# COMP30640 - Database Management System

**Student Number:** 12309511

**Name:** Stephen Gaffney

## Introduction:

I found this project to be a very enjoyable yet challenging task. It really helped develop my BASH programming skills and overall knowledge of operating systems and database management systems.

This report will cover the requirements of the database management system as outlined in the project description. It will also cover the architecture/design of the system that was implemented to tackle these requirements. It will cover some of the challenges I faced in constructing this system and how I overcame them. Finally, it will include a short conclusion which will give a brief summary and some key points of what I learnt from this project.

## Requirements:

**Creating the scripts that will be used.**

1.) Create database - Create a directory to store tables, takes 1 parameter.
Differentiate when:
   a. No parameter was provided
   b. The database already existed
   c. Everything went well

2.) Create table - Create a new file in a database folder and write the header of the file (the columns of the table), takes 3 parameters.
Differentiate when:
   a. Too few or many parameters
   b. The database does not exist
   c. The table already exists
   d. Everything went well

3.) Insert data - Write a tuple in a table file, takes 3 parameters.
Differentiate when:
   a. Too few or many parameters
   b. The database does not exist

c. The table already exists
d. The wrong number of columns entered
e. Everything went well

4.) Select data - Display data from a table specified by a query, takes 2 or 3 parameters. Differentiate when:
    a. Too few or many parameters
    b. The database does not exist
    c. The table does not exist
    d. No columns parameter specified, print whole table.
    e. Everything went well

**Server Script:** An endless loop that reads requests by clients from the server pipe, run requests *concurrently in the background* by calling relative scripts outlined above. Return results to client. Creates a named pipe for server. Trap ctrl_c interrupt. Four valid requests:

a. create database
b. create table
c. insert
d. select
e. shutdown

**Client Script:** Client application that sends requests to the server. Takes 1 parameter, ID. Endless loop that reads inputs from the user and if well formed sends to the server to be executed. Creates its own client pipe to communicate with server. Trap ctrl_c interrupt. Exit command.

## Architecture/Design:

This section will cover the design of my system and how it tackles each requirement as outlined above.

**Base Scripts:**

1.) Create Database:
Check number of parameters, error if less than or greater than 1. Print relevant error message.
If 1 parameter, check if directory (database) exists, if not create it otherwise error and exit.
Relevant exit codes for each code. Refer to exit code list in Appendix.

2.) Create Table:
Check number of parameters, error if less than or greater than 3. Print relevant error message.
Otherwise. Check if $1 directory (database) exists, if not error. If it exists, then check if table also exists. If $2 table exists, then error. Else create lock for this database whilst the table is

created. Create $1/$2 and print $3 into file. Remove lock. See below for lock and unlock scripts.

3.) Insert Row:
Check number of parameters, error if less than or greater than 3. Print relevant error message.
Otherwise. Check if $1 directory (database) exists, if not error. If it exists, then check if table exists. If $2 table does not exist, then error.
If table does exist. Compare that the number of columns entered to be inserted matches the number that are in the table e.g. the header. I did this by comparing the number of commas separating columns in each. If column/comma count do match, create a lock for this table which still allows other tables in this DB to be accessed. Print $3 into table. Remove lock and exit. If they do not match, print an error message saying this.

4.) Select (Query columns):
For this script the number of parameters can vary between 2, 3 and 5 (bonus where clause). If no specific columns are requested then the whole table will display whilst $3 will outline specific columns to be returned.

If 2 parameters are given, the existence of database $1 and table $2 needs to be checked. If true then "start_result", contents of the table (e.g. the whole table), "end_result" will be printed. Otherwise error message printed and exit.

If 3 parameters given, and database and table exist then else clause entered.
Variables are assigned;
numberLines – the number of lines in the table.
desiredCols – argument $3 translated to a string without ','s e.g. '1,3,4' translated to '134'.
commaTableHeader – the number of commas in the table's header row. This is achieved by taking the first row of the table file, grep -o ','s which puts each comma on a newline, then count the newlines e.g. the number of commas.
reverse – this is simply the desiredCols variable in reverse which will facilitate indexing the first number, for instance '134' string becomes '431'.

