Database exam Raport

Database Exam summer 2024

Samuel Tobias Roland Uyet

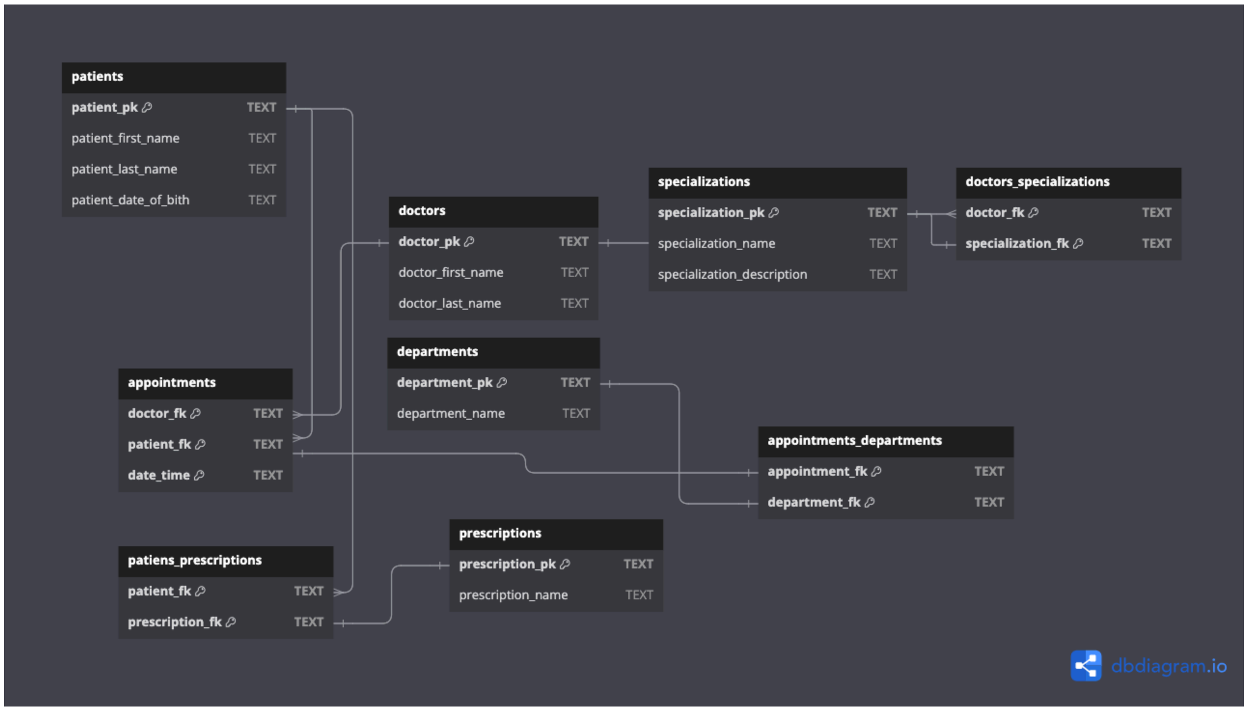
Database planning:

Before we can actually create the database we need to get an overview of what it's going to be about, and what kinds of tables we are going to have.
In this project we are going to be relating to a system for a hospital, where we will have tables with the following subjects:

1. Patients
2. Doctors
3. Appointments
4. Departments
5. Appointments_departments
6. Prescriptions
7. patients_prescriptions
8. specializations
9. doctors_specializations

Database design and the concept of Normalization

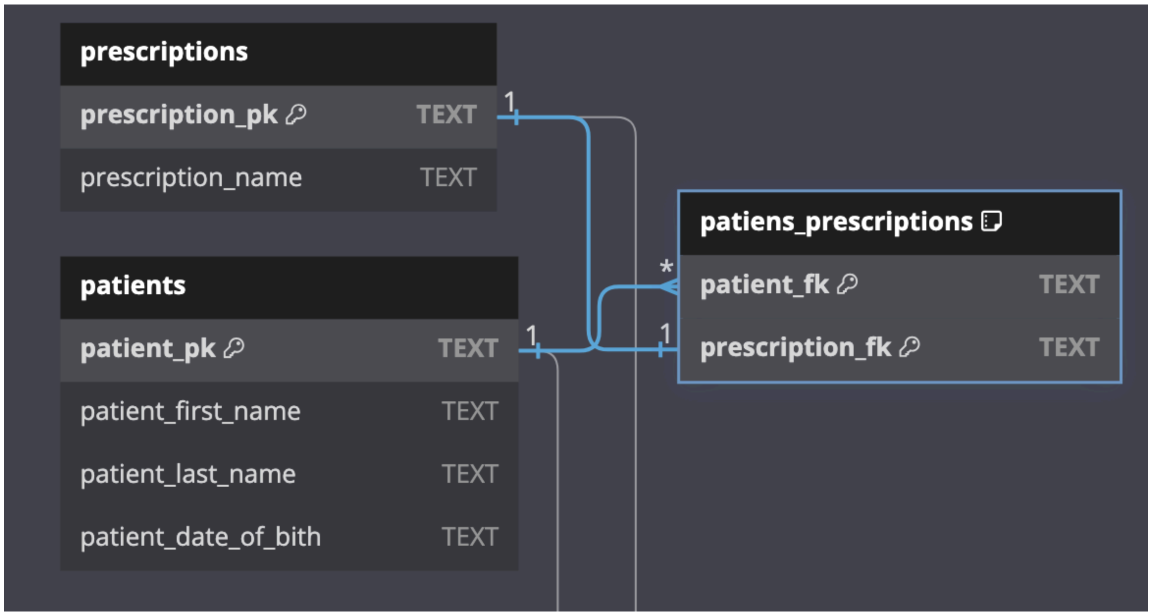
Database overview



This is a database overview for the hospital system mentioned above.
When designing the database it was important to apply the rules of normalization

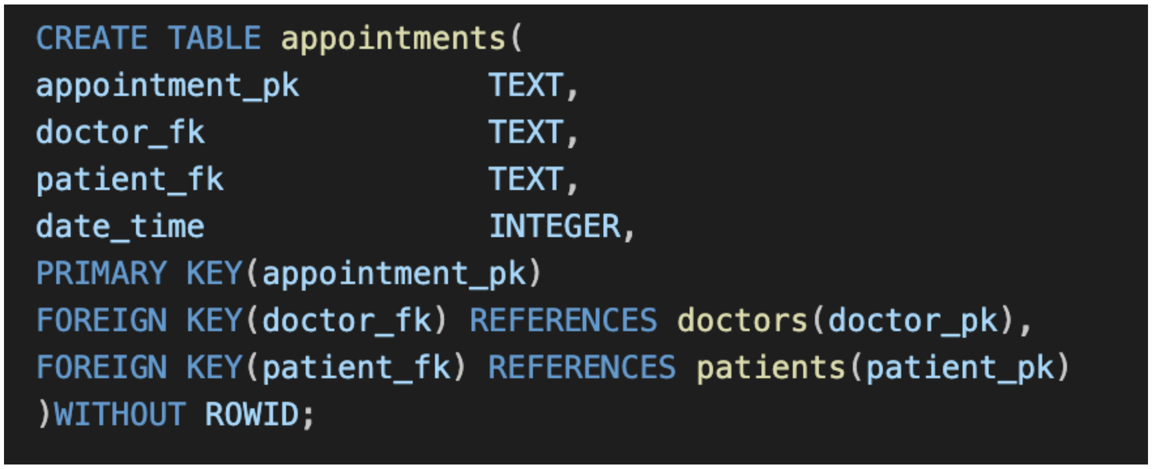
Normalization

Normalization is a database design principle that organizes data to reduce redundancy and improve data integrity. The primary goals are to ensure data is stored efficiently, to eliminate duplicate data, and to ensure relationships between data are logical and consistent.
To explain Normalization further we can split it into 3 forms of tables:

