CMP2806M Scalable Database Systems

ASSIGNMENT 1 – Report

January 27th, 2022

Mathews Joy (25186202)

University of Lincoln

Department of Computer Science

A picture containing text

Description automatically generated

***Contents Table***

***Introduction***

This is a report to help dive further into the process taken when choosing the overall design and implementation of the relational database (MYSQL) for a scenario bank constructed using content provided by a written brief. This report will begin by outlining the general design process then move onto the variety of queries written on the database, finally touching on scalability and security concepts/issues to consider moving forward.

***Design overview***

General design process

The overall design process was completed by strongly following the online Microsoft database design guide (support.microsoft.com, n.d.).

An effective database schema for a banking system was needed, ensuring the database was designed to be easily scalable and comply by third normal form standards, in order to reduce the duplication of data, avoid [data anomalies](https://en.wikipedia.org/w/index.php?title=Data_nomaly&action=edit&redlink=1), ensure [referential integrity](https://en.wikipedia.org/wiki/Referential_integrity), and simplify data management.

Firstly to complete the design process some assumptions had to be made, as the brief for this assignment did not give us detailed specifics. One of the initial assumptions made is that all typical bank account types should be possible to make e.g. current, savings and a joint account. This assumption then lead onto the need for a many to many relation between the customer and accounts table, due to joint accounts (more than 1 customer can hold the same account). Another assumption was this bank should be able to be scaled nationally and was not for a single bank branch. For this to be met a branch table was needed allowing a 1:1 relation with the account table. A final assumption that I had to infer was customers could live in same address and contact numbers (landlines), this led onto the need for many to many relations, which I will discuss later in this report.

The structure of the tables was where most of the time was spent. Using principles instructed by lecture content (Fathulla, 2021), I began by identifying the nouns and verbs in the context of our task assigned, for example some of the main nouns I identified (making sure to discard nouns that are irrelevant, misleading names, redundant, attributes, instances, or operations) was “bank”, “customer”, “account”, “loan”, “transaction”. Secondly verbs were highlighted that gave more context to the link between nouns, e.g. a customer “opens a” account, a bank is “offering” a loan to a customer. It then became clearer to pinpoint relations between nouns through logical reasoning, the verbs we spotted, and general research on how real-world banks operate. A customer will be able to open many accounts, and accounts can have many types, so a customer will have a ‘many to many’ relation with account, as an account could be of a joint account type. Loans would link to account via a ‘1 to many’ relation as 1 account can have many loans. Similarly an account would have many transactions, so the account and transaction table would link by a ‘1 to many’ relationship.

After recognising the key nouns (which would become the entities of the database) and establishing links between nouns through verbs (this would help establish the foreign keys between entities), it made it simpler to find the necessary attributes for the tables. Taking the account and customer table as an example, “a bank customer is required to provide their full name, date of birth, address, and telephone number. They are also required to deposit an initial sum of money called an opening balance (opening balance > £50)”, this portion of the brief shows first/last name and date of birth should all be attributes (table columns) of a customer. However for the opening balance and contact number we have to dive further. Opening balance shouldn’t be an attribute of customer as it has no direct meaning to a customer, rather should be an attribute of account. The contact number should not belong to neither account nor customer, but instead be established to the customer via a ‘many to many relationship’, as a customer could have the same number as some other customer, and in order to prevent database redundancy a junction table to form the many to many relations should be created. Primary keys and foreign keys were also made clear during this stage of the design process. Primary keys are the key identifier of the tables in order to uniquely identify each record, and foreign keys help link respective related tables. In addition composite keys were created for the junction tables to allow many to many relations between tables, e.g. between customer and account tables. The composite keys are identified as a candidate key that consists of two or more attributes (table columns) that together uniquely identify an entity occurrence (table row). In the example of the junction table between customer and account, the composite key is the combination of customer\_id and account\_id.

