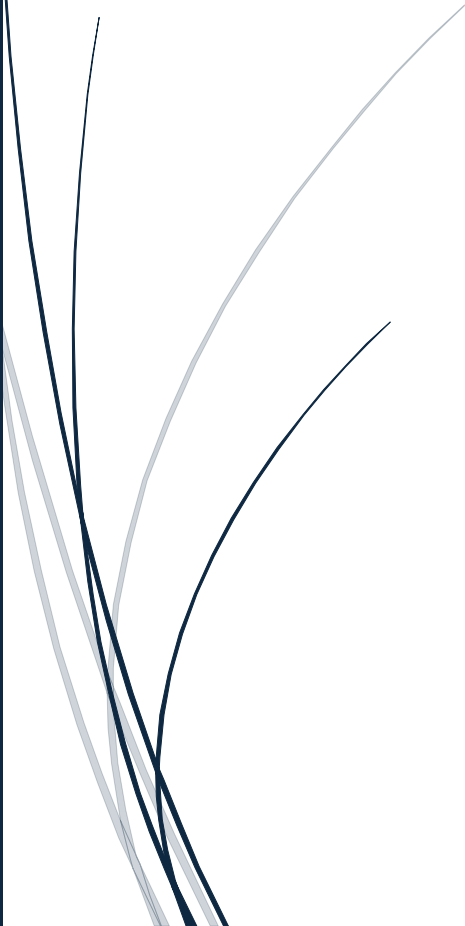




24/07/2026

MedAssist Clinic Database Design and Implementation



Oyebamiji Oluwafemi Oyedokun
2409692

Table of content

Contents

1.0	Introduction	2
2.0	Database Design	3
2.1	Main Entities, Attributes, And Relationships For Medassist Clinic	3
2.2	Database Design- ERD with all entities, keys, and attributes.....	5
2.3	Explanation of the design decisions, such as normalization and entity relationships.	5
3.0	Database Implementation.....	8
3.1	Table Creation and Constraint implementation	9
4.0	Data Manipulation Language (DML)	16
4.1	Sample data insertion.	17
4.2	Trigger Execution and stored procedure execution, Transaction Scenario in MedAssist Clinic database.	24
4.2.1	Example show Trigger Executes in the MedAssist Clinic database	24
4.2.2	Example show Stored Procedure Execution in the MedAssist Clinic database	27
4.2.3	Example show transactional Scenario in the MedAssist Clinic database.	29
4.2.4	Useful Select Queries	34
5.0	Performance Tuning Enhancement	36
5.1	Indexing strategy, focusing on performance critical queries.	36
5.2	How indexing benefits Campus Club's operational data handling.	37
5.3	Improvements in query speed and efficiency.	37
6.0	Conclusion	39
6.1	Reflection.....	39
	References	41

1.0 Introduction

The MedAssist Clinic project looks at how a safe, effective, and expandable relational database system was made to meet the clinic's growing operating needs. MedAssist clinic is an outpatient facility that offers consultations, diagnostics, and referrals to specialists.

Digitising these processes is the only way to bring them up to date, make services more efficient, reduce administrative work, and make sure that everyone always has access to correct health data. This situation is important because it shows real problems in healthcare, where data integrity, multi-user access, and transactional reliability all have a direct effect on the level of care for patients. To keep things consistent, stop mistakes, and allow for conditional commit or rollback, the project makes sure that the database design includes structured entities, indexing techniques, and transaction-controlled stored procedures.

The final answer meets MedAssist's functional requirements, which makes it a strong base for safe patient care, streamlined workflows, and long-term operational scalability.

2.0 Database Design

Database design is the process of figuring out what data a system needs, putting it in tables that are linked to each other, and setting the rules (like relationships, constraints, and keys) that make sure the data stays correct, consistent, and useful [1].

Database design is the methodical process of organising, arranging and modelling data such that a database can store information efficiently, securely and in a way that supports the real-world functions of an organization [2].

Managing patients, doctors, appointments, treatments, feedback, and making sure secure, consistent access to data are just a few of the operational issues that come up in the MedAssist Clinic. A good relational database solves these problems directly by putting data into structured entities, setting up relationships, and following rules that keep things correct and reliable [2].

2.1 Main Entities, Attributes, And Relationships For Medassist Clinic

Main Entities are the major items within a system for which data must be stored and a real-world object you want to keep information about. It usually becomes table in relational database [1].

Attributes are the informational components that characterise an entity. They become the columns inside a table [2].

Relationships describe how entities are connected to each other. They are implemented using foreign keys in a relational database [2].

The MedAssist Clinic scenario describes a healthcare environment where Patients, Doctor, Appointment, Department, Treatment, Staff (Non-medical staff), transaction, Feedback interact. This identifies the core entities required to support the MedAssist clinic's operations.