1. **First Normal form:** This form is about ensuring that each table cell only contains a single value, and every column should have a unique name. This helps with eliminating duplicate columns and ensures that every column only contains one type of data.
   Example from database:
   In the database above this principle shows if you look at the patients table and its relation to the prescription table:
   

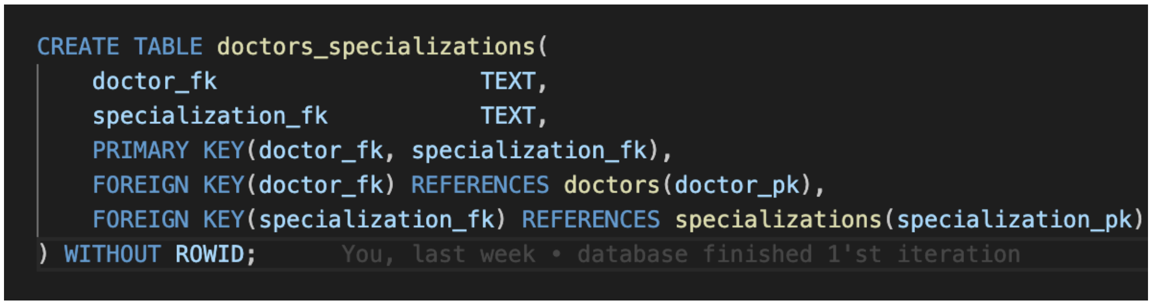
   The reason we choose to have a separate table for prescriptions, is because it is both possible for a patient to have zero and many prescriptions.
   This prevents us from having multiple columns like prescription_1, Prescription_2, prescription_3 and so on, it also makes sure that we don't have any empty cells in our table.
2. **Second Normal Form:** A table is in 2. normal form if it first of all respects the rules of 1. normal form. Furthermore it has the be so that all non-key columns are functional dependent on the primary key
   Example from database: This can be seen in the appointments table where patient_fk,

doctor_fk and date_time are all fully dependant on the appointment_pk



```
CREATE TABLE appointments(
appointment_pk          TEXT,
doctor_fk               TEXT,
patient_fk              TEXT,
date_time               INTEGER,
PRIMARY KEY(appointment_pk)
FOREIGN KEY(doctor_fk) REFERENCES doctors(doctor_pk),
FOREIGN KEY(patient_fk) REFERENCES patients(patient_pk)
)WITHOUT ROWID;
```

3. **Third Normal Form:** For a table to be in 3. form it first has to be in second form. On top of that it can't have any non-key columns, that depends on another non-key column.
   Example from database:
   The doctors_specializations table is in 3. form because there's
   no non-key columns, there's only docter_fk and specialitation_fk.

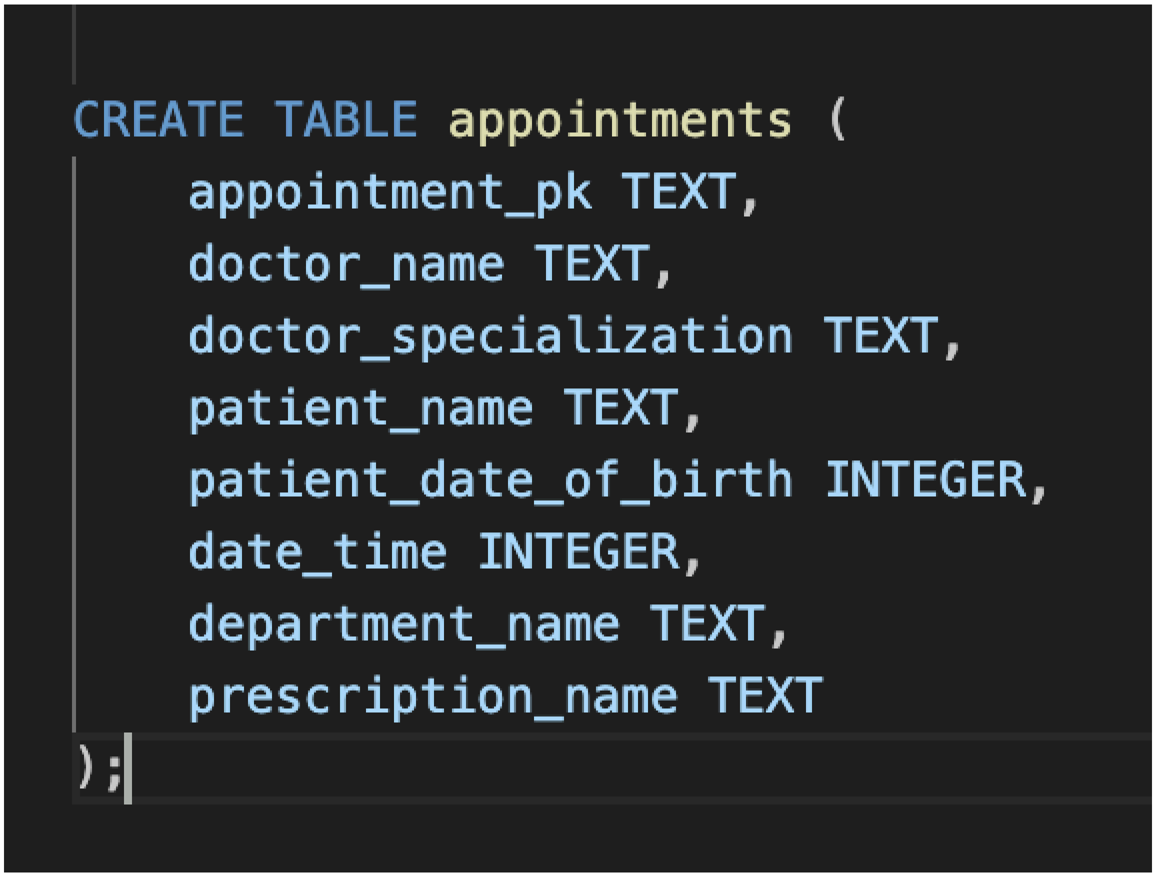


```
CREATE TABLE doctors_specializations(
    doctor_fk                   TEXT,
    specialization_fk           TEXT,
    PRIMARY KEY(doctor_fk, specialization_fk),
    FOREIGN KEY(doctor_fk) REFERENCES doctors(doctor_pk),
    FOREIGN KEY(specialization_fk) REFERENCES specializations(specialization_pk)
) WITHOUT ROWID;        You, last week • database finished 1'st iteration
```

Normalization is important because:

- **Reduced Data Redundancy:** By splitting the data into related tables, each piece of information is stored only once, reducing the chances of inconsistencies and doubles of the same data.
- **Improved Data Integrity:** Ensuring that logical dependencies are enforced helps maintain the correctness and consistency of data.
- **Ease of Maintenance:** With a clear structure, it is easier to manage and update the database, as changes in one place propagate correctly without needing multiple updates.

