then we print an error. If the query is correct the output will be given between `star_result` and `end_result`. If the user does not specify any column (i.e. there is no `$3`) the script outputs every column otherwise it prints only the selected column (using the cut function).

2- Creation of the server:

the script `server.sh` includes an infinite loop that constantly reads input. For testing purposes, initially the input was given through the terminal, after the implementation of the client the server reads from a named pipe.

In order to work with the input read by the script, we cast it into an array which splits the input and gives an index to each part. We can then assign a variable to each input index (e.g. index[0] will always be a basic command, index[1] will be the clientID, index[2] DB, etc.). The script then checks if the command (or index[0]) is valid and, if it is, it will call the relative script (e.g. the command select will activate the script `query.sh`). Each script will be executed with the arguments given in the input request (using the specific idexes). The server will recognize 5 commands including the 4 basic commands previously described and a "shutdown" instruction that will terminate the server. If the command given is not recognized an error will be displayed. To have a better understanding of what the server is doing, messages will be shown in the terminal (e.g. server waiting for instruction).

3- Semaphores to allow concurrency:

If we assume that the server will be accessed by many users at the same time, we must make sure that the server can run processes in the background at the same time. We also have to make sure that there is no interference between users requests and running process (e.g. when two commands access or modify the same file at the same time).

To allow processes to run in the background the symbol "&" has been added at the end of each basic command call in the `server.sh.`

To avoid synchronisation issues, semaphores scripts (`P.sh and V.sh`) were added in the critical section of the 4 basic commands scripts. The script `P.sh` creates a link with an atomic operation (`$name-lock`) that will be removed by `V.sh` once the operations in the critical section have been performed. This will guarantee that if more than one user tries to perform the same action at the same time, only one user will be able to

perform the task while the others will have to wait for `V.sh` to remove the lock.

In more detail, depending on the basic command, different things can be locked. In our DBMS in the `create_database.sh` the script itself is being locked, this does not allow the creation of more than one DB at the time (e.g. if 2 users try to create the same DB at the same time). In `create_table.sh` the database folder is locked by `P.sh` (i.e. will not be possible to create, within the same DB, two tables with same name at the same time). In `insert.sh` and `query.sh` the semaphores are applied to the table, this guarantees that if a user is asking for information about a table (with the select command) the table will be locked and no modification will be allowed until the scripts exits the critical section and the lock on the table is removed.

4- Creation of the client:

The `client.sh` script will send requests to the server and identifies each client with an ID. If no ID is given an error will be displayed, otherwise, the client enters an infinite loop that reads the user's input from the terminal and casts into an array. The script checks if the request is in the correct format (`command arguments`) and if it's not, an error will be displayed. In this phase we assume that, since the error handling has been already implemented in the basic command scripts and in the server script, the client only checks if the number of given parameters is correct. If the user enters the correct number of parameters but the command doesn't exist, the request will be sent to the server anyway and the client will receive an error message as a response (we use the `"Error: Bad request"` option implemented in `server.sh`).

With the user input split into an array we can do the error handling and print the request, including the client ID, in the format that will be sent to the server (`command clientID arguments`). To do this the script uses a loop that iterates through the array and adds each argument to the initial pieces (`command clientID).`

After the request is sent to the server through the `server.pipe`, the client will read the server response through the `clientID.pipe`.

Finally, I added a few `sleep` commands which may help the user to understand the steps that the program follows.

5- Communication with named pipes:

Now that all the needed scripts have been created, they need to communicate to each other. This key feature is implemented using named pipes which will be our "communication channels". Once initiated, `Server.sh` creates a unique named pipe `server.pipe`. `client.sh` will create a specific pipe for each client id as `clientID.pipe`. After the pipes are generated the communication can take place, in particular, the client script reads the user input and will then redirect it to the server through `server.pipe`. Now the server scripts can read the instruction from `server.pipe` and perform the requested command (we cast the input into an array for easier handling). The server now need to send

back to the client the result of the request. To do this, the output of each basic command is redirected to each specific `cliendID.pipe`. This design allows the server to send back to output to the correct client, this is possible because the client request includes the client ID, which is extracted by the server script and used to differentiate the clients accessing the server.

