Introduction to Databases TBA
May 2023

Miguel Angel Vinas

Student Number: x22116133

Index

1. Data Normalisation.
2. Data Management.
3. Structured Query Language.
4. Non-Relational databases.

1. Data Normalization

Review the term 'Data Normalization' in relation to database design and the use of Indexing. Your assessment must include detailed reasoning and list the relevant advantages to adopting this systematic process.

Data normalization is the process of mapping the entities in the conceptual design to tables in the logical design.

The way we organize the data in a database is by breaking it down into smaller, more manageable pieces and then linking them together. This process reduces data redundancy, ensures data consistency, and improves data integrity.

Normalization is typically divided into several stages, called normal forms, each with specific rules and guidelines to follow. The most commonly used normal forms are first normal form (1NF), second normal form (2NF), and third normal form (3NF).

- First Normal Form: Eliminating Repeating Data

First Normal Form (1NF) requires to eliminate all repeating data in columns, create separate rows for each group of related data and identify each record with a primary key.

In other words, First normal form involves making sure that each table in the database has a primary key, and that each column in the table contains only atomic (indivisible) values.

- Second Normal Form: Eliminating Partial Dependencies

Second Normal Form (2NF) requirements state that we must place subsets of data in multiple rows in separate tables. Second Normal Form also requires to create a relationship with the original table by creating a foreign key.

Put simply, second normal form means that we want to make sure that all the information in our table is properly organized to avoid any unnecessary duplication.

- Third Normal Form: Eliminating Transitive Dependencies

Third Normal Form (3NF) requires that each non-key attribute depends only on the primary key and not on any other non-key attribute in the table.

In a more simple way, In 3NF, each attribute (data field) in a table depends only on the primary key and nothing else.

Third normal form will cover well over 90 percent of the cases needed in business information systems, it's considered the "gold standard".

- Other normal forms beyond Third Normal Form

There are some other normal forms worth knowing: Boyce-Codd Normal Form (BCNF), Fourth Normal Form and Fifth Normal Form.

Adopting a normalization process in database design has several advantages:

1) It helps to eliminate data redundancy, which reduces storage requirements and enhances data consistency.

2) It improves data integrity by ensuring that each piece of data is stored in only one location.

3) Normalization simplifies database maintenance by reducing the complexity of queries and reducing the likelihood of data inconsistencies.

Normalization also facilitates the process of indexing, which enhances database performance by making it easier to manage; it also allows for the user to query the data effectively.

We use indexes to expedite the process of searching for a particular row of information. By using an internal filing system, SQL can jump to the approximate location of our data much more quickly. It does this through the use of indexes, also known as keys. SQL requires at least one index on every table, so that it has something to go by. Normally, we would use a primary key, or unique identifier that helps keep the data separate.

Normalization and indexing are closely related because normalization ensures that data is structured in a way that makes indexing more efficient. When data is normalized, it is easier to create indexes that accurately reflect the relationships between data elements.

References:

Oppel, A. J. (2009). Databases: A Beginner's Guide. McGraw Hill.

Naramore, E., Gerner, J., Le Scouarnec, Y., Stolz, J., K. Glass, Michael. (2005). Beginning PHP5, Apache, and MySQL® Web Development

Date, C. J. (2004). An introduction to database systems (8th Edition). Pearson Education.

Elmasri, R., & Navathe, S. B. (2016). Fundamentals of database systems (7th Edition). Pearson.

2. Data Management

Critically evaluate the use of 'Data Warehousing' and 'Data Mining' within the context of Data Management. Examine how this framework is employed across industry and detail its purpose in relation to data analysis. Compare and contrast OLTP versus Data Warehousing.

Data Warehousing is the process of storing and maintaining analytical data from various sources separately from transaction – oriented databases for the purpose of decision support. Data Warehousing tends to keep years' worth of data in order to enable analysis of historical data and provide decision makers, from middle management upward, with the right information to support decision making.