Example of appointments table without normalization:
If we had a single table storing all information without normalization


```
CREATE TABLE appointments (
    appointment_pk TEXT,
    doctor_name TEXT,
    doctor_specialization TEXT,
    patient_name TEXT,
    patient_date_of_birth INTEGER,
    date_time INTEGER,
    department_name TEXT,
    prescription_name TEXT
);
```
In this case, each appointment would repeat the doctor's name, specialization, patient's name, and other details, leading to a lot of redundant data.

Database as Document database or Graph database:

In the examples shown earlier we are only looking at the system as a relational database, but there are actually other options, which come with their pros and cons. One alternative is a document database, such as ArangoDB, and the nice thing about Arrangodb is that it also supports graph databases.

Document Database

A document database stores data in a semi-structured format, typically using JSON-like documents. Each document represents a single entity and its related data, encapsulating information in a hierarchical structure. This approach can simplify data representation and access when trying to read the data.

Example: Patients Collection
In a document database, the patients table from the SQL example can be represented as a collection of documents. Each patient document can have related data such as prescriptions and appointments directly within the patient document. This approach reduces the need for joins and can improve read performance, as all related data is stored together.

Pros and Cons of Document Databases

Pros:

- **Simplicity:** Easier to model and read complex, hierarchical data structures.
- **Performance:** Improved read performance for nested data as all related information is stored together.
- **Scalability:** Designed for horizontal scaling, making it easier to distribute data across multiple servers.

Cons:

- **Data Redundancy:** Can lead to redundant data if the same information is duplicated in multiple documents.
- **Complex Updates:** Updating nested data can be complex and may require rewriting entire documents.
- **Lack of Standardization:** No strict schema, which can lead to inconsistency if not managed properly.

Graph Database

A graph database represents data as nodes (entities) and edges (relationships), making it particularly suitable for data with complex relationships. Each node has a structure like a document, and edges tell us about the relationships between nodes.

What is a Graph Database

A graph database is designed to treat the relationships between data as equally important to the data itself. It consists of:

- **Vertices (Nodes):** Represent entities such as patients, doctors, appointments, etc.
- **Edges**: Represent relationships between entities, such as a patient having an appointment with a doctor.
- **Paths:** Sequences of edges that connect two or more vertices, representing a chain of relationships.
- **Attributes:** Properties or metadata associated with vertices and edges.

Pros and Cons of Graph Databases

Pros:

- **Complex Relationships:** Efficiently manages and queries complex relationships.
- **Flexibility:** Easily adapts to changes in the data model.

Cons:

- **Complexity:** Can be more complex to model and query compared to document or relational databases.

Conclusion:

In conclusion, while relational databases use primary and foreign keys to manage relationships between tables, document databases embed related data within a single document, and graph databases use edges to represent relationships explicitly. Each approach has its pros and cons and the best choice depends on the specific needs and complexity of the data and queries.

Handling Large Documents in a Document Database:

When a document contains too much data, it can lead to slower performance, such as slower read/write operations and increased complexity in managing and querying the data. To solve this situation, split the large document into smaller, more manageable pieces while maintaining the connections between related documents. Here will go through some of the ways to handle this:

1. Splitting Large Documents

Instead of having one large document, break it down into smaller, subdocuments. This approach keeps each document focused on a specific aspect of the data and helps keeping the individual document size down.

Example: Consider a large patient document with nested prescriptions and appointments Instead of storing everything in one document, you can create separate collections for prescriptions and appointments.

2. Storing Connections to Other Documents

This approach is about using references to connect the different documents to each other. So instead of embedding large amounts of data in every document we use references such as keys to connect the relevant data together. By doing it this way we can have a little more information in the patient Document, but can easily fetch the rest of the data from the other documents by using the keys.

But isn't it just a relational database then?

By following this approach there is surely a lot of resemblance from a relational database, but there are still key differences when it comes to how data is stored, accessed and managed.

Key differences :

1. Flexible Schema
    a. When working with relational databases the tables are predefining schemas/rules that we must follow when working with data.
    b. When working with document databases the Schema is flexible, meaning that Each document can have a different structure, making it easier to evolve the data model without downtime.
2. Embedded Data:
    a. **Relational database:**Normalization often requires multiple tables and joins to fetch related data, which can be complex.
    b. **Document Database:** Supports embedding related data within documents, reducing the need for joins and improving read performance. Even when using references, some data can still be embedded.
3. Horizontal Scalability:
    a. **Relational Database:** Typically scales vertically by adding more power to a single server.
    b. **Document Database:** Designed for horizontal scalability, making it easier to distribute data across multiple servers
4. Data Retrieval Patterns:
    a. **Relational Database:** Optimized for complex queries involving multiple joins and transactions.
    b. **Document Database:** Optimized for retrieving entire documents quickly. Queries can be simpler and more aligned with application requirements.

Full text search in relational and document database

Full text search is a technique used in databases to make it possible to have fast and efficient search for phrases in large text fields within documents. This is very useful if you want to be able to search on something and then get results of everything that is related to your search phrase.

How Full text search works

Full Text Search involves making an index with all the words in a text field and where they are in the text. When you run a search, the database looks at this index to quickly find and rank the documents that match the search words.

What is an In-Memory database?

An In-Memory database is a database that primarily stores data in the ram instead of storing data “on the disk”. This allows for significantly faster data access times compared to traditional disk based databases. In-Memory databases are particularly useful for applications that require real-time data processing and really fast read/write operations.

Full text search in hospital system

Imagine that each patient has notes on them which could contain some kind of medical history. If we make Full text search on these notes, then it will be possible to easily find the correct patient without the name of the patient. Instead you can just search for parts of their history.

Redis: An Example of an In-Memory Database

Redis (Remote Dictionary Server) is one of the popular in-memory databases, and is known for its performance, flexibility and simplicity. Redis supports different kinds of data structures like strings, hashes, lists and much more.

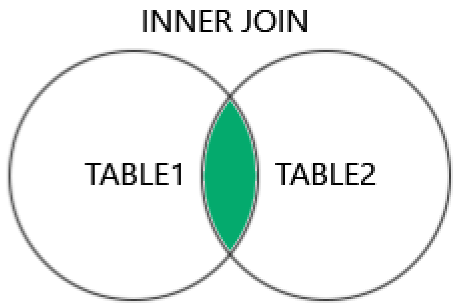
Parts of the System That Could Use In-Memory Databases

In the hospital system that we are building the use case of something like Redis could be data that is frequently accessed. This data could be saved as cache for easy and fast access. An example of this could be if a certain patient profile is frequently accessed, then it could make sense to save the data of this profile in redis. This will not only make the reads faster, but also take away stress from the main database which would be more inefficient finding the same data over and over.