Another important consideration when dictating the design of tables was the datatypes used for each column, this is very important as using the wrong datatype can lead to a lot of long-term issues. One example is when determining the datatype for money/amounts columns such as amount in transaction table. Initially it might be an easy chose to just use “FLOAT” or “DOUBLE”, however these types can cause problems with rounding due to limited precision. Therefore “DECIMAL(13,2)” was used, 13 is the precision (i.e. total length of value including decimal places) and 2 is the number of digits after the decimal point. Decimal eliminates rounding issues and preserves exact precision. A similar considerate process was used to establish appropriate datatypes for columns in the remaining tables.

Additional design inclusions

Some further design elements I justified having in my database design was the inclusion of “CHECK” constraints, these types of constraint adds an additional layer of security for our database by the database admin passing in a check to be met on a specific table, this check would then be ran and if not accepted the insert / update query to the table would not be executed. One example I used check constraints for was for making sure all accounts are opened with an opening balance greater then 50, this stop accounts less then 50 from being created and causes further issues. A further example of check constraints practicality is making sure a transaction is not both outgoing and incoming, this was done by XOR (exclusive or) check.

Text

Description automatically generated with medium confidence

A second feature which I choose to include in my database design was the inclusive of “ON UPDATE CASCADE ON DELETE CASCADE” (to necessary foreign key links). This block of code can be added when adding foreign key constraints too tables. It means that the child data is either deleted (set to NULL) or updated when the parent data is deleted or updated. This can help keep the maintainability of the data and prevent you from having to create additional complex queries such as transactions to keep your data up to date.

Normalisation

The final process of designing the tables was to reduce redundancy by apply the normalisation rules up to third normal form standards (3NF), as followed by Microsoft docs of “description of database normalisation basics” (MaryQiu1987, 2020).In order to satisfy the desired 3NF standards it is critical to initially comply by first normal form (1NF) and second normal form (2NF).

1NF was achieved by ensuring tables had a primary key, there was no column/attribute that allows multiple values to be inserted e.g. in a tuple format, and also no rows where duplicated.

To reach 2NF principles, we need to ensure all relations are “in [1NF] and every non-primary-key attribute is fully functionally dependent on the primary key, then the relation is in [2NF]” (GeeksforGeeks, 2019), thus no partial dependencies should exist. In order to achieve this I took steps such as moving the contact number in its own separate table as a customer could share the same phone number/landline as another customer or have multiple numbers. Same with addresses it has been given its own table, it would not make sense to store addresses with account, as address does not have a direct link to accounts. Another example of conforming to 2NF is separating account into its own table rather then having customer and account as one. This is because it is hard to justify columns such as opening balance with a customer and is more ideal to separate it out.

Third and final step of normalisation was to comply by 3NF standards. Attaining 3NF was possible by complying by both 1NF, 2NF and eliminating any transitive dependencies. When an indirect relationship causes functional dependency it is called Transitive Dependency, e.g. if X -> Y and Y-> Z is true, then X->Z is a transitive dependency. An example of steps taken to achieve 3NF was by moving account type and description into its own table (account\_id -> account\_type and account\_type -> description is true, so account\_id and description is a transitive dependency), similar to transaction types.

Further normalisation such as Boyce-Codd Normal Form or Fourth Normal Form (BCNF of 4NF) is unnecessary for the scope of this application, and with 3NF main data anomalies are eliminated. On the contrary, if a real production load is tested and measured to see how further normalisation effects query time, there could be justification for further normalisation.

***Entity-relationship diagram***

The design process was wrapped up by mapping out the entity-relationship diagram in a ‘crows foot notation’ (www.gleek.io, n.d.). In this notation each table represents an entity (noun), and each table row represents an attribute. The lines between the tables indicate the relations between entities.

Graphical user interface

Description automatically generated*Table

Description automatically generated*

***Queries performed***

Additional queries

An additional query I preformed that was not asked within the assignment brief was creating a “VIEW”, a view is commonly used in order to create a virtual table that from multiple other tables via joins or other forms. It can provide a layer of security as it get rid of any rows or tables not needed/shouldn’t be viewed. I decided to create a view combing the customer, customer\_account, account, and transaction tables as the result of all these tables combined was used later in a lot of queries performed, and to reduce code repetition I created the view “customer\_account\_transaction\_details” to be used. The view created joins the 3 tables by appropriate linking foreign keys/primary keys using an inner join to get only necessary data needed.

