COMP30640 Project report – DBMS

Giovanni Facchinetti – student ID 18202941

1. Introduction

This project is aimed at building a relational Database Management System application with multiple clients sending requests to a central server, that takes care of the overall database management and executes legal client requests.
A Database Management system could be defined as a "software system that enables users to define, create, maintain and control access to the database" (Elmasri et al., 2010) and its main purpose comprehends the storage, update and the fetching of data.
The basic system we are trying to build in this project is a multi-user Database Management System which is meant to run on a single machine, accessed through a terminal or terminal emulation software. With this regard, this constitutes a simplified abstraction of real world DBMS applications, where the client–server architecture typically consists of a number of clients accessing the database via some desktop end-user interface and a server that ensures the process distribution.

2. Requirements: what is the system supposed to do?

In our case, the server is supposed to manage a set of different databases, with each coming in the form of a directory with an assigned name. As our DBMS relies on the Relational Database Model, each database directory contains a number of tables that store data in comma-separated values.
The system is aimed to provide the following functions:

- Data definition – the user is allowed to create, modify and remove of data items, operations are carried out by the server upon legal user requests;

- Data update – the user is allowed to insert, modify, delete actual data items (i.e. data that are already present in the tables), operations are carried out by the server upon legal user requests;
- Data retrieval, i.e. showing relevant information. In our simplified version the user is allowed to select different columns that will be showed for his own information purposes. This constitutes the rudimentary query application of our DBMS, and the selected columns are required to exist in the actual queried tables.
- General administration, as the server track and monitor users that are accessing the database. At server-side we try to preserve data-integrity and achieve concurrency control (e.g. with the use of P(S) & V(S) semaphores, as well as client-server communication via named pipes).

3. Architecture and design

The system is designed in three different segments: the server commands, the server program, and the client program.

The basic server commands are built in the form of four simple bash shell scripts that are meant to run on server-side via command line interface. They are aimed at provide the basic essential DBMS functions:

- Create database (create_database.sh) – a script that takes one legal argument and creates a database (which in our abstraction is a file system directory). It provides a validity check for the number of arguments passed (if more than one argument is passed or if no argument is passed, it returns an error message) as well as a check to verify that the database being created does not exist already in the system (in that case, another error message is returned to the screen).
- Create table (create_table.sh) – a script that creates a table in a selected database directory with columns given by the user. It provides a validity check for the number of arguments passed, as the user is prompted to enter three different parameters:
    - First parameter: name of database directory

- Second parameter: name of table to be created
- Third parameter: table columns, separated by a comma (without any spaces).

If the user does not strictly comply with the three parameters required, the script returns an error (parameters problem). In the case that the argument test is passed, the script then checks if the database directory exists (a new table must be created inside an already existent database directory) and if a table with the same name already exists in the database directory. If a table exists already and has the same name of the table to be newly created, the script returns an error. With the aim to avoid inconsistencies in the case of multi-user requests, we identified the table creation line in the script code as a Critical Section and subsequently we placed P(S) and V(S) binary semaphores respectively before the Critical Section and right after it. The semaphores come in the form of two different bash scripts (p.sh and v.sh):

- p.sh represents P(S). After checking that the first parameter is entered by the user (and which should refer to the database directory in which he wants to create a new table) and after checking that it actually exists, the script achieves mutual exclusion by creating a lock referring to the database directory, i.e. it "keeps trying" to generate a "lock-folder" and if it cannot (in the case some process is already in the Critical Section and so a lock folder has already been generated), it enters a loop section that keeps it waiting for 1 second each loop run.
- v.sh represents V(S). After checking that the first parameter is non-empty (i.e. the db directory has been passed to the create_table.sh script as the first parameter), it removes the folder lock to the database directory. It is meant to run whenever a process is successfully exiting the critical section, so that another waiting user process is now made allowed to enter the Critical Section and thus proceed to create another new table.