Another use case could be using Redis for session storage. Here we could for example store a log of what a user has been doing, what are their settings, and maybe the latest data they have searched because they are likely revisiting to check the same data.

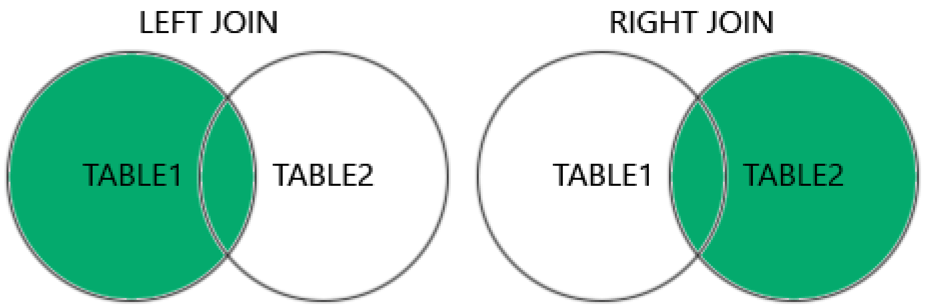
Different kinds of JOINS in sql:

INNER JOIN:



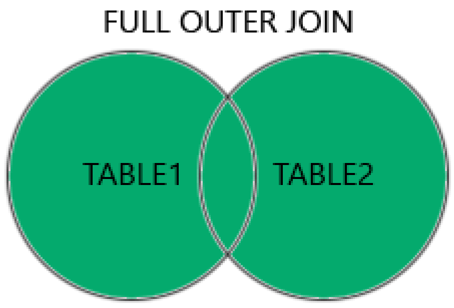
When doing an INNER JOIN we ask to get all the data from the tables where there are matching values in both tables. This means If we make an Inner join of appointments and then patients and doctors.Then if there is an appointment without a corresponding patient or doctor, it will not appear in the results.

LEFT JOIN and RIGHT JOIN:



A LEFT JOIN also takes multiple tables and shows the data when there is certain data that matches. The difference here is that if we have Appointments on the left side, then all appointments will show regardless of them missing a matching doctor or Patient.
The same applies for a RIGHT JOIN here the patients and doctors would always show, but maybe there would be no appointment to match.

FULL JOIN:



A FULL JOIN would return all the data if there is either a match on the LEFT or RIGHT site table. If it exists, then it will show in the results.

SELF JOIN

A SELF JOIN is a regular JOIN but where the table JOINS with itself. This could for example be useful if we wanted to see how many patients that had the same last name or how many doctors that have the same specialization.

Stored procedures, Functions and Routines

A **stored procedure** is a set of sql statements that can be stored in the database, and then be executed as a single statement. They are made so that they can accept parameters and perform operations like querying and updating data and return the results.

Functions are alot like stored procedures but only return a single value and has the option to be used in sql expressions. They also have the option to accept parameters and perform different kinds of operations.

The term **"routine"** is often used interchangeably with stored procedures and functions, referring to any reusable block of code stored in the database.

Advantages of Stored procedures and Functions

The use of Stored procedures and functions has several benefits and can be very helpful as a system grows in size.

Performance

By using stored procedures and functions we can have a positive impact on performance because they are stored in the database and it will lead to faster execution of tasks.

Security

By using these methods we are also introducing an extra level of security, because we control the access to the data through predefined procedures.

Reusability

With the help of stored procedures and functions we can create the complex logic once, and then reuse it as it fits throughout our application. This makes it a lot easier to make sure we reuse the code we already have which leads to maintainability.

Maintainability

By creating complex logic with the help of functions that can be reused in multiple places of the application we also make it alot easier to update and maintain the code.

Consistency

Last but not least the use of stored procedures and functions also helps us by letting us use the right function for the right task. This will make sure that things are happening consistently throughout the application.

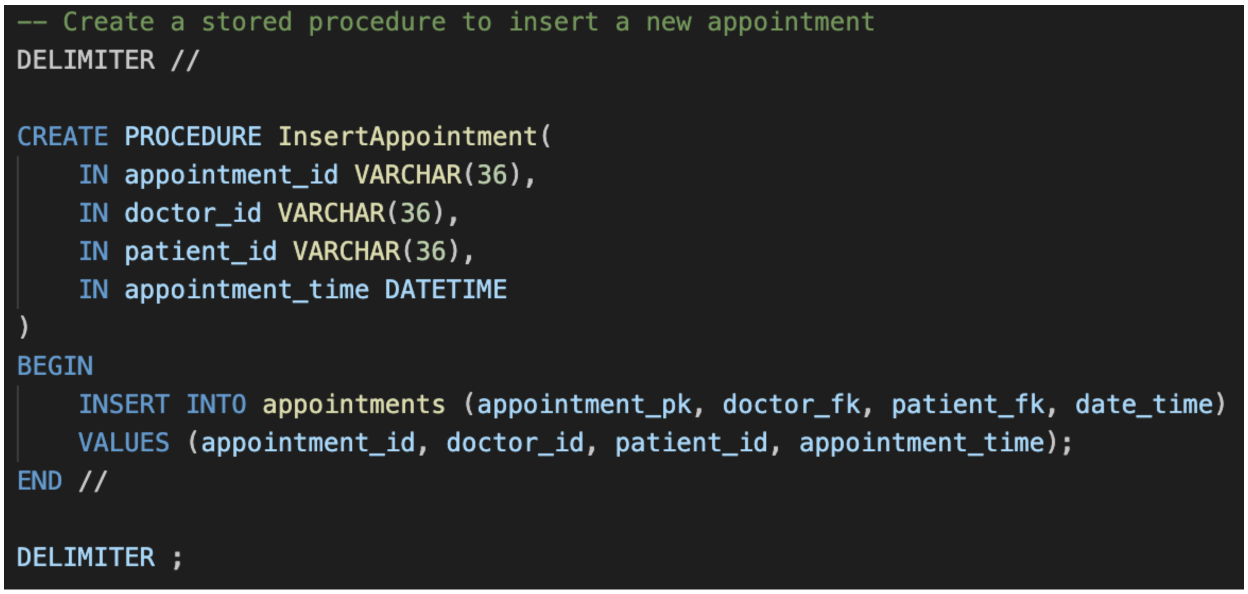
Stored procedures in my system

Because it is not possible to create stored procedures in SqlLite I have made two examples , which could run in a mySql database instead.