Text

Description automatically generated

4 instructed queries

1. *List all bank customers (including their name and account number) who have their loan payment due in the first 7 days of the month.*

Text

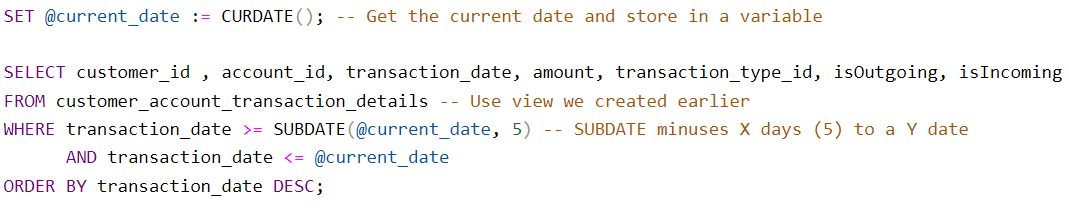
Description automatically generatedIn order to gather necessary data for this query we had to first identify what tables the data resided in, e.g. customer name, account number and loan payments laid in the customer, account, customer account (junction table) and the loan table. So I decided to use inner joins on the tables to select records that have matching values in both tables. I used joins over subqueries as there are quite a few tables we need data from and joins generally have faster execution speeds. A “WHERE” clause was used to filter down the data to just loans due in the first 7 days of a month.

Graphical user interface, application

Description automatically generated

1. *Extract all bank transactions that were made in the past 5 days (please include customer and account details).*

For this query task I had to retrieve first the current date in order to get transactions “made in the past 5 days”, this could be done using the SQL built in function “CURDATE()” which returns the current local date of the machine the query is executed on. I stored this date in a variable called @current\_date, so it is clearer for anyone viewing this query what it is used for. In the WHERE I then proceeded to use the “SUBDATE()” function which takes in a date value (@current\_date) and takes away X days which is given as the second argument (5) and returns the new date. I use this returned date and compare it with the transaction date for all transactions to check I only retrieve transaction in my select that are later or equal too then the minuses 5 days date. The WHERE clause is then finished off with a logical operator, “AND” that checks if the transaction date is also less than or equal to the current date. Finally a “ORDER BY” is by the transaction dates in descending order to make the results of the query tidier.





The above query didn’t give back any results, why this is the case is because as mentioned previously I used the CURDATE() function in the WHERE clause which gives me the current date. Therefore as off writing this report (20/01/2022) the previous 5 days of days of transactions would lay between the 20/01/2022 and 15/01/2022, however I did not insert any transactions into the transaction table between these dates, due to creating the database and data before January 2022, therefore no data should be expected to be returned by the query.

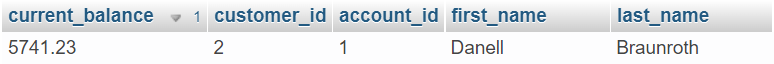
1. *List the customers with their current balance greater than £5000 (at least 1 customer should have their current balance greater than £5000). The current balance can be calculated by summing the opening balance of the account, all the incoming transactions of the account, and deducting the outgoing transactions of the account.*

This query involved the use of view created earlier as it includes all necessary data, including the “opening balance” of the accounts, all the “incoming transactions” and “outgoing transactions”.

In the select statement we are adding together the opening balance of each bank account to the sum of total transactions incoming and then taking away the sum of all transactions outgoing, this gets us the current balance. “IF” statements are used within the select statement to verify if a row is incoming or outgoing if it is, use the amount given in that row for the SUM otherwise use 0. We want to do this for each account so we do a “GROUP BY” account\_id, so the transactions outgoing, and ingoing are grouped by specific accounts, otherwise without the group by clause SQL would not know what incoming transactions and outgoing transaction to sum together. You can think of group by clause as a “for each”, e.g. for each account (account\_id) sum incoming transactions. Finally we have a “HAVING” clause in order to filter out the current account balances greater than the amount inputted as the argument for the stored procedure in this case 5000. A having clause is required instead of “WHERE” clause, due to us wanted to filter by  rows in the result set representing groups, however where clause only applies to individual rows.

