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

Text

Description automatically generated

*Introduction*

This is a report to help dive further into the process I took when choosing the overall design and implementation of the relational database (MYSQL) for a scenario bank. I will begin by talking about the general design process then move onto the variety of queries written on the database, finally touching on scalability and security concepts.

*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 for 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.

The structure of the tables was where most of the time was spent. Using principles instructed to us 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), made is similar 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 relation 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.

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 in to 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.

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 a 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

Text, Word

Description automatically generatedQueries performed

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

*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. Firstly I will mention how security was strongly considered in my database.

For this database I decided to create 2 basic users roles for now a read and write user (an admin) and a read only user. This would help stop those who shouldn’t need full access to the database the ability to read from the tables etc, however not make any major changes such a insert data, deleting data or dropping tables. This sort of user access can be left to the database administrator through the read and write role. Once this database scales further keywords such as “GRANT”, “DENY” and “REVOKE” can be use to control and manage user permissions to keep the database accessible only to the correct personnel. 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 a outage or security attack, helping recover vital data such as customer details, transactions etc.

SQL injection are a common wide spread 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 there 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 first way I could scale my database implementation further is by considering to move the storage of the database to a cloud service, rather then 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. Furthermore, to scale my database I could incorporate additional tables, such as an employee table, allowing the bank to store much wider range of data, rather than pure customer focussed.

*Conclusion*

In summary for the brief given I have implemented a successful database for a banking system, 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. Finally considerations for scalability and security have been discussed in depth, with given examples for specific ways to implement theses 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.

‌

‌

‌

‌