I have chosen to create some simple examples where I only include

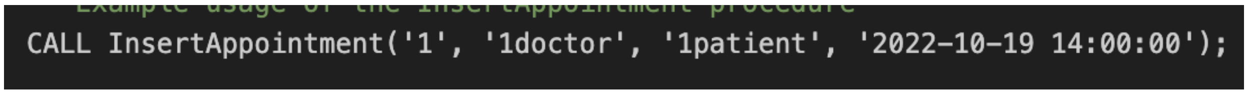
- patients table
- doctors tabe
- appointments table

Example 1:



```
-- Create a stored procedure to insert a new appointment
DELIMITER //

CREATE PROCEDURE InsertAppointment(
    IN appointment_id VARCHAR(36),
    IN doctor_id VARCHAR(36),
    IN patient_id VARCHAR(36),
    IN appointment_time DATETIME
)
BEGIN
    INSERT INTO appointments (appointment_pk, doctor_fk, patient_fk, date_time)
    VALUES (appointment_id, doctor_id, patient_id, appointment_time);
END //

DELIMITER ;
```

This Procedure is made to insert a new appointment with the needed information, which is just passed as parameters
In order to call the procedure we simply need to use CALL and include the needed parameters:



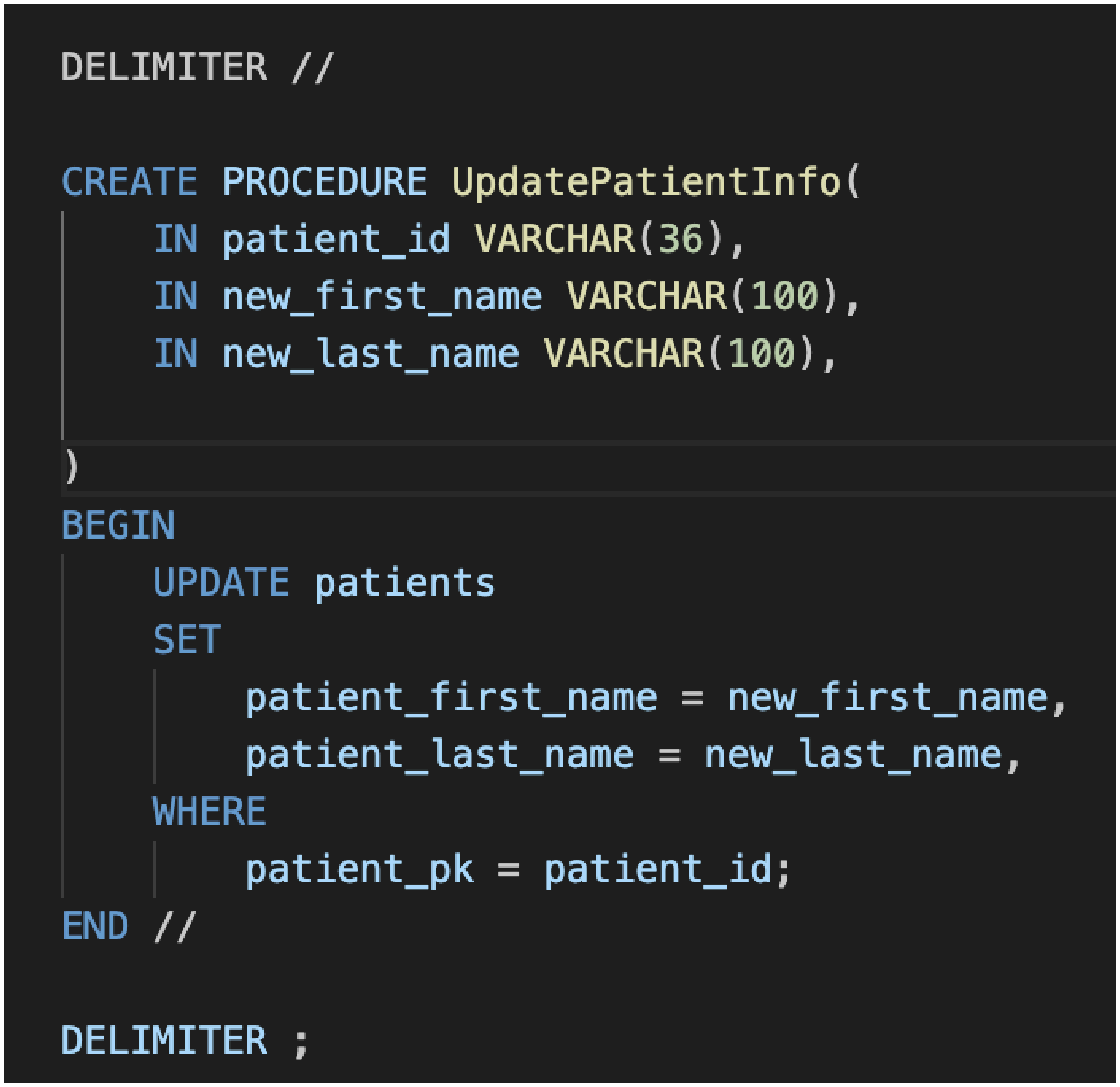
```
CALL InsertAppointment('1', '1doctor', '1patient', '2022-10-19 14:00:00');
```

This CALL would insert a new appointment where the id would be 1, doctor_id would be d1, patient_id would be p1 and the appointment_time would be the date of “2022-10-19 14:00”

Example 2:

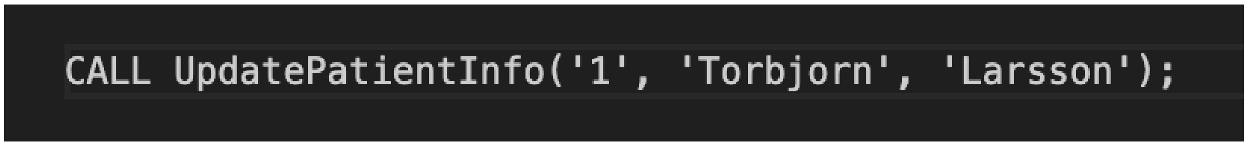
Imagine that a patient gets married and changes his last name, then we would need to have a way of updating this patient in the system.

Here we can also use a stored procedure in order to update the specific patient:



```
DELIMITER //

CREATE PROCEDURE UpdatePatientInfo(
    IN patient_id VARCHAR(36),
    IN new_first_name VARCHAR(100),
    IN new_last_name VARCHAR(100),

)
BEGIN
    UPDATE patients
    SET
        patient_first_name = new_first_name,
        patient_last_name = new_last_name,
    WHERE
        patient_pk = patient_id;
END //

DELIMITER ;
```

This code defines a procedure “UpdatePatientInfo” and asks for the parameters “patient_id”, “new_first_name” and “new_last_name”. It then uses the patient_id to find the specific patient and updates the first and last name to the values passed as parameters.
This is how the stored procedure would be called:



```
CALL UpdatePatientInfo('1', 'Torbjorn', 'Larsson');
```

This finds the patient with patient_id =1 and sets the first name to “Torbjorn” and last name to “Larsson”.

DELIMITER

The lines saying “DELIMITER //” and “DELIMITER ;” is there to allow the procedure to contain semicolons
The default delimiter in MySQL is a semicolon (;). However, when creating stored procedures, functions, or triggers, you often need to use semicolons within the body of these routines. This can confuse MySQL, as it will think that the routine definition ends prematurely.

To avoid this confusion, you can temporarily change the delimiter to something else, define your routine, and then change the delimiter back to the default semicolon. Common alternatives for the delimiter include // or $$.