2.2 Database Design- ERD with all entities, keys, and attributes

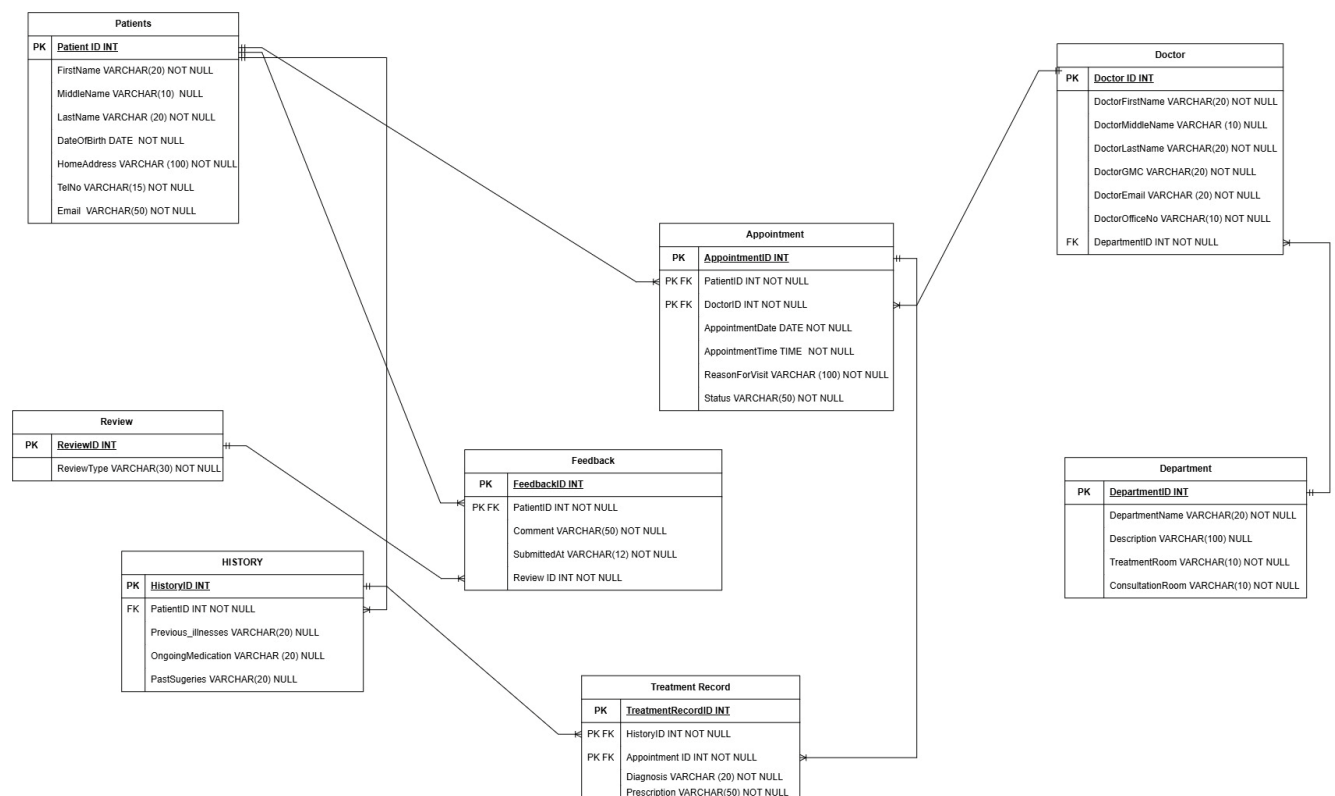


Diagram 1 show the Database Design

2.3 Explanation of the design decisions, such as normalization and entity relationships.

Normalisation is used in the database design to get rid of unnecessary copies and make sure the data is correct. There is a table for each main idea, such as Patient, Doctor, Appointment, Department, Treatment Record, History, Review, and Feedback. This makes sure that the organization follows Third Normal Form (3NF). For example, a patient's medical history is kept separate in a different entity called "History," so it doesn't have to be repeated between appointments. Similarly, the

Treatment Record is tied to both the Appointment Record and the History Record. This keeps diagnoses and prescriptions from being written twice.

The diagram shows a well-organised, normalised database where each table is for a single idea that doesn't cross.

First Normal Form (1NF) is achieved by ensuring all attributes are atomic, for example, The patient names were split into fields for the first, middle, and last names.

Patient ID	First Name	Middle	Last Name
1	John	Andrew	Smith

Table 1 shows the 1NF Table

The second normal form (2NF) is attained by separating appointment details from the Patient and Doctor tables. 2NF requires that each table only contains data that is dependent on the whole primary key. The AppointmentDate, AppointmentTime, and DoctorID are moved to the Appointment table, eliminating partial dependencies and enhancing data organization.

Patient ID	First Name	Middle	Last Name
1	John	Andrew	Smith

Table 2 shows Patient table is 2NF

AppointmentID	PatientID	DoctorID	AppointmentDate	AppointmentTime	ReasonForVisit
1	1	1	2026-06-19	09:00	Check-up

Table 3 show Appointment table is 2NF

Third Normal Form (3NF) was clear because non-key attributes only depend on their main keys. For example, medical history is in a separate History table in 3NF so non-key attributes depend only on their main key. Illnesses and medication do not need to be repeated in every appointment row since each patient's history is saved once, reducing redundancy and resulting in clean, consistent, well-structured medical records across the database.

HistoryID	PatientID	Previous_Illness	OngoingMedication	PastSurgeries
1	1	Asthma	Inhaler	None

Table 4 show that History table is 3NF

A patient can have multiple appointments, and the same doctor handles all of them. This is called a one-to-many relationship. Departments put doctors into groups, and Feedback lets customers read reviews of their experience. Appointments are needed to keep treatment records, so treatments can't happen without a meeting. These choices make a clean, scalable structure that allows for accurate patient management, safe entry by multiple users, and reliable transactional operations.

3.0 Database Implementation

The data implementation is the process of making the database plan in the diagram work in a real relational database system, like MySQL, SQL Server, or PostgreSQL [3]. Data implementation involves converting the ERD into real database structures [3]. There are tables, keys, constraints, and stored processes that are made from the conceptual and logical models at this stage [4].

In the MedAssist Clinic system, data Implementation is the step where the ERD is turned into real database structures that work and help with daily clinical tasks. Patients, Doctor, Appointment, Department, History, Treatment Record, Review, and Feedback are all represented in the diagram as different tables to show how the clinic works and keep the data safe.

3.1 Table Creation and Constraint implementation

Table for Patient

The Image below shows Patients table with some fields for name, contact info, and date of birth. It makes PatientID an auto-incrementing primary key, important fields are NOT NULL, and middle name, address, telephone and email are optional values.

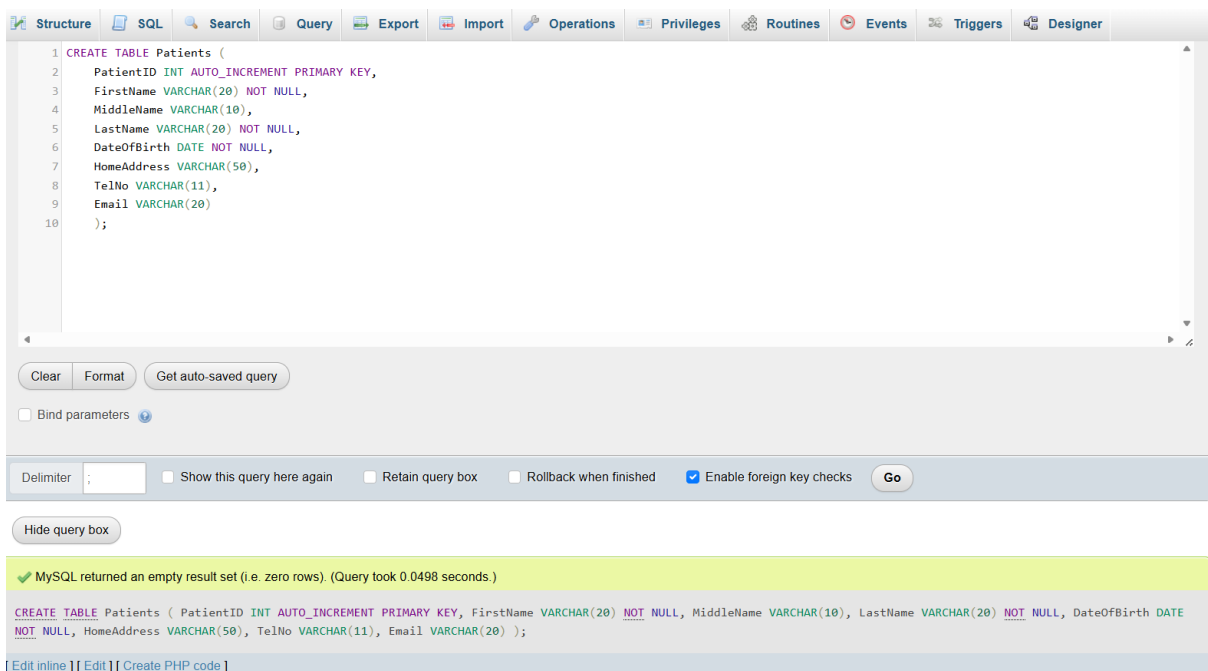
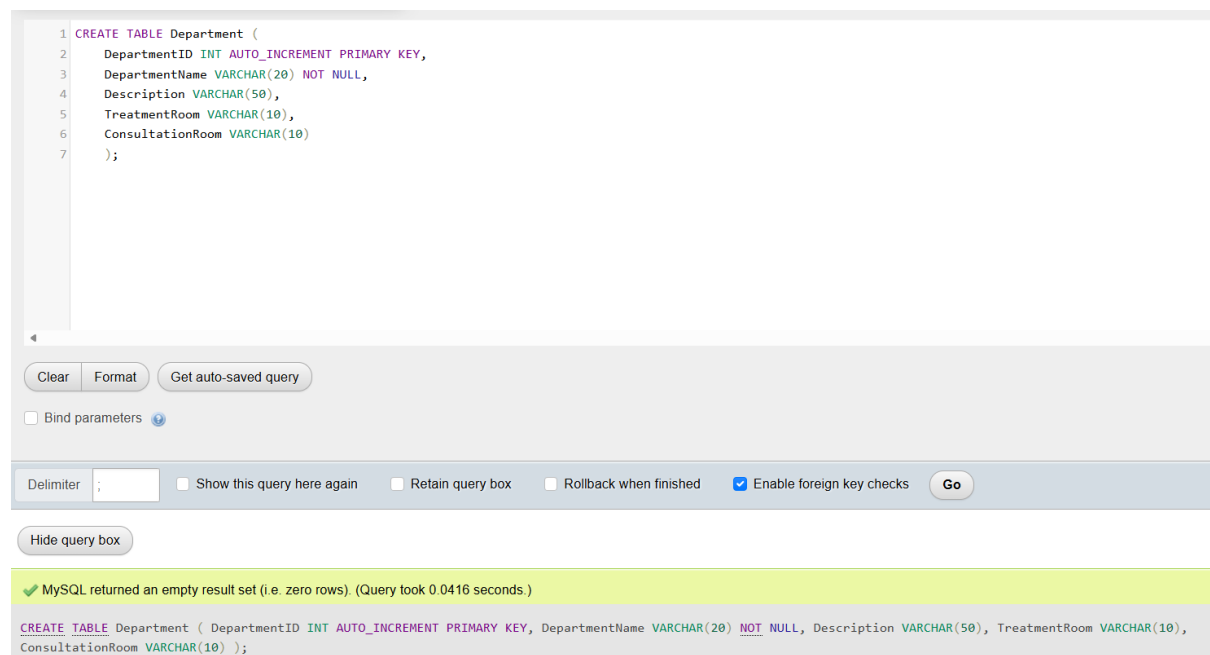