Data Mining, on the other hand, refers to the process of analysing (or mining) vast amounts of data to discover patterns, rules, trends, and relationships that are not readily apparent. Although some data mining features are being provided in Database Management Systems, data mining is not well integrated with them, and the field of data mining is popularly called business intelligence or data analytics.

The goal of a data warehouse is to support decision making with data by providing a centralized repository of data. Data mining can be used in conjunction with a data warehouse to help with certain types of decisions. Data mining helps in extracting meaningful new patterns that cannot necessarily be found by merely querying or processing data or meta-data in the data warehouse. Also, data mining tools should be designed to facilitate their use in conjunction with data warehouses. In fact, for very large databases running into terabytes and even petabytes of data the successful use of data mining applications will depend first on the construction of a data warehouse.

This knowledge can be used for decision making, identifying new business opportunities.

Data Warehousing and Data Mining are widely used across various industries, including finance or healthcare. In finance, Data Warehousing and Data Mining are used to analyse the creditworthiness of clients, performance analysis of finance investments like stocks and fraud detection. In

healthcare, applications include analysis of side effects of drugs and effectiveness of certain treatments, optimization of processes within a hospital and analysis of the relationship between patient wellness data and doctor qualifications.

OLTP (Online Transaction Processing) is used for handling high volumes of transactions while Data Warehousing is used for data analysis.

OLTP systems are designed to support the day-to-day transactional activities of an organization, such as processing orders, payroll, billing or updating customer records. They are designed for fast, efficient, and reliable transaction processing, typically with a focus on data integrity and consistency.

In contrast, Data Warehousing is organized around major subject areas of an organization, such as sales, customers or products. Data Warehouses are designed for efficient data retrieval and analysis, typically with a focus on data consistency and completeness.

In conclusion, Data Warehousing and Data Mining are essential components of modern data management. They are widely used across various industries to support business intelligence activities such as data analysis and decision making and are employed based on the specific needs of an organization.

OLTP Systems	Data Warehouse Systems
Hold current data	Hold historic data
Store current data	Store detailed data along with lightly and highly summarized data
Data is dynamic	Data is static, except for periodic additions
Database queries are short-running and access relatively few rows of data	Database queries are long-running and access many rows of data
High transaction volume	Medium to low transaction volume
Repetitive processing; predictable usage pattern	Ad hoc and unstructured processing; unpredictable usage pattern
Transaction driven; support day-to-day operations	Analysis driven; support strategic decision making
Process oriented	Subject oriented
Serve a large number of concurrent users	Serve a relatively low number of managerial users (decision makers)

Databases: A Beginner's Guide – Chapter 12: Databases for Online Analytical Processing, page 356

References:

Oppel, A. J. (2009). Databases: A Beginner's Guide. McGraw Hill.

Pare, R. C., Santillan, L. A., Costa, D. C., Ginesta, M. G., Escofet, C. M., Mora, O. P., (2005). Software Libre – Bases de datos (1° Edicion). UOC, Universitat Oberta de Catalunya.

3. Structured Query Language

Suppose that you are assigned to implement the following logical database design into a physical database design in one of the hospitals in Ireland. The hospital uses MySQL server for implementation of the system. Currently, the hospital registers individuals with Covid cases. The types of cases include, confirmed, recovered, deceased and intensive care.

- *Database Name: CovidTracking*
- *Database Tables:*
- *Cases (caseId: int, caseType: String, caseDescription: String)*
- *Person (PersonId:int, PersonFname:string, PersonLname:String, Eircode: String) Status (PersonId, Date, caseId)*

Write the SQL statements (DDL) required to implement the database and its tables.

*Write SQL statements (DML) to add **two** patient case records across the 3 tables.*

*Write SQL statements to generate the daily tracking report for number of **confirmed**, **recovered,** and **deceased** and **intensive care cases**. Use the SUM and count functions.*

SQL statements

Comments will be in format

Italic

SQL statements will be in format

Statement

-- We are going to create a database with the name CovidTracking,

-- However, before creating it, we are going to make sure that the hospital does not have a database named like that,

-- So we can implement it.

```
DROP database IF EXISTS CovidTracking;

CREATE database CovidTracking;
```

-- We are going to create the tables for the database CovidTracking.