Triggers

Triggers are special types of procedures, which execute automatically when a certain event happens. This event could be “INSERT”, “UPDATE” or “DELETE”. Triggers are used to automate specified actions and make the codebase easier to maintain.

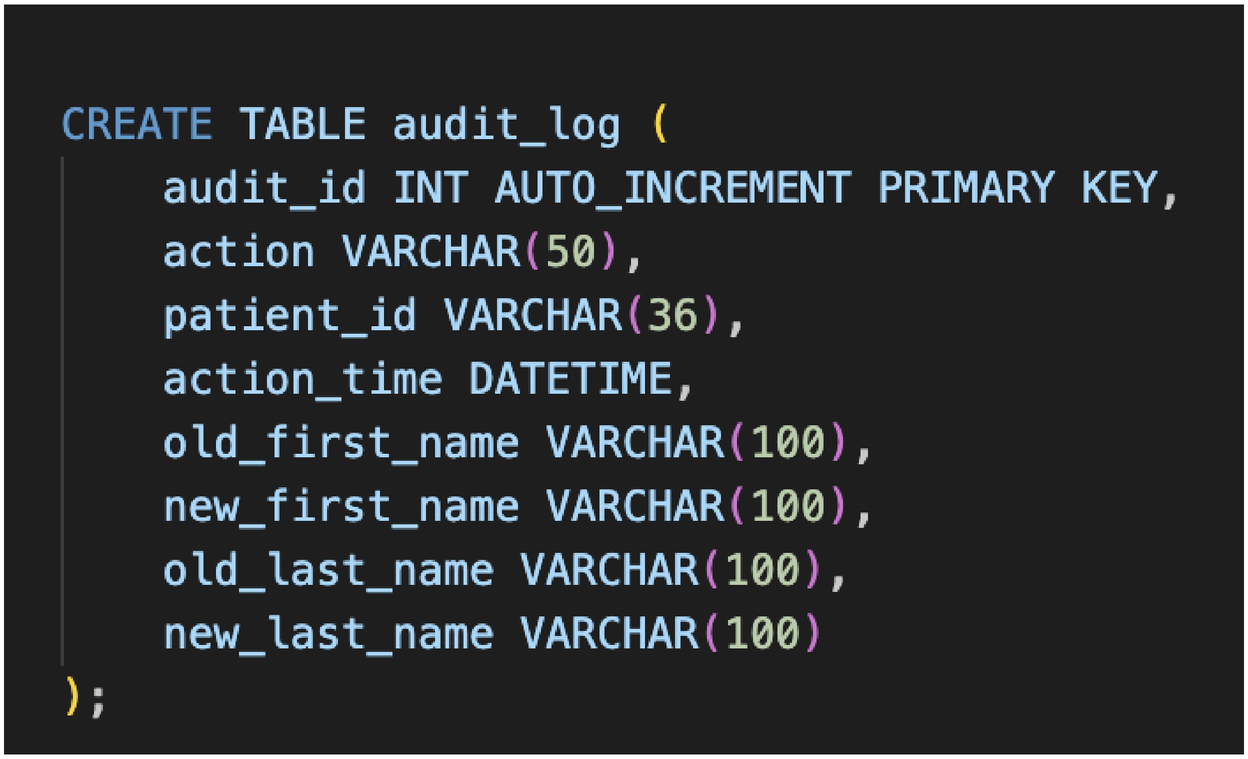
Some of the advantages of using triggers:

- **Automation:** Triggers can automate repetitive tasks and help to ensure that those tasks are performed consistently.
- **Audit Trails:** Triggers can automatically log changes to data, creating an audit trail.
- **Complex Logic:** Triggers can encapsulate complex logic that needs to be executed automatically when data changes.

Trigger Examples:

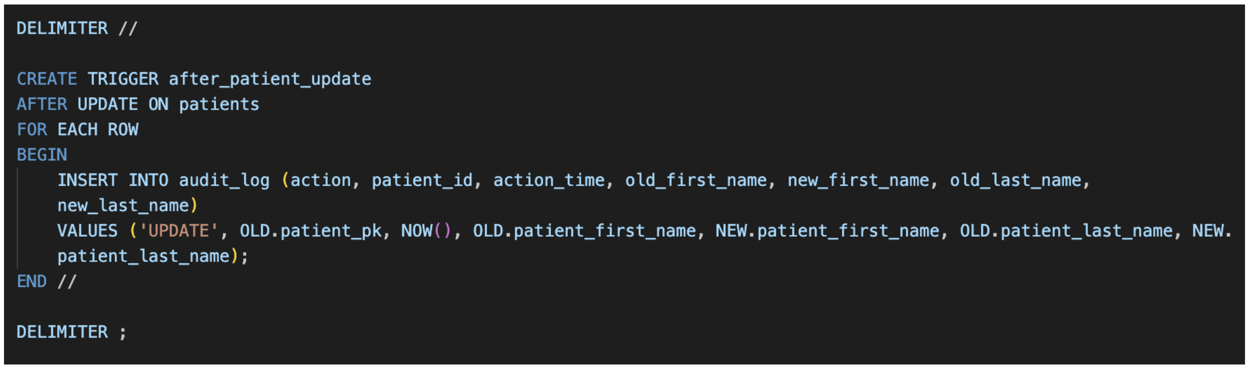
Example 1:

In this example we use a trigger to create an audit trail. This first of all requires a “audit_log” table:



```sql
CREATE TABLE audit_log (
    audit_id INT AUTO_INCREMENT PRIMARY KEY,
    action VARCHAR(50),
    patient_id VARCHAR(36),
    action_time DATETIME,
    old_first_name VARCHAR(100),
    new_first_name VARCHAR(100),
    old_last_name VARCHAR(100),
    new_last_name VARCHAR(100)
);
```

Where we can store the log data, when certain events happen. Then we need to Create the Trigger:



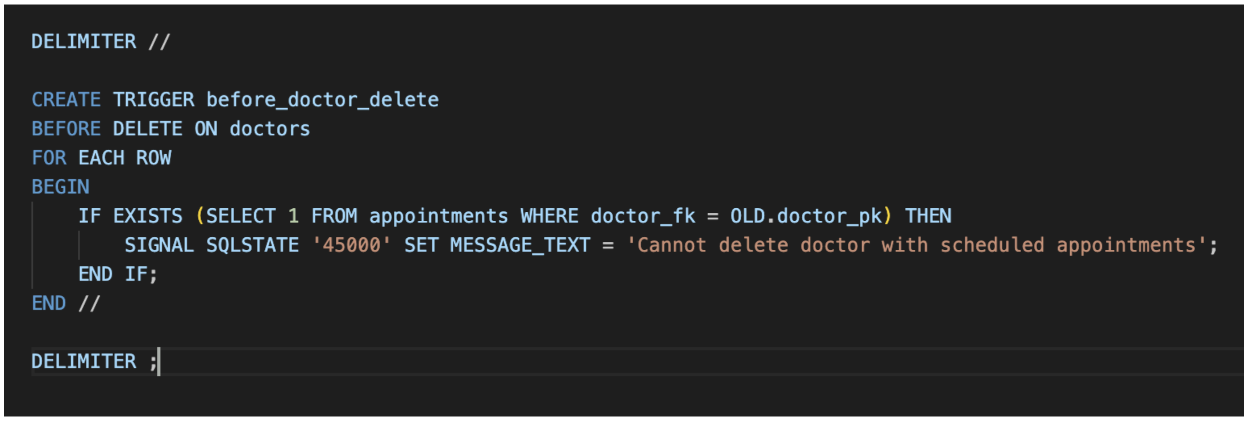
```sql
DELIMITER //

CREATE TRIGGER after_patient_update
AFTER UPDATE ON patients
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (action, patient_id, action_time, old_first_name, new_first_name, old_last_name,
    new_last_name)
    VALUES ('UPDATE', OLD.patient_pk, NOW(), OLD.patient_first_name, NEW.patient_first_name, OLD.patient_last_name, NEW.
    patient_last_name);
END //

DELIMITER ;
```