Graphical user interface, text, application

Description automatically generated with medium confidence



1. *The Bank management team often needs to know the "Total Outstandings" of the bank. The Total Outstandings is the up-to-date balance over all bank accounts and includes the sum of the opening balance of every bank account deducting all outgoing payments and adding all incoming payments of every bank account.*

This query was performed using subqueries in order to retrieve the correct columns needed. First we get the sum of all opening balances from account table, add the sum of all incoming transactions (by a WHERE clause check to seeing if a transaction is “isIncoming”, we don’t need a “= true” check as it is redundant).

Graphical user interface, text, application

Description automatically generatedAdditionally we take away the sum of all outgoing transactions (“isOutgoing”) and give the result an alias name of “total\_outstanding”.

Graphical user interface

Description automatically generated with low confidence

Stored procedures

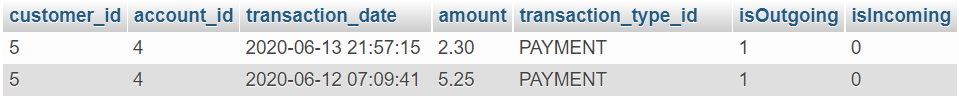
1st stored procedure

This is the first query I decided to create as a procedure as it would be a query that could be often executed by the database admin (finding transactions in past 5 days).

The query starts by defining a DELIMITER to tell the MySQL client to treat the statements, functions, stored procedures, or triggers as an entire statement. Normally in a .sql file you set a different DELIMITER like $$. The DELIMITER command is used to change the standard delimiter of MySQL commands (;). As the statements within the routines (functions, stored procedures, or triggers) end with a semi-colon (;), to treat them as a compound statement. If not defined when using different routines in the same file or command line, it will give syntax error. The procedure has 1 input value of type “DATE”, this input is used in the where clause to select from what date you want to find the past 5 days of transactions. Therefore you can manually choose what date you want to find past transactions from rather then the current date used in our original query. Below we test the procedure using “2020-06-14” as this is a date I know I have inserted transactions for.

Graphical user interface, text, application, email

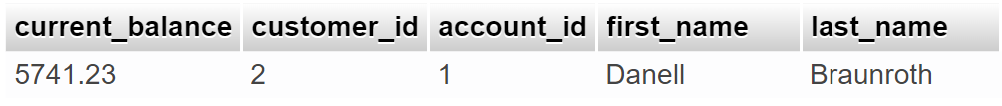
Description automatically generated



2nd stored procedure

This is the second query I decided to construct as a stored procedure (finding customers with a balance greater than a specific amount).