The first index of reverse will be the last column entered by the client in their request. In the table there will be one more column than there are commas in the header row e.g. 'header1,header2,header3' – 2 commas, 3 columns.
If the value of the first index [0] of 'reverse' is not greater than the value of 'commaTableHeader +1' then proceed to next step, otherwise the columns specified do not exist in this table i.e. trying to query column 6 in a table that only has 5 columns.

If the column range specified is not out of the table column range, then print 'start_result' and next two for loops are used to create the table. If the column range is out of the table column range, then an error message will be printed stating this.
The first loop will iterate through the number of lines in the table and will print a newline each time it cycles e.g. for a new row in the table. The second loop is within the first loop and is responsible for printing each cell on each line within each specified column.
The loop ranges from 0 to length of the 'desiredCols' string – e.g. for string '134' the range is 0-2 as it is inclusive with its stop attribute i.e. j will have the value 0,1,2. This value is then

used to index 'desiredCols' string to return the column requested by the client e.g. index 1 in this example will return value/column '3' which is assigned to variable 'fieldIndex'.
This value e.g. '3' will be used as a field index in the next step. The 'substring' variable is assigned to the print of only the line specified by the outer loop e.g. 'i', which is then cut using ',' as the delimiter and 'fieldIndex' as the specified field. For instance, fieldIndex '3' will return the third field split by commas in the table. 'col1,col2,col3' → 'col3'. This substring will then be printed suppressing the newline character.
When the outer loop has finished searching each line a newline character followed by 'end_result' will be printed.

Bonus Where Clause:

For the "bonus" select where query, the parameters are the same as the select query with the addition of the column number of the where value and this value.
For instance: select database table 1,2,4 2 Murphy
--> return columns 1,2,4 where column 2 = 'Murphy'.
The code for this is copied from the original select query and modified to check each line if the value specified exists in the specified column. It will only print if it exists else it will loop to the next line. A counter is used to check if it returns no results which in that case it will return "No results found". If this was an initial aspect of the project requirements, I would merge the two "elif" statements into one to reduce the code however I wanted to keep this "bonus" section separate to my original work for clarity.

5.) P.sh:
This script is responsible for acting as the locking mechanism of a semaphore. It will lock the critical section of other scripts specified by its $1 argument. If no file/directory name is provided an error message is printed. Otherwise it will enter a loop in which it attempts to create a lock for the target file/directory if no link exists. It will exit the loop at this point. Error messages are redirected from standard output.

6.) V.sh
This script is responsible for acting as the unlocking mechanism of a semaphore. It will unlock the critical section of other scripts specified by the $1 argument. If no file/directory name is provided an error message is printed. Otherwise it will remove the lock for the target file/directory.

**Client Script:**

The client script is responsible for allowing users to send requests to and receive results from the server. The client script takes one argument, client ID. If number of arguments is not equal to one, then an error exit message will be printed, and program will be exited.

The client script will check if there is an existing client_pipe for this ID and if there is then someone else is currently using this client ID and the script will exit with an error message. If there is not a client_pipe with this ID then this ID is not in use and client can proceed, it

creates a client_pipe$1 which is unique to this script. “admin” can be entered to run as the administrator.

A loop is entered which prompts user for his/her request. It reads this input and sets input to a variable(‘arr’) in an array. Input format should be ‘request’ ‘database’ ‘table’ ‘headers/cols’ dependent on how many arguments required for each request. The array ‘arr’ is indexed [0] to retrieve the request which is assigned to a variable ‘request’. Once this request is a well-formed e.g. a valid request (‘create_database’, ‘create_table’, ‘insert’, ‘select’, ‘shutdown’) then the client ID ‘$1’ and ‘$userinput’ is passed into the ‘server_pipe’. The server script and the individual scripts will be responsible for the individual request error checking once a valid request is sent. If a bad request is entered, then an error message is printed saying such and the loop is entered again e.g. prompt user for input. If the request ‘exit’ is entered, then the client_pipe for this script is removed and the script is exited with exit code 0. Once a valid request has been sent to the server the client script will grep everything from its pipe that doesn’t contain ‘start_result’ or ‘end_result’ (useful for the select query). It will then enter its loop again until the script is exited or interrupted.