This Trigger is executed after an update happens in the patients table. The trigger then goes through each row which has been affected by the update and Then INSERTS it into the audit_log table.

By having this trigger we will always be able to go back and get an idea of how the data was changed over time.

Example 2:

My second trigger has the purpose of preventing doctors from getting deleted in the system if they still have appointments to attend.


```
DELIMITER //

CREATE TRIGGER before_doctor_delete
BEFORE DELETE ON doctors
FOR EACH ROW
BEGIN
    IF EXISTS (SELECT 1 FROM appointments WHERE doctor_fk = OLD.doctor_pk) THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot delete doctor with scheduled appointments';
    END IF;
END //

DELIMITER ;
```

This trigger is specified to happen BEFORE the deletion of a doctor, which can be seen in the line saying:
“BEFORE DELETE ON doctors”
It then goes through each row and checks if any appointment associated with the doctor exists. If an associated appointment exists, then the line starting with “SIGNAL SQLSTATE” raises an error which prevents the deletion of the doctor.

Views

Views are virtual tables which consist of data from other actual tables. Views gives us a way of providing data in a table like structure even though not all the data lives in the same actual table. This can simplify data access and at the same time enhance security because it can access specific data.

Benefits of Views

- **Simplified Querying:** makes complex queries into simpler, reusable structures.
- **Security:** Restrict access to sensitive data by exposing only specific columns or rows.
- **Abstraction:** Hide the complexity of underlying tables and their relationships.
- **Data Integrity:** Provide a consistent view of data, even if the underlying tables are complex.

Union, Having and Group By

Sql has different features when it comes to querying and manipulating data, and some of the useful ones are “UNION”, “HAVING” and “GROUP BY”.
These are used to combine results of queries and filtering.

UNION

Union is used to combine two or more “SELECT” queries into a single result. In order to use UNION it is required that the queries have the same number of results.

When to use UNIONS

we use unions if we want to combine results from multiple tables or queries and want so see them in one single result

GROUP BY

GROUP BY is when you put rows with the same values in certain columns together into summary rows. It's like counting how many rows are in each group. It's often used with things like COUNT, MAX, MIN, SUM, and AVG to do certain things to each group of rows.

When to use GROUP BY

We can use GROUP BY when we want to summarize data from multiple columns, such as calculating totals or maybe averages.

HAVING

Having is used to filter groups based on certain conditions. it’s very similar to WHERE but is used with GROUP BY.

When to use HAVING

HAVING is often used to filter results after summarization and or to apply conditions to the summarized data. Like when er use a SELECT query with WHERE.

SQL constraints

Constraints are the ruleset which is applied to the data columns in our database. These are essential in order to ensure data integrity and consistency. They can prevent us from entering invalid data and help us make it easier to maintain the quality of our database.

Types of contractions

- **Primary Key:** Uniquely identifies each row in a table.
- **Foreign Key:** Ensures the value in one table matches a value in another table, establishing a relationship between them.
- **Unique:** Ensures all values in a column are unique.

- **Not Null:** Ensures a column cannot have NULL values.
- **Check:** Ensures the values in a column meet specific criteria.
- **Default:** Assigns a default value if no value is provided.

CASCADE ON DELETE

CASCADE on delete is kind of a foreign key constraint, which automatically deletes rows in a child table when the row with the primary key is deleted in the parent table. This is very useful for making sure that we won't have any foreign key without a corresponding primary key.
This could especially be useful in our system if we delete a patient, then this could ensure that the related appointment will also be deleted when the patient is no more in the database.

Vertical vs Horizontal Scaling

Vertical scaling:

Vertical scaling is simply about upgrading the hardware on where the database is run e.g. CPU, RAM or Storage.
Pros:
- Simpler to implement as it doesn't require changes to the application architecture.
- Easier to manage since everything is on a single machine.
- Beneficial for applications with high transaction rates requiring strong consistency.

Cons:
- There are hardware limits to how much a single machine can be scaled up.
- Single point of failure; if the server goes down, the entire application is affected.
- Typically more expensive per unit of performance compared to horizontal scaling.

Horizontal Scaling:

Horizontal scaling is about adding more servers in order to split the load from the individual machines
Pros:
- Can theoretically scale indefinitely by adding more servers.
- Provides high availability and redundancy. If one server fails, others can take over.
- Cost-effective as commodity hardware can be used.

Cons:
- More complex to implement and manage due to the need for distributed systems.
- Requires changes to the application architecture to handle distributed data.
- Ensuring data consistency across multiple servers can be challenging.

Database type and it’s scalability:

Relational databases - scaling

Vertical scaling is traditionally what relational databases are best suited for due to their reliance on the ACID principles.
Horizontal scaling can be used for archiving, but is more complex, and often requires splitting the database into smaller more manageable pieces.

Document database - scaling

Document databases can be vertically scaled, but is truly designed to scale horizontally, where they can distribute data across multiple servers.

Graph database - scaling

Often initially scaled vertically. Graph operations can be resource-intensive and benefit from powerful components like a good CPU or more and fast RAM.

Horizontal scaling is possible but often comes with compromises to performance and unwanted complexity.

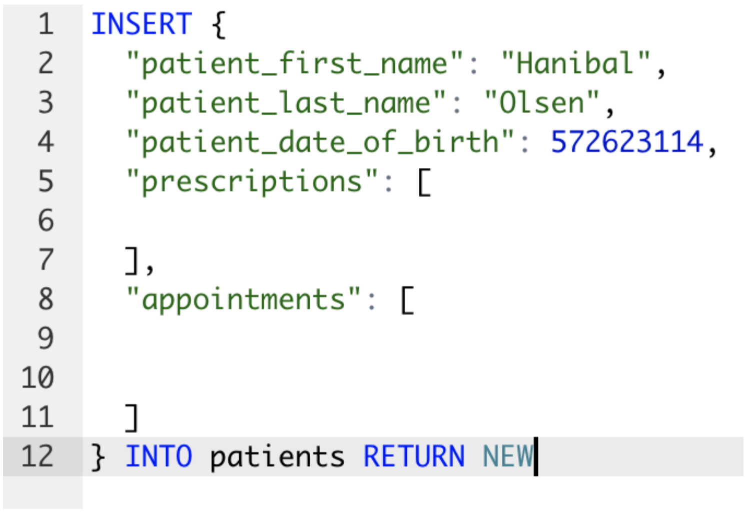
CRUD document database:

CRUD (create, read, update, delete) queries when used in arragodb is written in AQL (ArangoDB Query Language)

Create:

This query inserts a new patient in the patients collection.
The first_name of the new patient is “Hanibal”, last_name “Olsen”, his date of birth in an epoch format and an empty list of prescriptions, and also an empty list of appointments.

