Database and SQL, Final Assignment

**DBMS** (Database Management System) is a set of software and linguistic tools for general or special purposes that manage to creation and use of database.

Data **redundancy** occurs when the same piece of data exists in multiple places, whereas data inconsistency is when the same data exists in different formats in multiple tables. Unfortunately, data redundancy can cause data inconsistency, which can provide a company with unreliable and meaningless information.

**Normalization** is a database design technique that reduces data redundancy and eliminates undesirable characteristics like Insertion, Update and Deletion Anomalies. Normalization rules divides larger tables into smaller tables and links them using relationships. The purpose of Normalisation in SQL is to eliminate redundant (repetitive) data and ensure data is stored logically.

There are a few normal forms, but the first 3 (1FN, 2FN, 3FN) are most useful.

1NF: A relation is in 1NF if all its attributes have an atomic value.

2NF: A relation is in 2NF if it is in 1NF, and all non-key attributes are fully functional dependent on the candidate key.

3NF: A relation is in 3NF if it is in 2NF and there is no transitive dependency.

**SQL and NoSQL** differ in whether they are relational (SQL) or non-relational (NoSQL), whether their schemas are predefined or dynamic, how they scale, the type of data they include and whether they are more fit for multi-row transactions or unstructured data.

The main advantages of SQL databases are:

* Adjusted, standardised and affordable.
* Availability of more SQL database-oriented product suites and integrations.
* Existing for a longer time, having a larger community and more support systems behind them.
* Operational atomicity.
* Easy rollbacks and setups.
* Compatibility when working with the right data types and information.
* Efficient with complex queries and ongoing query operations.

The main disadvantages of SQL databases are:

* Not efficient with large volumes of data.
* Database execution can sometimes be challenging.
* Complicated to use and SQL is more complex than modern document-oriented methods.
* Not scalable enough, especially horizontally.
* Lack of decentralisation and flexibility.
* Only effective with structured data.
* Requires more powerful hardware and better IT infrastructure.
* Slow compared to modern NoSQL-based systems.
* Not great for huge user loads or multiple concurrent operations.

The main advantages of NoSQL databases are:

* Decentralisation and adaptability.
* Flexible and easy to integrate with applications or development pipelines.
* More accessible and easy to use as a database.
* Adjustable according to operational requirements with minimal data movement.
* Enormous scalability and efficiency when working with big data or massive volumes of data.
* Supports structured, unstructured, and semi-structured data.
* Can easily work on weaker hardware and simple IT architecture.
* Supports a massive number of concurrent users.

The main disadvantages of NoSQL databases are:

* Not effective with maintaining atomicity or accuracy, decreasing the trustworthiness of the information.
* NoSQL databases are more unorganised and can face attribution problems.
* Lack of standardisation.
* Lack of smooth transmutation across other NoSQL databases. This makes it extremely tough to change one’s setup or move to another database system.
* Less support and fewer available frameworks to ensure a stable future for databases working on non-Linux systems.

**JSON** (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

**XML** has a variety of uses for Web, e-business, and portable applications. For example: Web publishing - XML allows you to create interactive pages, allows the customer to customize those pages, and makes creating e-commerce applications more intuitive. PDF files, PostScript files, Microsoft Word documents, PowerPoint documents, and RTF text files are also stored as XML.

**Data integrity** is the overall accuracy, completeness, and consistency of data. Data integrity also refers to the safety of data regarding regulatory compliance — such as GDPR compliance — and security. It is maintained by a collection of processes, rules, and standards implemented during the design phase. When the integrity of data is secure, the information stored in a database will remain complete, accurate, and reliable no matter how long it’s stored or how often it’s accessed.

**SQL Server** is a database server by Microsoft. The Microsoft relational database management system is a software product which primarily stores and retrieves data requested by other applications. These applications may run on the same or a different computer.

**Three primary components** make up SQL Server architecture: Protocol Layer, Relational Engine, and Storage Engine.

**Tables** are database objects that contain all the data in a database. In tables, data is logically organized in a row-and-column format like a spreadsheet. Each row represents a unique record, and each column represents a field in the record.

**A view** contains no data of its own, but it is like a ‘window’ through which data from tables can be viewed or changed. A view is a query of one or more tables that provides another way of presenting the information. Views do not actually store data rather they derive their data from tables on which they are based, referred to as the base table of the view. Creating a view fulfils the requirement without storing a separate copy of the data because a view does not store any data of its own and always takes the data from a base table.

Using **logic in the database** is acceptable to speed up the work with the database, for example, reducing the number of variables checked when searching. Or, to reduce the amount of data retrieved from the database, for example, to work with the phone numbers of a client, there is no need to extract all the phone numbers of this client for the entire time of using this database, only the most recent one is needed.

The use of **logic in applications** is necessary for the correct use of databases, entering data into them in the correct format, extracting the necessary and not redundant amount of data, working with data without changing or distortion the database.

Bank

Bank\_ID (PK)

Bank\_name

SWIFT

Address

Acredditation\_in\_Sweden

Account

Account\_ID (PK)

Bank\_ID (FK)

Client\_ID (FK)

Currency\_ID (FK)

Account\_open\_date

Account\_close\_date

Transaction\_account

Account\_ID\_from (FK)

Account\_ID\_to (FK)

Transaction\_date

Transaction\_value

Client

Client\_ID (PK)

Client\_name

Client\_surmane

Client\_telefone\_number

Client\_email

Exchange\_rate

Currency\_ID\_from (FK)

Currency\_ID\_to (FK)

Exchange\_rate\_value

Currency

Currency\_ID (PK)

Currency\_name

The project consists of the 6 tables and simulate a simple Banking system.