-- We are going to create a table called Cases with attributes called caseId, caseType, caseDescription. Each of those attributes

-- has a data type (Integer and Varchar (since we have to store a string)) and a value. All of them are NOT NULL because they must be filled in when creating the database.

-- As the caseTypes can only be four ("Confirmed", "Recovered", "Deceased" and "Intensive Care") I have decided to create a constraint to make sure that the end user can only INPUT values between 1 and 4.

-- The primary key is caseId and it will be auto incremented when the end user adds one record. There is no foreign key.

```
USE CovidTracking;

CREATE TABLE Cases

(

        caseId INTEGER (10) AUTO_INCREMENT PRIMARY KEY NOT NULL,

        caseType VARCHAR (255) NOT NULL CHECK (caseType BETWEEN 1 AND 4),

        caseDescription VARCHAR (255) NOT NULL

);
```

-- We are going to create a table called Person with attributes called PersonId, PersonFName, PersonLName and Eircode. Each of those attributes

-- has a data type (Integer and Varchar (since we have to store a string)) and a value. All of them are NOT NULL because they must be filled in when creating the database.

-- I have chosen to give the Eircode a lenght of 255 characters as the Eircode in the future might grow to more characters.

-- The primary key is PersonId and it will be auto incremented when the end user adds one record. There is no foreign key.

```
USE CovidTracking;

CREATE TABLE Person

(

        PersonId INTEGER (10) AUTO_INCREMENT PRIMARY KEY NOT NULL,

        PersonFname VARCHAR (255) NOT NULL,

        PersonLname VARCHAR (255) NOT NULL,

        Eircode VARCHAR (255) NOT NULL

);
```

-- We are going to create a table called Status with attributes called PersonId, Date and caseId. Each of those attributes

-- has a data type (Integer, Varchar (since we have to store a string) and DATETIME) and a value. All of them are NOT NULL because they must be filled in when creating the database.

-- The primary key is a compound Primary Key made out of two columns PersonId and Date.

-- I have chosen DATETIME as a datatype because we are going to use the ISO 8601 format when inserting data (YYYY-MM-DD HH:MM:SS) because it can include hours, minutes and seconds on top of the normal date.

-- Since a person can have several status (like CONFIRMED in one day and DECEASED in another one) the combination of the two columns, PersonID and Date creates

-- a unique identifier for each record in the Status Table.

-- So there cannot be two records with the same PersonID and Date values in the table!

-- When creating a compound Primary Key there is no need to add NOT NULL (You can't do that actually) because SQL already knows that a Primary Key is NOT NULL by definition!.

```
USE CovidTracking;

CREATE TABLE Status

(

        PersonId INTEGER (10) NOT NULL,

        Date DATETIME NOT NULL,

        caseId INTEGER (10) NOT NULL,

        PRIMARY KEY (PersonId, Date),

        FOREIGN KEY (caseId) REFERENCES Cases(caseId)

);
```

-- We are going to insert the following values for the people who go into the hospital into the Cases table.

-- caseType, caseDescription. ***We are not going to insert caseId (which should be the first value in the row) because the table will autoincrement it for us.***

```
INSERT INTO Cases (caseType, caseDescription) VALUES
("1", "Confirmed"),
("2", "Recovered"),
("3", "Intensive Care"),
("4", "Deceased"),
("3", "Intensive Care"),
("4", "Deceased"),
("2", "Recovered"),
("1", "Confirmed"),
("2", "Recovered"),
("4", "Deceased"),
("3", "Intensive Care"),
("4", "Deceased"),
("2", "Recovered"),
("1", "Confirmed"),
("1", "Confirmed"),
("1", "Confirmed"),
("2", "Recovered"),
("2", "Recovered"),
("4", "Deceased"),
("3", "Intensive Care");
```

-- We are going to insert the following values for the people who go into the hospital into the Person table.