Once the command output is redirected back to the client through the `clientID.pipe`, the client script can read the server response and display it to the user. In order to display the required information to the user, the client script reads from the pipe with a newline delimiter (this allows to display multiple line results). Furthermore, if the response from the server begins with "`OK:`" or "`Error:`" we know that the command was executed correctly (the "`Error:`" is given by the error handling implemented in the other scripts) and we display a message to the user. If instead the request begins with "`start_result`" we want to display the result of the query removing the first line (`start_result`) and the final line (`end_result`), to do so we use `sed.`

6- Exit and shutdown commands:

The last phase was the implementation of exit and shutdown commands. To do so a trap function was added to the client script, this function allows to modify the behaviour of the keyboard exit command `CTRL+C`. This extended functionality allows to delete the clientID named pipe when quitting the script and print a notification for the user. A similar function was also added in the server.sh that allows to quit the server and delete the `server.pipe` by hitting `CTRL+C.`

Since the client script reads user input, a few exit options were implemented: the command "exit" will quit the client.sh and remove `clientID.pipe`, the command "shutdown" will be sent to `server.pipe` and will quit server.sh and remove `server.pipe`, this command will also quit the client script that sent the request (since the client will not be able to operate without the server, there is no reason to keep the script open). We can now close the server and each client, but what happens if a client requests the server to shutdown and other clients are still active?

To solve this issue the client script checks if the server is active (i.e. if there is the `server.pipe`) and if it's not the client will quit before sending the query to the server pipe.

Challenges and Solutions

These are the main challenges I faced during the project:

`query.sh`

determine if a requested column does not exist

To solve this problem, I followed an approach where the script counts the columns in the table and then it finds the max and min column indexes requested by the user (to do this I `echo $3,` I substitute comma with newline, I sort and then pick only the first value). Once I had these values I checked if the user requests a column that was not in the table (e.g. column 0 will never be in the table; a column with index higher than the number of columns in the table will net exist either).

show the specific column(s) requested

The second problem was to find a way to extract the requested columns (if in the table). To perform this I use `tail` (to remove the heading) piped into `cut` where the delimiter is a comma "," and the selected fields are `$3`.

`server.sh`

`server.pipe: Interrupted system call`

This error appeared every time the server was executing a command and was solved simply by adding a `sleep` command. This is added after each process is being sent in the background.

How to manipulate the input read from the server named pipe

I found that the easiest solution to work with the input was to cast it into an array (with `read –a`) and then assign, based on the index, each array value to a corresponding variable.

`client.sh`

How to manipulate the input from the user

Also in this case I used the technique to cast the input into an array. I found it difficult to print the request in a different format (i.e. including the client ID), I used a loop to make sure that the correct amount of arguments is added. This way the basic commands can spot if there is a parameter problem and send back an error to the user.

How to read the server response from the `clientID.pipe`

It was challenging to read the server response from the clientid.pipe and display it correctly as only the first row was showing. I solved the issue using `read –d '/n'` which allows the input to be read line by line.

How to display result of select query

At first I had troubles finding a way to display the columns of the select query removing `start_result` and `end_result` but I solved this using `sed '1d;$d'` which eliminates first and last rows.

General challenges

How to communicate to all the clients if the server was down.

I decided to solve this problem by adding a simple check in the client script that verifies if the `server.pipe` file exists and if it doesn't (i.e. the server is not active) the client won't send the command but will quit.

What to lock with the semaphores

The first issue was given by the type of links that `ln` generates, this was solved by creating a symbolic link using `ln –s`.

Depending on the command, different things could be locked. So I made the following decisions:

- in the `create_database.sh` I lock the script itself, this does not allow the creation of more than one DB at the time (e.g. if 2 users try to create the same DB at the same time)
- in `create_table.sh` the database folder is locked (i.e. will not be possible to create, within the same DB, two tables with same name at the same time)
- in `insert.sh` and `query.sh` I decided to lock just the table, this guarantees that if a user is asking for information about a table (with the select command) the table will be locked and no modification will be allowed until the scripts exits the critical section and the lock on the table is removed. Locking just the table also allows the DB folder to be accessible by other clients (e.g. to create a new table or send queries to another table).

Extra features

I have implemented 2 additional scripts to allow the user not only to create but also to delete information.

- `delete_database.sh:` the script allows to delete an existing folder representing a DB. The script includes error handling and outputs the result of the request.
- `delete_table.sh:` the script allows to delete an existing table inside a DB folder. Also in this case the program includes error handling and outputs the result of the request.

These two commands have also been implemented in `server.sh` in addition to the basic commands requested.

Conclusion

To conclude, this was a complex project where many challenges had to be faced (and `bash` is not always your friend). However, I found it helpful that the assignment was broken down into small and relatively simple scripts. I also liked that only at the end all the pieces would need to communicate with each other to form a rudimental DBMS.

I found interesting the fact that we were able to apply and implement the different commands and `bash` features that we learned throughout the semester to solve a “real” problem.
It was also fascinating to have a deeper understanding of the general versatility of `bash`, and to analyse a problem taking into account also the aspects and possible issues relatively to the operating system.