Image 1 show Patient Table was successfully created

Table for Department

The image below shows department table. It has a DepartmentID as an integer primary key and department name, description, treatment room and consultation room fields. Required fields are marked with NOT NULL, so optional fields can be NULL, leading to a clean and structured table design.



The screenshot displays a SQL query editor interface. The query text is as follows:

```
1 CREATE TABLE Department (  
2   DepartmentID INT AUTO_INCREMENT PRIMARY KEY,  
3   DepartmentName VARCHAR(20) NOT NULL,  
4   Description VARCHAR(50),  
5   TreatmentRoom VARCHAR(10),  
6   ConsultationRoom VARCHAR(10)  
7 );
```

Below the query text, there are several controls: a 'Clear' button, a 'Format' button, and a 'Get auto-saved query' button. A checkbox labeled 'Bind parameters' is also present. At the bottom of the editor, there is a 'Delimiter' dropdown set to ';', and several checkboxes: 'Show this query here again', 'Retain query box', 'Rollback when finished', and 'Enable foreign key checks' (which is checked). A 'Go' button is located to the right of these checkboxes. Below the editor, there is a 'Hide query box' button.

The execution result is shown in a green banner at the bottom: 'MySQL returned an empty result set (i.e. zero rows). (Query took 0.0416 seconds.)' Below this banner, the executed SQL statement is repeated in a smaller font.

Image 2 show department table was inserted successfully

Doctor Table

The image shows structure of the Doctor table. It makes DoctorID the primary key and has fields for the doctors' names, GMC number, email and office number. It associates each doctor with a department using a foreign key. Optional fields permit NULL if applicable.



```
1 CREATE TABLE Doctor (
2   DoctorID INTEGER PRIMARY KEY NOT NULL,
3   DoctorFirstname VARCHAR(20) NOT NULL,
4   DoctorMiddlename VARCHAR(10),
5   DoctorLastname VARCHAR(20) NOT NULL,
6   DoctorGMC VARCHAR(20),
7   DoctorEmail VARCHAR(20),
8   DoctorOfficeNo VARCHAR(10),
9   DepartmentID INT,
10  FOREIGN KEY (DepartmentID) REFERENCES Department(DepartmentID)
11 );
```

Clear Format Get auto-saved query

☐ Bind parameters

Delimiter ; ☐ Show this query here again ☐ Retain query box ☐ Rollback when finished ☒ Enable foreign key checks Go

Hide query box

✓ MySQL returned an empty result set (i.e. zero rows). (Query took 0.0561 seconds.)

```
CREATE TABLE Doctor ( DoctorID INTEGER PRIMARY KEY NOT NULL, DoctorFirstname VARCHAR(20) NOT NULL, DoctorMiddlename VARCHAR(10), DoctorLastname VARCHAR(20) NOT NULL, DoctorGMC VARCHAR(20), DoctorEmail VARCHAR(20), DoctorOfficeNo VARCHAR(10), DepartmentID INT, FOREIGN KEY (DepartmentID) REFERENCES Department(DepartmentID) );
```

Image 3 show Doctor Table was inserted successfully

Appointment Table

The image below shows Appointment table and explains how the records of appointments will be stored. Primary key is AppointmentID. It has foreign keys for Patient and Doctor to join each appointment to a patient and doctor. It also has fields for date, time, reason for visit and status. It offers a straightforward relational framework for scheduling data.



```
1 CREATE TABLE Appointment (
2   AppointmentID INTEGER PRIMARY KEY NOT NULL,
3   PatientID INT NOT NULL,
4   DoctorID INT NOT NULL,
5   AppointmentDate DATE NOT NULL,
6   AppointmentTime TIME NOT NULL,
7   ReasonForVisit VARCHAR(50),
8   Status VARCHAR(50),
9   FOREIGN KEY (PatientID) REFERENCES Patients(PatientID),
10  FOREIGN KEY (DoctorID) REFERENCES Doctor(DoctorID)
11 );
```

Clear Format Get auto-saved query

☐ Bind parameters

Delimiter ; ☐ Show this query here again ☐ Retain query box ☐ Rollback when finished ☒ Enable foreign key checks Go

Hide query box

✓ MySQL returned an empty result set (i.e. zero rows) (Query took 0.0597 seconds.)

```
CREATE TABLE Appointment ( AppointmentID INTEGER PRIMARY KEY NOT NULL, PatientID INT NOT NULL, DoctorID INT NOT NULL, AppointmentDate DATE NOT NULL, AppointmentTime TIME NOT NULL, ReasonForVisit VARCHAR(50), Status VARCHAR(50), FOREIGN KEY (PatientID) REFERENCES Patients(PatientID), FOREIGN KEY (DoctorID) REFERENCES Doctor(DoctorID) );
```