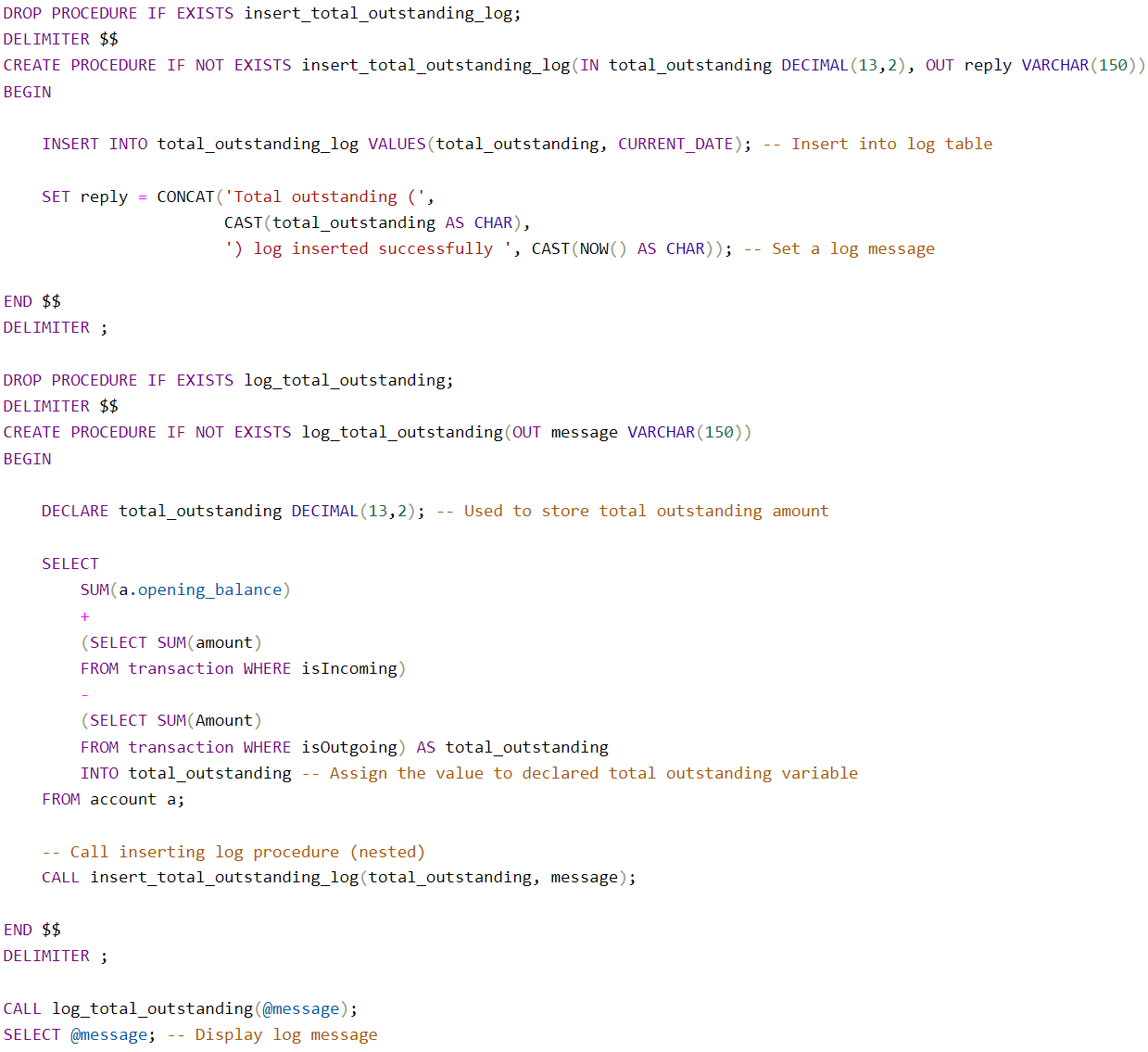
The input for the procedure is the amount you want to find for customers with a balance greater then (for this scenario I’m passing in 5000, as asked by the requirements mentioned in the assignment brief).

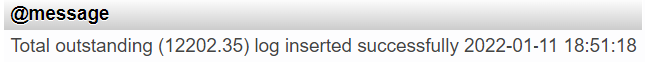
Graphical user interface, text, application, email

Description automatically generated

3rd stored procedure

I decided to include a 3rd procedure that wasn’t required by the brief in order demonstrate my understanding of more complex stored procedure layouts (nested stored procedure). 2 procedures were made in order to able to effectively log total outstanding amounts to a separate log table. The reason this was made is in a banking application it could be a useful feature to allow the database administrator to log total outstanding amounts along with a timestamp, this data could then be used later for the banks analytics (example below).

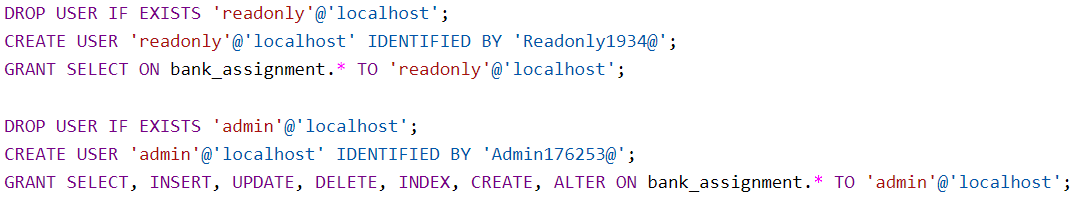




First a log table was made that had 2 columns 1 to log the total outstanding amount and another for the time stamp. The first procedure enables us when called to insert directly into the log table this procedures accepts 2 parameters, 1 input (amount) and one out parameter (log message reply). The 2nd procedure created is the main one that should be called by who ever wishes to log the total outstanding amount, this procedure accepts 1 parameter which is an out parameter for log message reply. This procedure will calculate the current total outstanding and then pass the value into a variable called ‘total\_outstanding’. Next a nested called to the first stored procedure is made which then inserts the log into the log table and this procedure replies back through an out parameter which was passed in by the 2nd procedure. We then select the message which displays the log message including the total outstanding amount and the timestamp.