- Insert data (insert.sh) – aimed at insert tuples in an existing table, accepts only 3 parameters like the create_table.sh above and checks few things prior to proceed:
    - db folder existence
    - table existence
    - checks that a valid tuple is being inserted. To perform this check, the script ensures the values to be inserted as a tuple correspond to the number of columns in the selected table by separating the column headers by each comma, placing them on new lines and counting those lines first and subsequently compare them with the count of tuple values after having separated them by comma and placed each one of them on a new line.

  The identified Critical Section here is represented by the code line in which the tuple are effectively inserted in the table, therefore P(S) and V(S) semaphores are deployed before and after it to avoid inconsistencies as in the create_table.sh script. The script returns error messages if any of the aforementioned checks fails.
- Select data (query.sh) – aimed at selecting some columns identified by numbers that the user is prompted to enter. As usual, the user should enter the database name first, followed by the table name and (as the third argument) the number of the selected columns he wants to be displayed on the screen. The script performs at first the usual error checking about the number of arguments passed - the existence of the db directory and the existence of a table (must have been previously created). Exceptionally, two arguments can be accepted (a valid db name and a valid table name) and the entire table is displayed as a result. In the case that the user enters specific column numbers (separated by comma, no spaces) that he wants to see, the script checks then if the selected columns are valid. To do so, it verifies that no column with the value "0" is entered by the user and that the selected numbers (comma-separated) do not represent empty columns in the table (in this case, we assume that the column does not exist). For example, if the table has only 3 columns and the user enters the number 4,

the script will find that at position "4" in the table the column is empty. All these column number errors are tracked in a counter, so that the select script performs the required operation only if the error counter is equal to zero (that is, no errors are detected).

The server-client architecture is based on named pipes to make processes communicate and pass information between each other.

The server program runs on a script called server.sh and as it starts execution, it creates a named pipe called "server.pipe" (with the command "mkfifo"). Server.pipe will serve as a unique collector of requests from the multi-client channel: each client process will write the users' requests on the server.pipe and the server program will then read them in sequence, so that the server reads inputs from the server.pipe. The server executes the requests as it reads them on its pipe, and will write the outcomes sorting them to each "private" named pipe that are specific to each client program (i.e. user). The syntax for the commands passed from client to server is as follows: request, database folder name, table name, column numbers.

The server works entering in an infinite loop and keeps reading the commands and executing every request in the system background, and a selection amongst different possible action is provided via a "case" statement: the server basically compares what is coming from server.pipe to a number of predefined commands (i.e. create_database, create_table, insert, query, shutdown) and if a request corresponds to each of those, it proceeds with execution calling the basic command scripts explained above and writing the outcome to the individual client pipe. Identification of client pipes is made through client id numbers, that users are requested to enter whenever they start their client.sh programs.

The client processes execute the code contained in the script client.sh, and it prompts the user to enter a valid identification number that will be also the identifier for the server answer that will be written on the specific client named pipe (a validity check is performed and the client.sh program is terminated if the user fails to enter a valid id number). The first thing that client.sh does when it has accepted a valid id number from the user is to create a client pipe that will take the name [client id].pipe, so it will be unique for each user sending requests to the server. The client process will then enter an infinite loop followed by a prompt (command “read”) through which the user will enter the desired request with the aforementioned syntax. The client script then writes the user request on the server.pipe and waits for the server to write a response on the [client id].pipe. The scripts will read the response from [client id].pipe and print it on the screen. In case of a shutdown request or a bad request (i.e. unrecognised syntax) the script will remove the [client id].pipe to perform a clean exit.

Both server.sh and client.sh are equipped with an “exit function” that is called when the ctrl+c keyboard combination is pressed on the keyboard, causing an interrupt: it prints an alert message on the screen indicating that the script is about to exit and removes respectively server.pipe and [client id].pipe to perform a “gentle” and clean exit after a user-triggered abrupt interruption.