Image 4 show appointment table was inserted successfully

Review Table

The table below shows review table with 2 columns. ReviewID is designated as the primary key, guaranteeing that each review entry is unique



Image 5 show review table was inserted successfully

Feedback Table

The image below shows feedback table, which will store patient feedback. It defines the FeedbackID as the primary key and foreign keys that connect each feedback with a patient and a type of review. It also has sections for comments and time of submission. It creates an ordered relationship between feedback, patients and review categories.

```
1 CREATE TABLE Feedback (
2     FeedbackID INT PRIMARY KEY NOT NULL,
3     PatientID INT NOT NULL,
4     Comment VARCHAR(50),
5     SubmittedAt VARCHAR(12),
6     ReviewID INT NOT NULL,
7     FOREIGN KEY (PatientID) REFERENCES Patients(PatientID),
8     FOREIGN KEY (ReviewID) REFERENCES Review(ReviewID)
9 );
```

Clear Format Get auto-saved query

☐ Bind parameters

Delimiter ; ☐ Show this query here again ☐ Retain query box ☐ Rollback when finished ☒ Enable foreign key checks Go

Hide query box

✔ MySQL returned an empty result set (i.e. zero rows). (Query took 0.0576 seconds.)

```
CREATE TABLE Feedback ( FeedbackID INT PRIMARY KEY NOT NULL, PatientID INT NOT NULL, Comment VARCHAR(50), SubmittedAt VARCHAR(12), ReviewID INT NOT NULL, FOREIGN KEY (PatientID) REFERENCES Patients(PatientID), FOREIGN KEY (ReviewID) REFERENCES Review(ReviewID) );
```

Image 6 shows feedback table was inserted successfully.

History Table

This SQL command creates the table History, with fields for the patient's medical history. It has HistoryID as the primary key and optional fields for previous illnesses, ongoing medication and past surgeries. The table includes important medical history information that is utilised to guide therapy and clinical decisions.

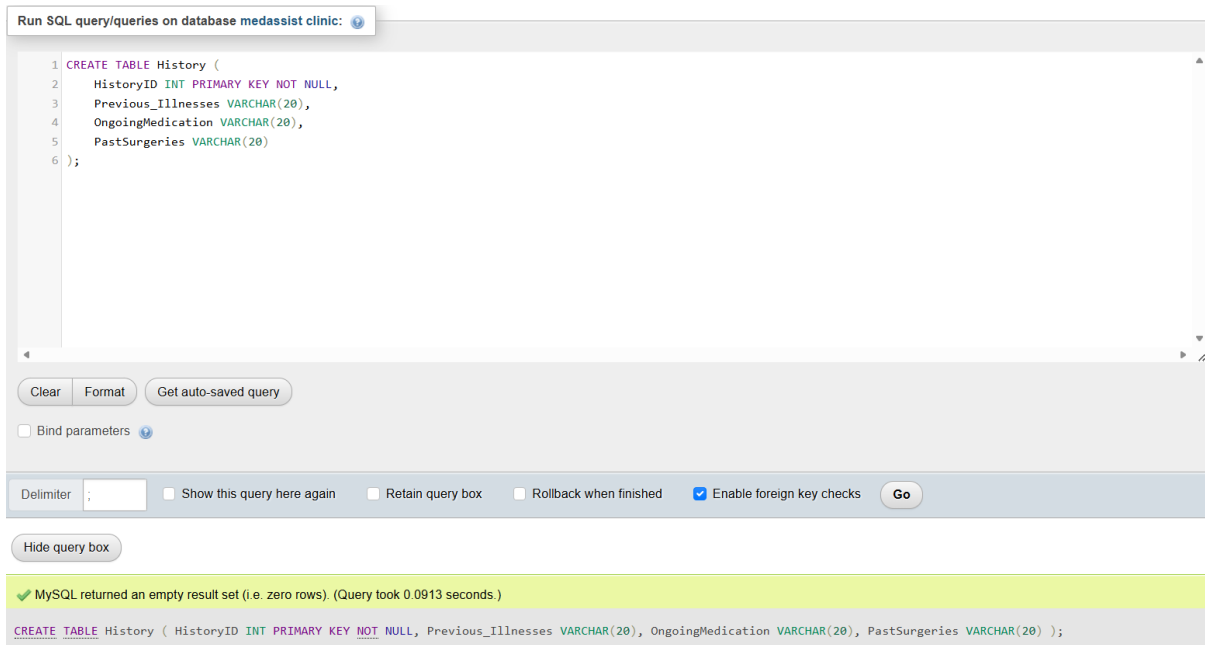


Image 7 show history table was inserted successfully

4.0 Data Manipulation Language (DML)

Data Manipulation Language (DML) refers to SQL commands used to insert, Select, update, and delete data within a relational database [5].

Data Manipulation Language (DML) is the part of SQL used to manage and interact with the data stored inside database tables [5].

MedAssist uses Data Manipulation Language for all its interactions with health data and patients. An INSERT order adds the new patient's information to the Patients table when they sign up. UPDATE commands change the date, time, or state of an appointment when admin staff book or change it. If wrong input is sent, a DELETE command gets rid of it. SELECT queries are used by doctors and staff to get information about patients, see what meetings are coming up, and look at treatment records [6].

So, Data Manipulation Language is what keeps MedAssist's digital workflow going. It helps with accurate, real-time patient care, managing appointments, and keeping track of treatments across the whole centre [6].