-- PersonId, PersonFname, PersonLname, Eircode. ***We are not going to insert personId (which should be the first value in the row) because the table will autoincrement it for us***

```
INSERT INTO Person (PersonFname,PersonLname,Eircode) VALUES
 ("Miguel Angel","Vinas","K78HP08"),
 ("Wayne","Duncan","V2I4M71"),
 ("Colette","Britt","I8R5T7A"),
 ("Wanda","Mcgee","C7R5E8B"),
 ("Vera","Dyer","X9E4I9C"),
```

("Hiram","Baldwin","L1X1R3D"),

("Dillon","Walters","S0F6Q7E"),

("Bertha","Houston","W6H1Y8F"),

("Lev","Stuart","P4F6J2G"),

("Bruce","Hopper","S6B8P1H"),

("Brady","Olsen","N8O8Q6A"),

("Peter","Mckay","T4Z1G3B"),

("Leonard","Clements","O1N0S6D"),

("Heidi","Johnson","F5F5S1F"),

("Lev","Contreras","I3X5V7Q"),

("Irma","Dennis","R8G3J7L"),

("Dorothy","Britt","D1T9S3M"),

("Lila","Moody","Q1L8Y9N"),

("Marvin","Bailey","G1H8B4P"),

("Dean","Decker","O4Y6I5Q");

-- We are going to insert the following values for the status of the people who are in the hospital into the Status table.

-- PersonId, Date, caseId.

-- For the value Date we are going to use the ISO 8601 format (YYYY-MM-DD HH:MM:SS) because it can include hours, minutes and seconds on top of the normal date.

-- Using the ISO 8601 format will allow us to have more compatibility with different systems and have a really unique primary compound key.

INSERT INTO Status (PersonId, Date, caseId) VALUES

(1,"2023-04-19 01:16:15", 1),

(2,"2023-01-07 10:06:24", 2),

(3,"2023-05-06 02:34:09", 3),

(4,"2023-01-02 09:54:38", 4),

(5,"2023-04-26 06:47:59", 5),

(6,"2023-01-21 10:25:32", 6),

(7,"2023-01-17 04:50:13", 7),

(8,"2023-01-09 10:15:56", 8),

(9,"2023-03-28 01:43:41", 9),

(10,"2023-03-05 06:28:29", 10),

```
(11,"2023-01-14 04:39:40", 11),
(12,"2023-04-14 07:14:12", 12),
(13,"2023-03-24 07:31:18", 13),
(14,"2023-02-20 05:36:57", 14),
(15,"2023-02-13 06:48:44", 15),
(16,"2023-05-06 10:34:09", 16),
(17,"2023-01-04 10:59:38", 17),
(18,"2023-01-06 08:01:35", 18),
(19,"2023-02-13 01:40:47", 19),
(20,"2023-01-14 08:04:33", 20);
```

-- https://www.w3schools.com/sql/func_mysql_sum.asp

-- The SUM function is used to calculate the sum of values in a specified column.

-- The syntax is as follows: SELECT SUM(column_name) FROM table_name;

-- In this case we want to SELECT the columns caseType and caseDescription, we are going to COUNT all the values in them and give this column a name AS Daily_Count (or else it will appear like COUNT(*) in the query.

-- We want to SELECT those two columns FROM the Cases table.

-- We could assign an alias to each table but I still prefer to see the Table AND the column I am referring to.

-- Then we are going to SUM each of the CASES when the caseDescription = "Confirmed" or "Recovered" or "Intensive Care" or "Deceased".

-- If the caseDescription = "Confirmed" (etc) then we are going to add 1 to the count.

-- If the caseDescription does not equal "Confirmed" (etc) we are going to do nothing.