This script also contains two functions for the traps ‘INT’ and ‘SIGSTP’. When this script is interrupted or stopped the client_pipe$1 for this script is removed and script exited.

**Server Script:**

The server script should run continuously while the database wants to be accessible. The server script takes no arguments. The server script will create a named pipe ‘server_pipe’. It then enters a loop where it will read input from the server_pipe e.g. requests received from client scripts. This input is set as the arguments for the respective script to be called. The respective case statement is entered and one of these cases will always be true as the client script will only pass valid requests. Hence the ‘*’ case is now obsolete. *Added “shutdown” request as required. If request “shutdown” is received, then script will check if user requesting this is the admin. If not, an error message is returned to client. If client “admin” requests “shutdown” all pipes will be removed, remaining locks will be removed, and all client scripts will be ended and a message indicating this is returned to the admin client.

The respective script for each request is called with the arguments from $3 on. The output of these scripts is redirected to the client_pipe$1 e.g. the pipe of the client that sent the request. These scripts are run in the background which allows the server script to continuously read from its pipe and create other instances of the scripts concurrently. The loop will only end by using an interrupt or terminal stop. These are both trapped outside the loop at the start of the script. Two functions are defined for both these traps, both functions being the same. These functions are responsible for clean up when the server is interrupted or stopped. It will remove the server_pipe, kill all the user’s client scripts running, remove all client_pipes that exist, remove all locks that may have been left behind if a client script exited in the critical section, and it will exit the server script.

## Challenges:

In this section I will outline several challenges that I faced in this project and how I overcame them.

**Insert Script:**

The first challenge I had to overcome was comparing the number of tuples being inserted to the existing number of headers in the table. This number must match to insert a tuple. The difficulty arose in attempting to count the words entered as $3. What I decided to do was compare what both the tuple and table have in common, commas. For the tuple being inserted I used grep -o ‘,’s using only matching (-o) option (prints only matching parts of the line with each part outputted to a separate line). I then used wc -l to count the number of lines. Similarly, I used grep and wc to count the commas in the first line of the table e.g. the header. The number of commas stored as variables were easily compared and used for error checking.

**Select Script:**

The next and personally the biggest challenge for me was identifying the specified columns for the select script (query script) and the error handling to match this. I tried to avoid using too much new code but inevitably I used a lot of online resources to further my learning and gain a deeper understanding of some of the logic behind such queries. Most notably Stack Overflow.

The main issue was identifying which columns were to be queried and then using this to then return the specified columns. After much research and trial and error I managed to complete this task successfully. I used a combination of indexing strings, tuples and cutting via fields to achieve this. I converted the desired columns into a string and used a loop through the length of this string to index the string to retrieve the value of each character in the string. E.g. ‘1,2,4’ → ‘124’ → ‘124’ [2] = ‘4’. This value was then used to identify the field to be returned by the ‘cut’ -d ‘,’. Some error handling that proved difficult was preventing the user from attempting to retrieve a column that is out of range of the target table. E.g. column 5 in a table with 3 columns. To overcome this, I reversed the string and checked that the value of index [0] was not greater than the number of columns in the table (commas +1).

**Client Script Trap and Interrupt:**

I had difficulty passing an argument into a function, specifically for the traps of interrupt and stop. I quickly learnt the syntax of function calls and learnt that there are many different types of commands that can be trapped.

I also struggled to print more than one line from the client pipe e.g. for the select query. I initially thought the individual select script was only returning one line to the server script but with the use of a print statement I soon realised that it was the client script that was not retrieving all the lines from its pipe. I used grep -v “start_result\|end_result" which retrieves every line but the lines with “start_result” or “end_result” in them.

**Server Script:**

In this script I initially had my functions killing all client.sh scripts on interrupt and stop and it was causing permission denied messages as it attempted to kill client scripts being run by other users. I originally solved this by passing in my user ID "cs12309511", but I was quick to realise that this would render the functions useless for other users. One of the teaching assistants helped to bring "$USER" to my attention which allowed me to carry this out for all users of the script.

The last significant challenge I encountered was that in the server script if I passed in empty arguments into the select.sh script is was causing an issue. This should be allowed for the select query script (specifying no columns will return the whole table). My solution for this was that rather than passing through $3, $4, $5…etc (even when $5 was empty) I passed in ${@:3} e.g. the arguments from $3 onwards, this prevented an empty $5 being passed through.