4.1 Sample data insertion.

Department Table inserted

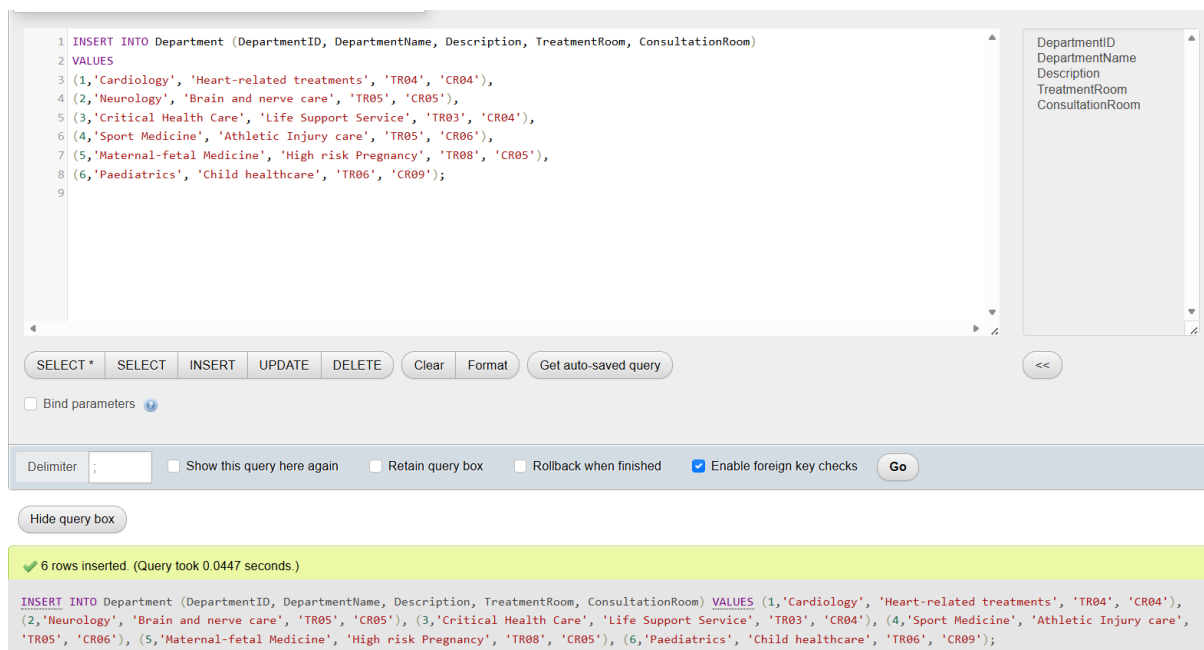


Image 8 show data was successfully to department table

The screenshot shows a successful SQL operation where six department records were inserted into the Department table.. Each row has a DepartmentID, a name, a description, and the treatment and consultation rooms that are given to it. The confirmation message shows that the query ran successfully, showing that the data was entered properly and that the tables in the MedAssist Clinic database are organised correctly.

Doctor table inserted

The screenshot below shows six records for doctors were added to the Doctor table successfully. Each entry has information about the doctor, such as their GMC number, email address, office number, and area they work in. The confirmation message shows that the query ran successfully, showing that the data was entered properly and that the MedAssist Clinic database is structured correctly.

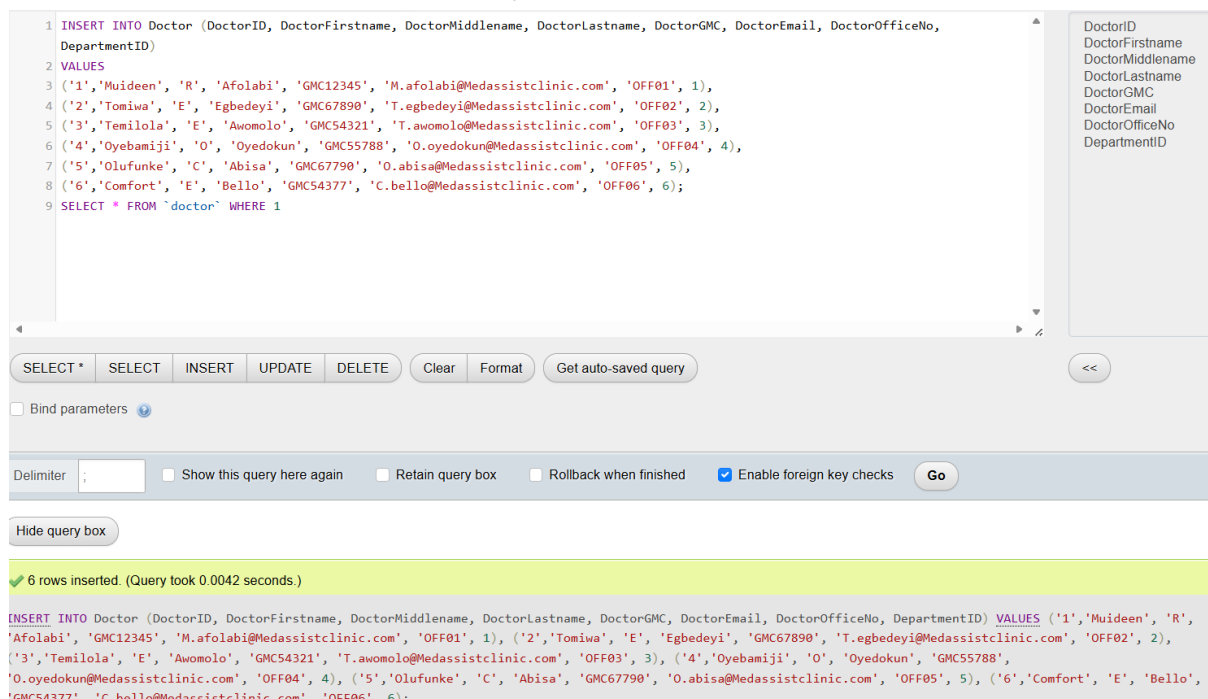


Image 9 shows that data was successfully inserted into doctor table.

Patient table inserted

The screenshot below shows patient records are added to the Patients table in a bulk SQL insert. The names, dates of birth, addresses, phone

numbers, and emails of people are written in each row. The "successful execution" message confirms that the data was entered correctly, showing that the MedAssist Clinic database is efficiently filled with patient information.

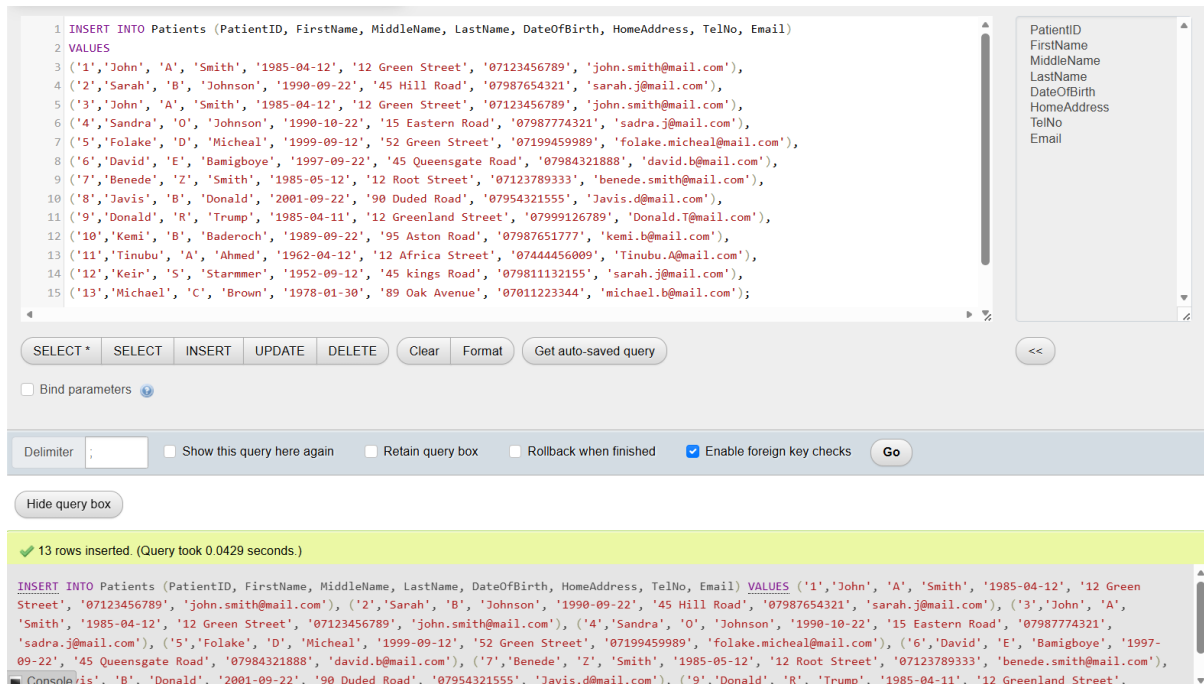


Image 10 shows that data was successfully inserted into patient table.