-- We will count of the "descriptions" and give them a name AS "Total Confirmed" (etc) or else the name of the column would be pretty hard to read. We could give them a total different name if needed.

-- And as we said before, we want to SELECT all the cases FROM the Cases table AND (INNER JOIN) the Status table.

-- We use ON to tell SQL the rows we want to look at from the TWO previous tables.

-- In this case we want to know the caseType and the caseDescription from each caseId.

-- We are JOINING the Cases and Status tables based on their caseId columns.

-- WHERE our date in the Status table is LIKE the date that we want to input.

-- Since it is a DATETIME and the DATETIME contains more characters than the YYYY-MM-DD we can use the % at the end of the characters.

-- The % is the wildcard in SQL so it will give us all the data available for (in this case) "2023-05-06".

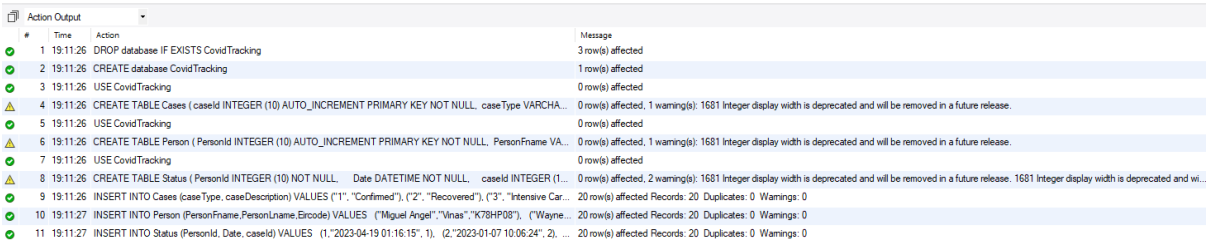
-- And finally we are going to group them by the caseType and the caseDescription.

```
USE CovidTracking;
SELECT Cases.caseType, Cases.caseDescription, COUNT(*) AS Daily_Count,
        SUM(CASE WHEN caseDescription = 'Confirmed' THEN 1 ELSE 0 END) AS "Total_Confirmed",
   SUM(CASE WHEN caseDescription = 'Recovered' THEN 1 ELSE 0 END) AS "Total_Recovered",
```

```
   SUM(CASE WHEN caseDescription = 'Intensive Care' THEN 1 ELSE 0 END) AS
"Total_Intensive_Care",

   SUM(CASE WHEN caseDescription = 'Deceased' THEN 1 ELSE 0 END) AS "Total_Deceased"

FROM Cases

INNER JOIN Status ON Cases.caseId = Status.caseId

WHERE Status.Date LIKE "2023-05-06%" -- We could also use WHERE Status.Date = CURDATE() and that will give us a report
based on the CURRENT DATE.

GROUP BY caseType, caseDescription;


-- IF we want to go a step further we could create a Store Procedure called Daily Report.

-- Where we will give the user the ability to enter a date range between the previous day and the current day.

-- That will create a report in a very user friendly way.

USE CovidTracking;

DELIMITER $

CREATE PROCEDURE Daily_Report (IN StartDate DATETIME, IN EndDate DATETIME)

BEGIN

        SELECT Cases.caseType, Cases.caseDescription, COUNT(*) AS Daily_Count,

                SUM(CASE WHEN caseDescription = 'Confirmed' THEN 1 ELSE 0 END) AS
        "Total_Confirmed",

                SUM(CASE WHEN caseDescription = 'Recovered' THEN 1 ELSE 0 END) AS
        "Total_Recovered",

                SUM(CASE WHEN caseDescription = 'Intensive Care' THEN 1 ELSE 0 END) AS
        "Total_Intensive_Care",

                SUM(CASE WHEN caseDescription = 'Deceased' THEN 1 ELSE 0 END) AS
        "Total_Deceased"

        FROM Cases

        INNER JOIN Status ON Cases.caseId = Status.caseId

        WHERE Status.Date BETWEEN StartDate AND EndDate

        GROUP BY caseType, caseDescription;

END $

DELIMITER ;


USE CovidTracking;

CALL Daily_Report ("2023-05-06", "2023-05-07");
```