***Security & Scalability***

When implementing any database, whether it is a relational or non-relational, it is always important to consider security and scalability. These are the two main concepts that can effectively “make or break” your database.

For this database design it could be a beneficial idea to create 2 basic users roles (can add further roles if needed) a read and write user (an admin) and a read only user (refer to example below).

This would help stop those who shouldn’t need full access to the database the ability to read from the tables, however, prevent the ability to make any major changes such an insert data, deleting data or dropping tables. This sort of user access can be left accessible only to the correct personnel, such as the database administrator through the read and write role. It is also important to involve the use of secure backups and transactions in order for the database to be rolled back in case of an outage or security attack, helping recover vital data such as customer details, transactions etc.

SQL injection are a common widespread issue on all types of relational databases. These SQL injections are “a common attack vector that uses malicious SQL code for backend database manipulation to access information that was not intended to be displayed.” (Learning Center, n.d.). For instance a banks website will have forms for customer to fill out (username, email, password etc), when accessing their online account, there forms are perfect for attackers to inject short SQL code. These SQL code snippets will usually contain characters such as “ or ; in order to trick an unsecure banks database. Common ways to prevent these attacks are to sanitise user inputs on the banks website using well tested libraries that can easily spot suspicious user input, and also using stored procedures allow statements to be automatically parameterized to prevent SQL attacks. However, most of the SQL injection prevention will occur on the backend for any online forms provided by the bank.

Moving onto scalability, one of the vital methods to scale this database implementation further is by considering moving the storage of the database to a cloud service (Azure, AWS or Google cloud), rather than on premise. This allows for much better hardware scalability to allow your database to easily expand to suit the demand of the banks growth, this is known as vertical scaling. Furthermore, moving to the cloud allows for greater data redundancy and uptime, as your database can be easily replicated in case one of your virtual machines storing the companies data goes down. This is especially important for our banking scenario. Additionally, to improve this database, it is worth incorporating additional tables, such as an employee table, allowing the bank to store much wider range of data, rather than pure customer focussed; also this would match closer with real world banking.

***Conclusion***

In summary a successful database for a banking system has been implemented, that can support a large array of customers. SQL queries as stated by the requirements have been crafted, analysed, and tested to see they give the desired outputs while minimising query time. Finally considerations for scalability and security have been discussed in depth, with given examples for specific ways to implement these 2 topics.

***References***

support.microsoft.com. (n.d.). *Database design basics*. [online] Available at: <https://support.microsoft.com/en-us/office/database-design-basics-eb2159cf-1e30-401a-8084-bd4f9c9ca1f5>.

Fathulla, K. (2021). *The relational model*. [online] blackbaord.lincoln.ac.uk. Available at: https://blackboard.lincoln.ac.uk/ultra/courses/\_167949\_1/cl/outline [Accessed 18 Oct. 2021].

‌ MaryQiu1987 (2020). *Database normalization description - Office*. [online] docs.microsoft.com. Available at: <https://docs.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description>.

‌ www.gleek.io. (n.d.). *Crow’s foot notation in entity-relationship diagrams*. [online] Available at: <https://www.gleek.io/blog/crows-foot-notation.html>.

GeeksforGeeks. (2019). *Second Normal Form (2NF) - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/second-normal-form-2nf/>.

Learning Center. (n.d.). *What is SQL Injection | SQLI Attack Example & Prevention Methods | Imperva*. [online] Available at: https://www.imperva.com/learn/application-security/sql-injection-sqli/#:~:text=SQL%20injection%2C%20also%20known%20as.

‌