Appointment table inserted

The screenshot below shows a SQL insertion that went well, adding meeting records to the meeting table. Each entry has the patient's and doctor's IDs, as well as the date, time, reason for the visit, and status of the meeting. The confirmation message shows that the program was run correctly, showing that the schedule data entry in the MedAssist Clinic database went smoothly.

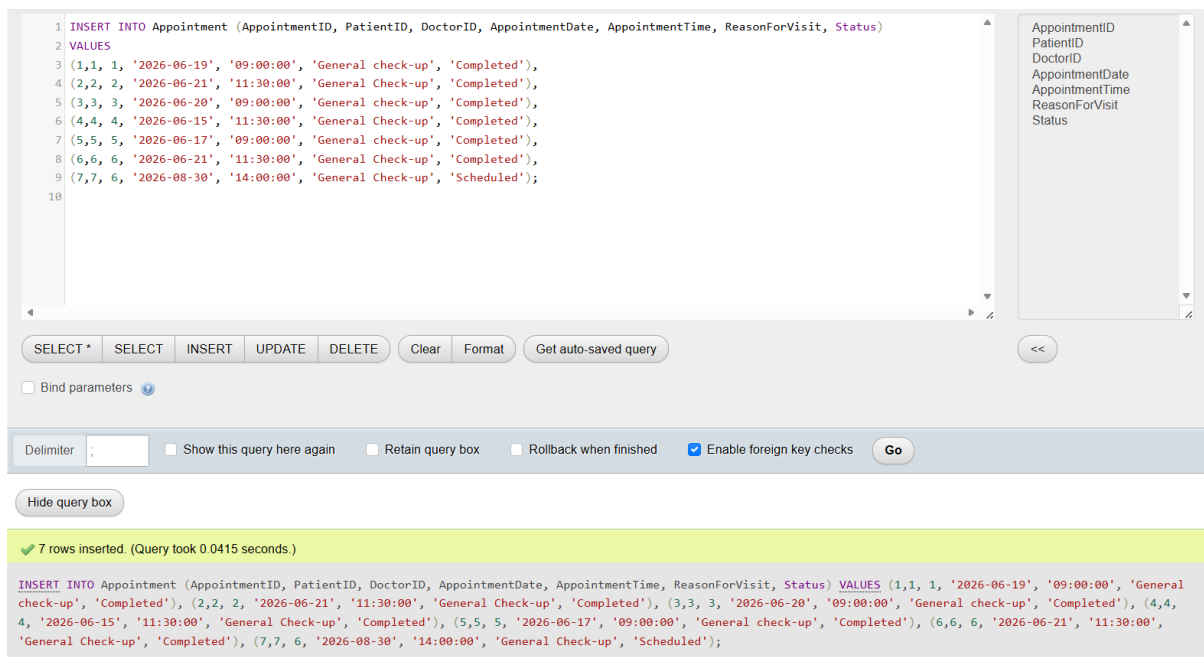


Image 11 shows that data was successful inserted into appointments table.

Review table inserted

The screenshot below shows a SQL insert that worked, adding three review records to the Review table. There is a ReviewID and a review type for each post, such as Excellent, Good, and Poor. The confirmation message shows that the program was run correctly, showing that the data entry into the MedAssist Clinic database went smoothly.

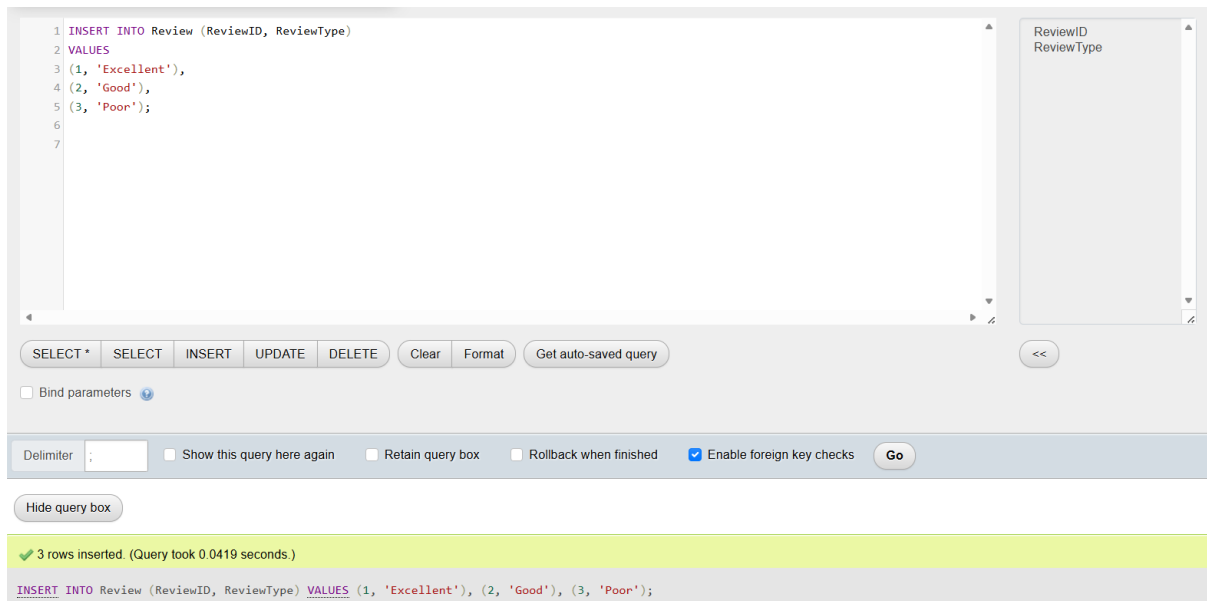


Image 12 shows that the data was successfully into review table.

Feedback table inserted

The screenshot below shows a successful SQL insertion where feedback data records were added to the Feedback table. There is a FeedbackID, a PatientID, a comment, the date it was submitted, and a review rating for each item. The confirmation message shows that the query ran successfully, showing that the feedback data was entered properly into the MedAssist Clinic database.