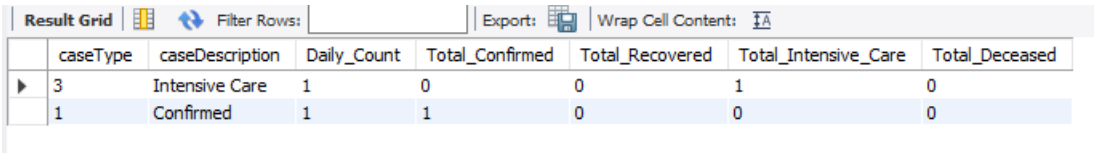
-- Or we could also create a view using the function CURDATE in the WHERE Status.Date = and that would give us a report based on the CURRENT DATE.

```
USE CovidTracking;

CREATE VIEW Daily_Report AS

SELECT Cases.caseType, Cases.caseDescription, COUNT(*) AS Daily_Count,

   SUM(CASE WHEN caseDescription = 'Confirmed' THEN 1 ELSE 0 END) AS "Total_Confirmed",

   SUM(CASE WHEN caseDescription = 'Recovered' THEN 1 ELSE 0 END) AS "Total_Recovered",

   SUM(CASE WHEN caseDescription = 'Intensive Care' THEN 1 ELSE 0 END) AS
"Total_Intensive_Care",

   SUM(CASE WHEN caseDescription = 'Deceased' THEN 1 ELSE 0 END) AS "Total_Deceased"

FROM Cases

INNER JOIN Status ON Cases.caseId = Status.caseId

WHERE Status.Date = CURDATE()

GROUP BY caseType, caseDescription;


SELECT * FROM Daily_Report;
```



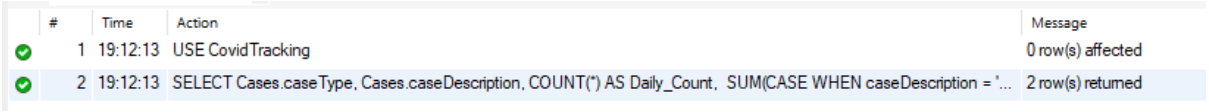
Action Output

#	Time	Action	Message
1	19:11:26	DROP database IF EXISTS CovidTracking	3 row(s) affected
2	19:11:26	CREATE database CovidTracking	1 row(s) affected
3	19:11:26	USE CovidTracking	0 row(s) affected
4	19:11:26	CREATE TABLE Cases (caseId INTEGER (10) AUTO_INCREMENT PRIMARY KEY NOT NULL, caseType VARCHA...	0 row(s) affected, 1 warning(s): 1681 Integer display width is deprecated and will be removed in a future release.
5	19:11:26	USE CovidTracking	0 row(s) affected
6	19:11:26	CREATE TABLE Person (PersonId INTEGER (10) AUTO_INCREMENT PRIMARY KEY NOT NULL, PersonFname VA...	0 row(s) affected, 1 warning(s): 1681 Integer display width is deprecated and will be removed in a future release.
7	19:11:26	USE CovidTracking	0 row(s) affected
8	19:11:26	CREATE TABLE Status (PersonId INTEGER (10) NOT NULL, Date DATETIME NOT NULL, caseId INTEGER (1...	0 row(s) affected, 2 warning(s): 1681 Integer display width is deprecated and will be removed in a future release. 1681 Integer display width is deprecated and wi...
9	19:11:26	INSERT INTO Cases (caseType, caseDescription) VALUES ("1", "Confirmed"), ("2", "Recovered"), ("3", "Intensive Car...	20 row(s) affected Records: 20 Duplicates: 0 Warnings: 0
10	19:11:27	INSERT INTO Person (PersonFname,PersonLname,Eircode) VALUES ("Miguel Angel","Vinas","K78HP08"), ("Wayne...	20 row(s) affected Records: 20 Duplicates: 0 Warnings: 0
11	19:11:27	INSERT INTO Status (PersonId, Date, caseId) VALUES (1,"2023-04-19 01:16:15", 1), (2,"2023-01-07 10:06:24", 2), ...	20 row(s) affected Records: 20 Duplicates: 0 Warnings: 0