4. Challenges faced

The main challenged I faced are listed as follows:

- How to manage all the “exits” in each script case, especially after error checking: whenever an action is considered “legal” and its inputs are checked as “valid” it produces an exit code of 0, whereas some error occurs, they return the exit code of 1 or greater numbers to distinguish between the different types of errors that have occurred;
- Building the semaphores P(S) and V(S) and place them at the right lines in the code: I tried to position them right before and immediately after what I considered to be the Critical Section in the server basic commands (i.e. the creation of database directory line, the creation of table line, the appending tuples line and the query show line in each of the server’s basic commands). I chose to lock the database directory in order to allow an ordered chain of process execution in the case of multi-user access and in order to avoid inconsistencies. Semaphores are not necessary in the create_database script, since in this case it makes no sense to lock a directory that does not exist yet.

- In the insert.sh script, the most difficult task was to build a validity check for the tuple the user is trying to enter in a table: to perform this error handling I used a pipeline of commands:
  - Head [-n 1] to display the first row of the table (headers) piped into:
  - Grep [-o ","] to find the exact match of commas in the table headers, piped into:
  - wc [-l] to count the lines produced by the previous command.

  This pipeline is applied to both the existing table and the tuple that the user is trying to insert, in order to verify if such table has all the values required by the columns in the table (I chose to compare the number of commas as a proxy of the number of columns).
- In the query.sh script, the most difficult task I faced was to check if the entered column numbers by the user actually exist in the table. To perform this validity check I built a for loop as following:

  column=$(echo $3 | tr "," " ") → create a string with column numbers entered by the user separated by spaces instead of commas

  count=0 → initialise the counter for "bad" cases

  for col in $column; do → start the iteration over the col numbers entered by the user

  if [$col -eq 0] | | [$(cut -d "," -f $col "$1/$2" | wc -w) -eq 0]; then ((count++)) → check first if the entered col numbers are 0 (invalid value) or if their content in the table is empty (I assume that if nothing is inside those columns, they don't exist in the table).

  If the counter of "bad" cases remains at zero, it means that the columns entered by the user are valid and thus exist in the table: we can then proceed to enter the critical section and print the selected columns on the screen. To do that, I wrote this pipeline: *cut -d "," -f$3 "$1/$2" | tail -n +1 | tr '\n' ' '*

  Where I cut the table columns with "cut" using commas as delimiters and taking the entered column numbers as fields to be selected. I pipe then the output into the command "tail" to show every row of the table (including headers, this is my choice as I think that it is more useful to see headers in

the table output) and finally I pipe the result into “tr” to print the result on a single row instead of newlines (separating each row with a space). I had to find this workaround as the named pipes seem not to accept multi-line result to be printed and then read. In the client.sh I had then to put newline characters back when reading the result from the client pipe, and I did it through the pipeline
echo $message | sed 's/ /\n/g'
Echo reads the message from the client pipe and its output is piped into the stream editor “sed” to substitute spaces with newline characters, allowing the program to display the data nicely back in table form.

- I faced some difficulty when I tried to understand the whole system of writing/reading messages to/from named pipes, and especially how to distinguish between a single server.pipe where every client program writes requests and the server writes the results on multiple, individual [client id].pipes specific for each client that has started the client.sh script.

5. Conclusions

I found this project quite complex and pretty challenging mostly because of my inexperience with Bash shell scripting. Overall I felt satisfied when I made all the script working and when I understood the whole structure of such rudimentary Database Management System.
I chose not to pursue bonus points as I don’t feel confident nor familiar enough with this language to enter such unexplored territory, but I appreciated the fact that this project allowed me to work in group with class peers. This has proven to be really important to absorb further concepts and ways to and approach problems.

6. References

Elmasri, R., Garcia-Molina, H., Ullman, J. D., Widom, J., Özsu, M., Valduriez, P., … Virk, R. (2010). *Database Systems: A Practical Approach to Design, Implementation, and Management. International Journal of Computer Applications*. https://doi.org/10.1007/978-1-4842-0877-9_10