The table Bank consists of the 5 variables: Bank\_ID (primary key), Bank\_name, SWIFT (unique bank number), Address and Accreditation in Sweden.

The table Client consists of the 6 variables as: Client\_ID (primary key), Client\_name, Client\_surname, Client\_telefone\_number and Client\_email.

The table Currency consists of the 2 variables: Currency\_ID (primary key) and Currency\_name.

The table Exchange rate consists of the 3 variables:

* Currency\_ID\_from (foreign key from Currency - Currency\_ ID) this variable is indicated to which currency ID comes from the operation;
* Currency\_ID\_to (foreign key from Currency - Currency\_ ID) this variable is indicated which currency ID had the operation;
* Exchange rate value

The table Account consists of the 6 values:

* Account\_ID (primary key)
* Bank\_ID (foreign key from Bank - Bank\_ID)
* Client\_ID (foreign key from Client – Client\_ID)
* Currency\_ID (foreign key from Currency – Currency\_ID)
* Account open date (the date account was open)
* Account close date (the date account was close)

The table Transaction\_Account consists of the 4 variables:

* Account\_ID\_from (foreign key from Account – Account\_ID), variable is indicate where did the operation comes from
* Account\_ID\_to ( foreign key from Account – Account\_ID), variable is indicate where the operation go
* Transaction\_date
* Transaction\_value

First, I created a database, then 6 tables. After that I insert the information into the tables. Every table has at least 5 rows. Totally it is 5 banks, 5 clients, 5 types of currency, 15 rows of exchange rate variations, 9 rows with accounts – because a person can have a few accounts in different banks or accounts can have a different currency. And 14 rows in Transactions - I choose 3 clients and their accounts in different banks/currency.

As main point in the project I choose the account 201041 and want to calculate its balance. First, I need to select all the operations with this account. Second to indicate which of them come to the account and which go from. Last, I need to calculate the sum of the values.

As manipulate the database. Here I write some examples of my code to explain how I used different functions (for more detailed information write a code comments)

**The CRUD – CREATE, INSERT, UPDATE and DELETE**

CREATE DATABASE "FA";

INSERT INTO Bank (Bank\_name, SWIFT, Adress, Accreditation\_in\_Sweden)

UPDATE Client SET Client\_name = 'Oscar', Client\_surname = 'Ek', Client\_telefone\_number = '0723465321' WHERE Client\_id = 4;

DELETE FROM Client WHERE Client\_ID = 4;

**Nested Queries**

DECLARE @Account INT = 201041;

WITH Account\_Balance (Account\_ID, Balance) AS (

SELECT @Account AS Account\_ID, SUM (CASE WHEN Account\_ID\_from = @Account THEN -Transaction\_value ELSE Transaction\_value END) AS Balance

FROM Transaction\_Account

WHERE Account\_ID\_from = @Account OR Account\_ID\_to = @Account

)

SELECT t3.Bank\_name, (t4.Client\_surname + ' ' + t4.Client\_name) as full\_name, t1.Account\_ID, t5.Currency\_name, t2.Balance

FROM Account t1

JOIN Account\_Balance t2 ON t1.Account\_ID = t2.Account\_ID

JOIN Bank t3 ON t1.Bank\_ID = t3.Bank\_ID

JOIN Client t4 ON t1.Client\_ID = t4.Client\_ID

JOIN Currency t5 ON t1.Currency\_ID = t5.Currency\_ID;

**Math functions:**

SELECT SUM (Transaction\_value) FROM Transaction\_Account WHERE Account\_ID\_to = 201041;

SELECT MIN (Transaction\_value) FROM Transaction\_Account;

**Join**

SELECT

t2.Bank\_name, (t3.Client\_surname + ' ' + t3.Client\_name) as full\_name, t1.Account\_ID, t4.Currency\_name

FROM

Account t1

JOIN Bank t2 ON t1.Bank\_ID = t2.Bank\_ID

JOIN Client t3 ON t1.Client\_ID = t3.Client\_ID

JOIN Currency t4 ON t1.Currency\_ID = t4.Currency\_ID

ORDER BY t2.Bank\_name, full\_name;

**Order by and Range**

SELECT Transaction\_date, (CASE WHEN Account\_ID\_from = 201041 THEN -Transaction\_value ELSE Transaction\_Value END) AS Val,

SUM(CASE WHEN Account\_ID\_from = 201041 THEN -Transaction\_value ELSE Transaction\_Value END) OVER (

ORDER BY Transaction\_date

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS balance

FROM Transaction\_Account

WHERE Account\_ID\_from = 201041 OR Account\_ID\_to = 201041;

**Group by**

DECLARE @Cur\_Name CHAR(3) = 'USD';

SELECT t2.Currency\_name, @Cur\_Name AS To\_Currency, AVG (t1.Excange\_rate\_value) AS Average\_rate

FROM Excange\_rate t1

JOIN Currency t2 ON t1.Currency\_ID\_from = t2.Currency\_ID

WHERE t1.Currency\_ID\_to = (SELECT Currency\_ID FROM Currency WHERE Currency\_name = @Cur\_Name)

GROUP BY t2.Currency\_name;

**Top**

SELECT TOP 5 \* FROM Excange\_rate WHERE Currency\_ID\_from <> 1226;

**@variabler**

DECLARE @Variable1 INT = 80;

DECLARE @Variable2 INT = 500;

SELECT \* FROM Excange\_rate WHERE Excange\_rate\_value BETWEEN @Variable1 AND @Variable2;