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

caseType	caseDescription	Daily_Count	Total_Confirmed	Total_Recovered	Total_Intensive_Care	Total_Deceased
3	Intensive Care	1	0	0	1	0
1	Confirmed	1	1	0	0	0



#	Time	Action	Message
1	19:12:13	USE CovidTracking	0 row(s) affected
2	19:12:13	SELECT Cases.caseType, Cases.caseDescription, COUNT(*) AS Daily_Count, SUM(CASE WHEN caseDescription = '...	2 row(s) returned

References:

SQL SUM. (n.d). Retrieved from https://www.w3schools.com/sql/func_mysql_sum.asp (Accessed: 13 May 2023)

D. Wdzieczna (n.d), SQL SUM() Function Explained with 5 Practical Examples. Retrieved from https://learnsql.com/blog/sql-sum-function-explained/ (Accessed: 13 May 2023)

SQL CLAUSES. (23 July 2021). Retrieved from https://www.javatpoint.com/sql-clauses (Accessed: 13 May 2023)

Oracle SUM. (2022). Retrieved from OracleTutorial.com on https://www.oracletutorial.com/oracle-aggregate-functions/oracle-sum/ (Accessed: 13 May 2023)

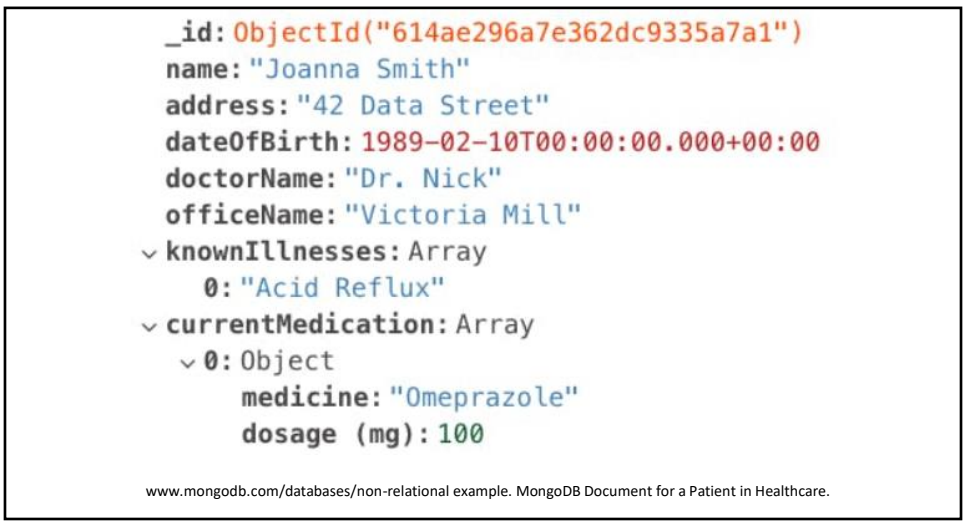
4. Non-Relational databases

A Non-Relational database provides a mechanism for storage and retrieval of data that is modelled in means other than the tabular relations used in relational databases. Critically evaluate the reasoning behind the adoption of such a model, discuss the details related to the use of ACID properties and key research behind this specific data model. Discuss the four types of non-relational databases along with the scenarios suitable to use and avoid them.

A non-relational database stores data in a non-tabular form, and tends to be more flexible than the traditional, relational database structures.

It does not follow the relational model provided by traditional relational database management systems. Instead, non-relational databases might be based on data structures like documents.