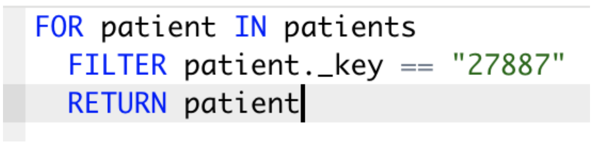
After inserting him in the patient's collection the new patient is returned.



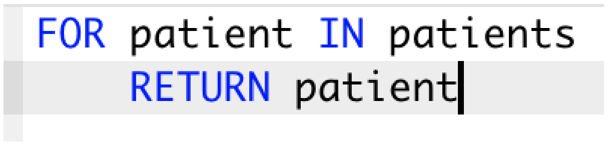
```
INSERT {
  "patient_first_name": "Hanibal",
  "patient_last_name": "Olsen",
  "patient_date_of_birth": 572623114,
  "prescriptions": [

  ],
  "appointments": [


  ]
} INTO patients RETURN NEW
```

Read:

For reads I have two queries. The first one is used to get a specific patient and only return one patient, where the other is made to get all patients and return them one by one.



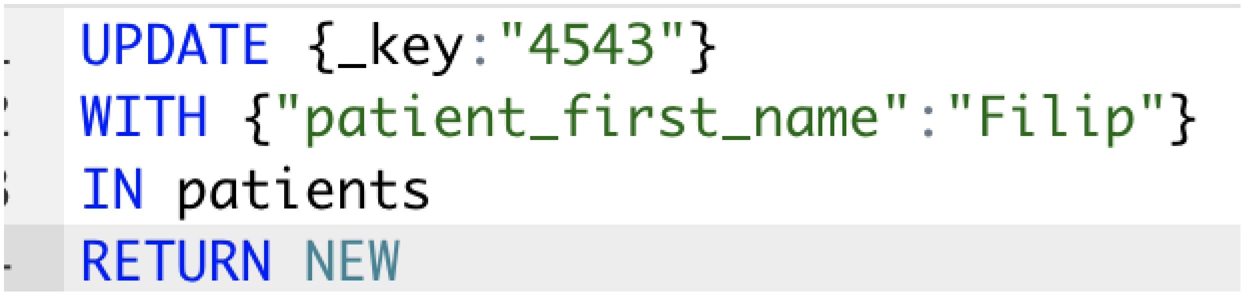
```
FOR patient IN patients
  FILTER patient._key == "27887"
  RETURN patient
```



```
FOR patient IN patients
     RETURN patient
```

Update:

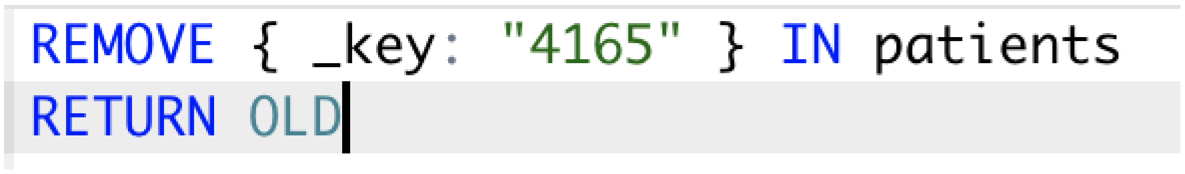
This update query is used to update a specific patient chosen by the provided key.
This query changes the first name of the patient to “Filip”, and then returns the newly updated patient.



```
UPDATE {_key:"4543"}
WITH {"patient_first_name":"Filip"}
IN patients
RETURN NEW
```

Delete:

This query deletes a patient from the patients collection by the provided key and then returns the one patient which is no longer part of the patients document.



```
REMOVE { _key: "4165" } IN patients
RETURN OLD
```

ACID

In order to explain the ACID principle, we need to address something called a transaction. A transaction is often described with the example of moving money from one bank account to another. This action has multiple steps, which all need to succeed in order for the transaction to actually happen. If we want to move 100$ from account-1 to account-2 we could have the following steps:

1. Check if account-1 has a balance of at least 100$
2. Subtract 100$ from account-1
3. Add 100$ to account-2

The idea is, that if any of these steps fail then nothing will happen. This ensures that we can't subtract money from account-1 without adding them to account-2 or that we can't subtract money from account-1 if the account has a balance which is too low.

In our database an example could be that in order to make an appointment the following checks needs to succeed:

1. Check that the patient exists
2. Check if doctor with the right specialization is available
3. Check if doctor is available in the chosen date
4. Check if the department has room for the appointment on the chosen date.
5. Book appointment.

So in order to book an appointment all these steps need to succeed. If any of them fail and we still allow the booking to be made, then we can have appointments with no doctors, lack of rooms for the doctor to see the patient or a booked room and doctor, but no patient because of a typo in the patients security number.

So in short an ACID transaction is a group of database read and write operations that only succeeds if all the operations within succeed.

ACID stands for:

- Atomicity

- Consistency
- Isolation
- Durability

Atomicity

Atomicity the rule of dividing each step so it only contains one action which can either fail or succeed. This is the principle explained above, where we can't book an appointment without having a doctor who's free at the chosen date.

When interacting with the database through python we can use the `try`, `except`, `finally` approach. This ensures that first we `try` to execute the actions/steps, and if they all succeed, then we move to `finally` . But if any of the actions fail, then we are sent to `an exception` which throws an error and stops the execution of actions/steps.

Consistency

In ACID Consistency is ensured by setting up some criteria/rules that need to be met in order for the transaction to happen.

In our example above we need to have some rules that checks if the doctor is available for the chosen date/time so that a doctor can't be booked for two appointments at the same time. These checks/rules ensure that our database will always be in a correct state where everything fits together.

Isolation

Isolation is about only running one transaction at a time, or in other words "isolate" the transactions. in our example above this could be that we won't be allowing 2 appointments to be created at the same time, because the same doctor could be booked for two appointments at the same time, which wouldn't be possible in the real world.

Therefore we will always complete one transaction before starting another.

Durability

Durability ensures that when a transaction completes and the changes are written to the database, then the changes are saved. So if the server should crash, or in case of an power outage, then our transaction has still happened and can be seen in the database. On the other hand, if the transaction is not committed/written when the server crashes or the power goes out, then it will be as if nothing ever happened and no changes to the database will be made. This is again secured by the combination of the terms above and the `try` , `except` , `finally` approach when interacting with the database.