## Assumptions & Limitations:

For this project I made some assumptions. In my project duplicate rows can be inserted into a table as we did not use any primary keys. Duplicate rows could be avoided by using a primary key and checking against all the primary key values each time a tuple is inserted.

The next assumption is in my select query, I assume that the user will input the desired columns in numerical order e.g. 1,2,4 and not 2,4,1, this is one of the limitations of my script. This could be overcome by iterating through the values and assigning variables to each value and sorting them numerically.

The select query does not use a semaphore as it will not be editing any data. However, if a row is inserted concurrently it will not affect the select results as the number of lines are counted when the select script is executed, and only this number of lines will be printed. If there was a request to edit an existing tuple then a semaphore/lock would be needed for the select query as the select query should not return results that are currently being changed.

The final limitation is that if a client script dies while inside the critical section, the lock for this section will remain in place and no other clients would be able to access that table or database. Restarting the server would correct this problem by removing all locks. However, it would be difficult to identify when the server needs to be restarted.

## Conclusion:

In conclusion, this database management system covers all the requirement outlined by the project description. Some additional features have also been included which help with error handling and clean-up. I thoroughly enjoyed this project and found it very challenging yet rewarding. Initially it seemed like a very large task but after breaking it down into smaller feasible parts, as outlined by the project description, it became much more manageable. This project helped to piece together all the areas that we have used so far in this module and I found myself also using logic and problem solving that I developed in other modules such as Programming I. Following completion of this project I have a lot of interest in operating systems and BASH and I hope to pursue these areas further.

# Appendix:

**Exit & Error Codes:**

0.) "Successful command" - Successful command, no errors.
1.) "Error: no parameter given" – No argument.
2.) "Error: too many parameters given" – Too many arguments.
3.) "Error: database already exists" – Existing database
4.) "Error: too few parameters give" – Not enough arguments.
5.) "Error: database does not exist" – Database specified does not exist.
6.) "Error: table already exists" – Failed to create new table as it exists with that name.
7.) "Error: table does not exist" – Target table does not exist.
8.) "Error: number of columns entered does not match number of columns ($) in this table" – Column numbers do not match.
9.) "Error: columns specified in query out of range, ($) columns in this table." – Columns specified out of column range in target table.
10.) "Error: bad server request, restart server." – Redundant error message now.
11.) "Error: enter only one parameter, client ID number." – No client ID entered.
12.) "Usage $0 mutex-name" – Empty string passed as target for lock (semaphore).
13.) "Error: This client ID is already in use." – Specified ID is already in use.

**README:**

This is the read me file for my database management system. The zipped folder should contain, the README, a pdf report and 8 scripts (create_database, create_table, insert, select, client, server, P, V). Download all scripts and run in terminal.

1.) Run the server.sh with no arguments in one terminal.

2.) Run the client.sh with an ID(s) of your choice or "admin" in one or multiple terminals.

3.) The client will now be prompted to enter their request.

4.) Valid requests:

- create_database database_name
- create_table database_name table_name header_columns(separated by commas)
- insert database_name table_name column_entries(seperated by commas)
- select database_name table_name
- select database_name table_name queried_columns
- select database_name table_name queried_columns column where_value (where clause)
- exit
- shutdown (admin use only)

5.) The client script should be exited using request "exit" however ctrl_c and ctrl_z will also exit and clean-up sufficiently.

6.) The server script can be kept running however it should be stopped before the terminal is closed so that the appropriate clean-up is carried out, this can be achieved by running request "shutdown" as client admin or using ctrl_c or ctrl_z.

Notes on requests & parameter entries:

Only the admin may run the request shutdown which shuts the server and logs off clients. Any entry that requires columns separated by commas cannot have spaces between words and the commas.
For the select query, specified columns must be in order e.g. 1,2,4, not 2,4,1 this will ensure correct error handling is carried out.
For the "bonus" select where query, the parameters are the same as the select query with the addition of the column number of the where value and this value.
For instance: select database table 1,2,4 2 Murphy
--> return columns 1,2,4 where column 2 = 'Murphy'.

**Graphical Representation from Project Description:**