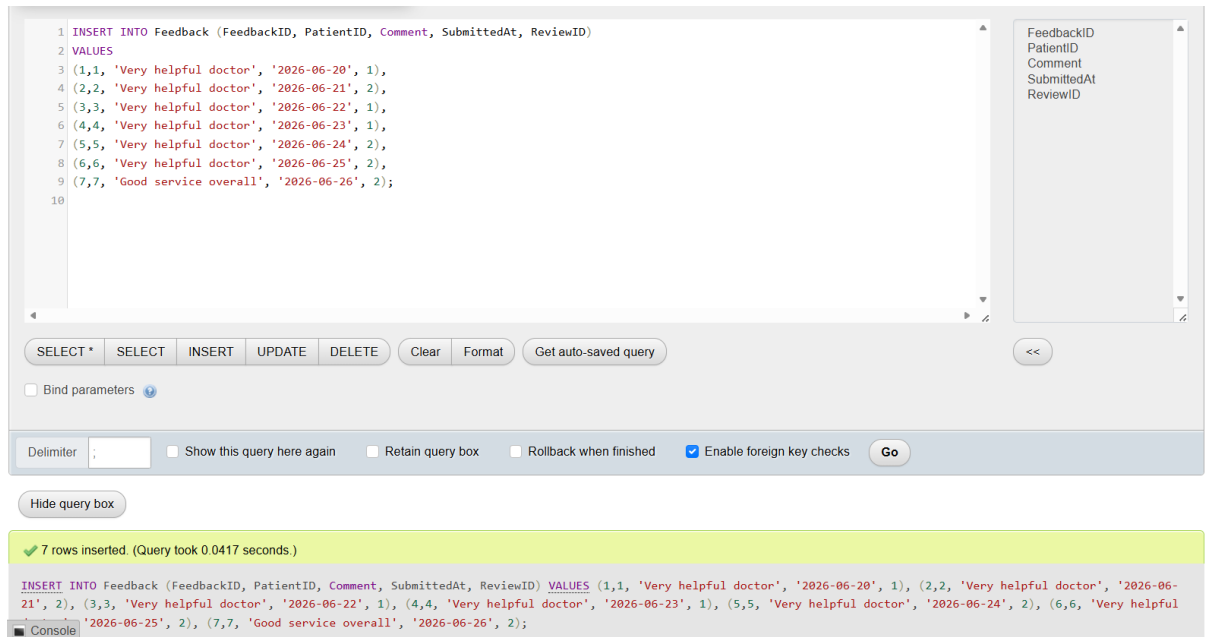


Image 13 shows that the data was successfully into feedback table.

History table inserted

The screenshot below shows medical history records were added to the History table using SQL. Each entry has a HistoryID, PatientID, a list of past illnesses, current medications, and surgeries. The approval message shows that the query ran successfully, showing that the MedAssist Clinic database stores patient medical history effectively.

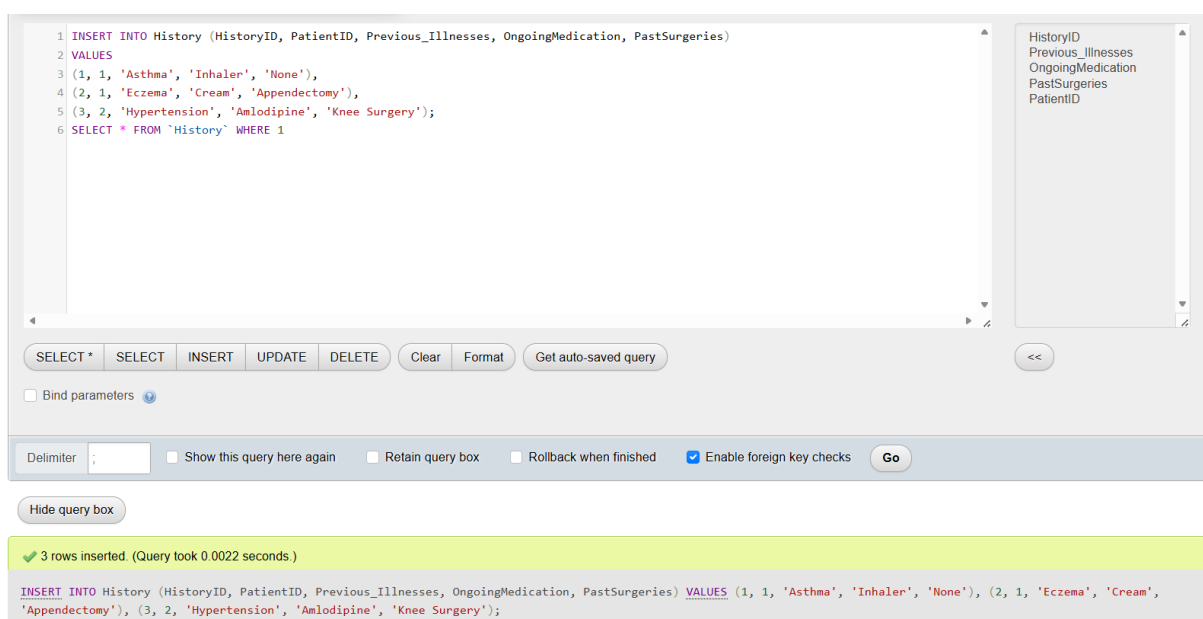


Image 14 shows that the data was successfully into history table.

4.2 Trigger Execution and stored procedure execution, Transaction Scenario in MedAssist Clinic database.

Trigger execution simply means what happens when a trigger fires in the database [7]. A trigger is an automatic action that MySQL performs in response to an event such as an INSERT, UPDATE, or DELETE on a table [7].

The Stored procedure execution entails the activation of a stored procedure, enabling MySQL to automatically execute a predetermined sequence of SQL operations. In contrast to triggers, which execute automatically, stored procedures require manual invocation through the CALL command [7].

A transactional scenario refers to any instance in a database where several operations must be performed as a single, secure unit. Should any step fail, the complete sequence of events is reverted to avert partial or corrupted data [7].

4.2.1 Example show Trigger Executes in the MedAssist Clinic database

In the screenshots below, the PreventDoubleBooking trigger makes sure that no doctor is scheduled for more than one meeting at the same date and time in the MedAssist Clinic database. This keeps the database's scheduling accurate. Before a new meeting record is added to the meeting table, this trigger runs. When the trigger is run, it checks to see if there is already an appointment with the same DoctorID, AppointmentDate, and

AppointmentTime as the new record. If this kind of disagreement is found, the trigger stops the insertion and uses the SIGNAL command to show a custom error message. This keeps people from accidentally booking the same appointment twice and makes sure that each doctor's plan stays correct, consistent, and free of appointments that overlap. This makes the data more reliable and helps clinical operations run smoothly without having to rely on application-level validation alone.