A document can be highly detailed while containing a range of different types of information in different formats. This ability to digest and organize various types of information side by side makes non-relational databases much more flexible than relational databases.



www.mongodb.com/databases/non-relational example. MongoDB Document for a Patient in Healthcare.

The adoption of Non-Relational databases has been driven by the need to handle large quantities of complex and diverse data need that needs to be organized, such as social media data, which are not easily represented in a tabular format.

Non-relational databases often perform faster because a query doesn't have to view several tables in order to deliver an answer, as relational datasets often do. Non-relational databases are therefore ideal for storing data that may be changed frequently or for applications that handle many different kinds of data. They can support rapidly developing applications requiring a dynamic database able to change quickly and to accommodate large amounts of complex, unstructured data.

There are several advantages to using non-relational databases, including:

1) Massive dataset organization: Non-relational databases can not only store massive quantities of information, but they also query these datasets with ease.

2) Flexible database expansion: As more information is collected, a non-relational database can absorb these new data points even if they don't fit the data types of the previously existing information.

3) Multiple data structures: Non-relational databases can group and query different information types together in the same document.

4) Built for the cloud: Non-relational databases can grow exponentially and they need a hosting environment that can grow with them.

Some Non-Relational databases are typically designed to support ACID (Atomicity, Consistency, Isolation, and Durability) properties, which ensure data integrity and consistency. However, the philosophy behind Non-Relational databases goes against the strict ACID rules. As a result a new database model called BASE (Basically Available, Soft State, Eventually Consistent) was designed, reflecting the properties of Non-Relational databases.

There are four main types of Non-Relational databases:

1) Document databases: These databases store data in documents, typically in JSON or XML format. Document databases are popular with developers because they have the flexibility to rework their document structures as needed to suit their application, shaping their data structures as their application requirements change over time. Document-oriented databases are used in scenarios such as e-commerce or trading platforms.

2) Key-value databases: This is the most basic type of database, where information is stored in two parts: key and value. The key is then used to retrieve the information from the database.

The simplicity of a key-value database is also an advantage. Because everything is stored as a unique key and a value that is either the data or a location for the data, reading and writing will always be fast.

However, this simplicity restricts the type of use cases it can be used for. More complex data requirements can't be supported.

Key-value databases are used in scenarios such as shopping carts or user profiles.

3) Column-oriented databases: These type of databases store and organize data as a set of columns rather than rows, columns are often of the same type and benefit from more efficient compression, making reads even faster.

Column-oriented databases are used in scenarios such as data warehousing or content management.

4) Graph databases: These type of databases are the most specialized of the non-relational database types, they store data as nodes and edges, which contain attributes, making them suitable for use cases where data has a complex relationship structure.

Graph databases are used in scenarios such as social networks or gps map systems.

Suitable scenarios for the use of Non-Relational databases include large-scale data processing, real-time data analysis, and high scalability requirements. However, Non-Relational databases may not

be suitable for scenarios where data consistency is critical, such as finance, accounting, enterprise resource planning, or where data relationships are simple and well-defined.

In conclusion, Non-Relational databases provide high performance, availability and scalability and they can handle a variety of data types and structures. They are designed to support ACID properties but may prioritize availability and partition tolerance over immediate consistency.

References:

MongoDB. (n.d.). Retrieved from https://www.mongodb.com/databases/non-relational (Accessed: 13th May, 2023)

MongoDB. (n.d.). Retrieved from https://www.mongodb.com/compare/relational-vs-non-relational-databases (Accessed: 13th May, 2023)

IBM (n.d.). Retrieved from https://www.ibm.com/topics/nosql-databases (Accessed: 14th May, 2023)

RavenDB (n.d.). Retrieved from https://ravendb.net/articles/acid-transactions-in-no-sql-ravendb-vs-mongodb (Accessed: 14th May, 2023)

ACID vs. BASE: Comparison of Database Transaction Models, PhoenixNAP (n.d.). Retrieved from https://phoenixnap.com/kb/acid-vs-base (Accessed: 15th May, 2023)