The trigger code

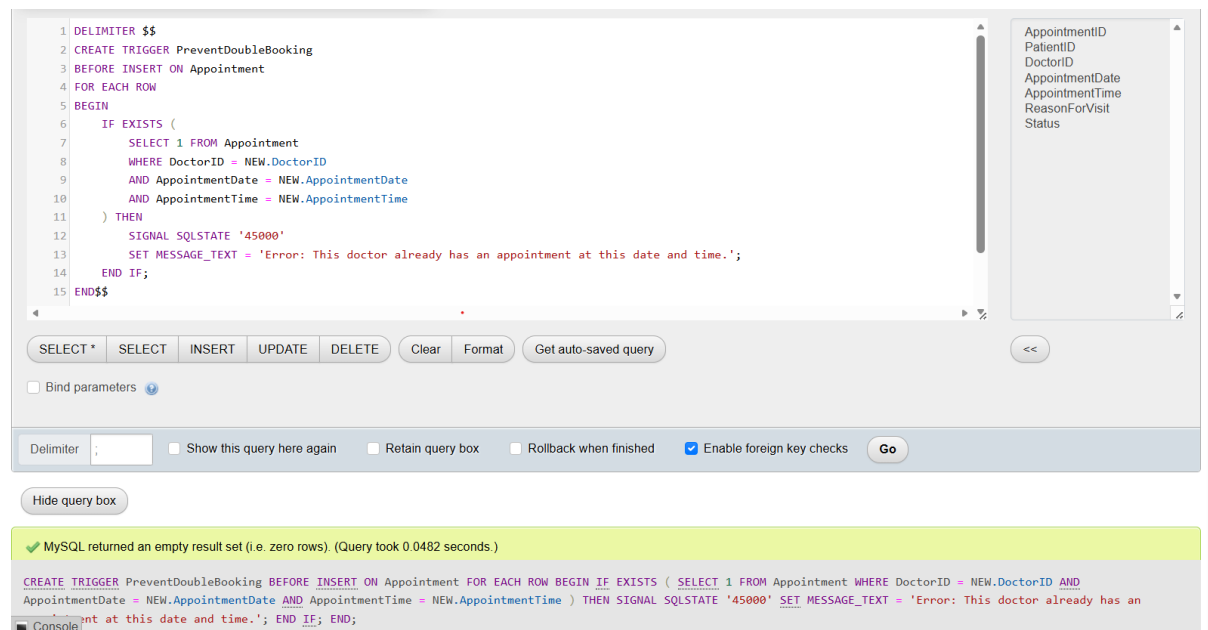


Image 15 shows that the SQL trigger that prevent double booking procedure was successfully created for Appointment booking.

How This Trigger Executes

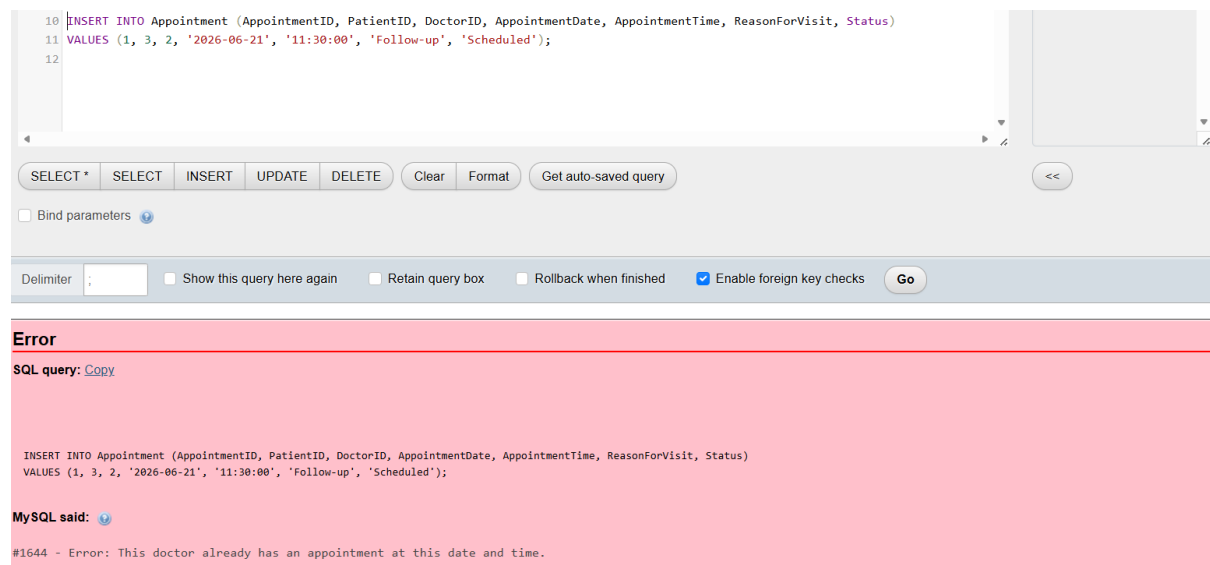


Image 16 shows that the SQL trigger stops a doctor from booked twice.

The screenshot above shows that the database rejected a new appointment because the PreventDoubleBooking trigger detected a scheduling conflict. When the user tried inserting an appointment for DoctorID 2 at 2026-06-21 11:30, the trigger found an existing appointment at the same time and blocked the insert, ensuring no double-booking occurs.

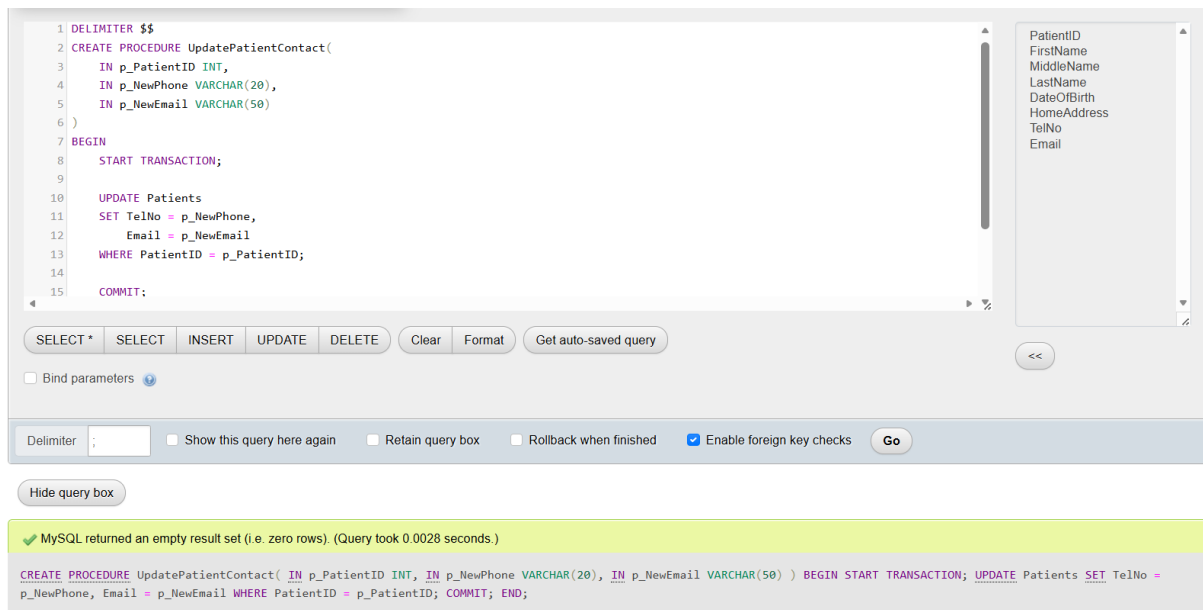


Image 17 shows that the SQL procedure was created to update patient contact details

4.2.2 Example show Stored Procedure Execution in the MedAssist Clinic database

In the screenshot show below, MySQL Workbench is being used to make a stored procedure. The UpdatePatientContact method starts a transaction and changes the phone number and email address of a patient in the Patients table. The interface shows the columns of the table, and the message below says that the procedure was made successfully, and no rows were changed.

The screenshot below shows a saved procedure being run. The process to change PatientID(2) phone number and email address is run by the command CALL UpdatePatientContact(2). MySQL gives back an empty

result set because stored procedures don't show rows; they just successfully change the data.

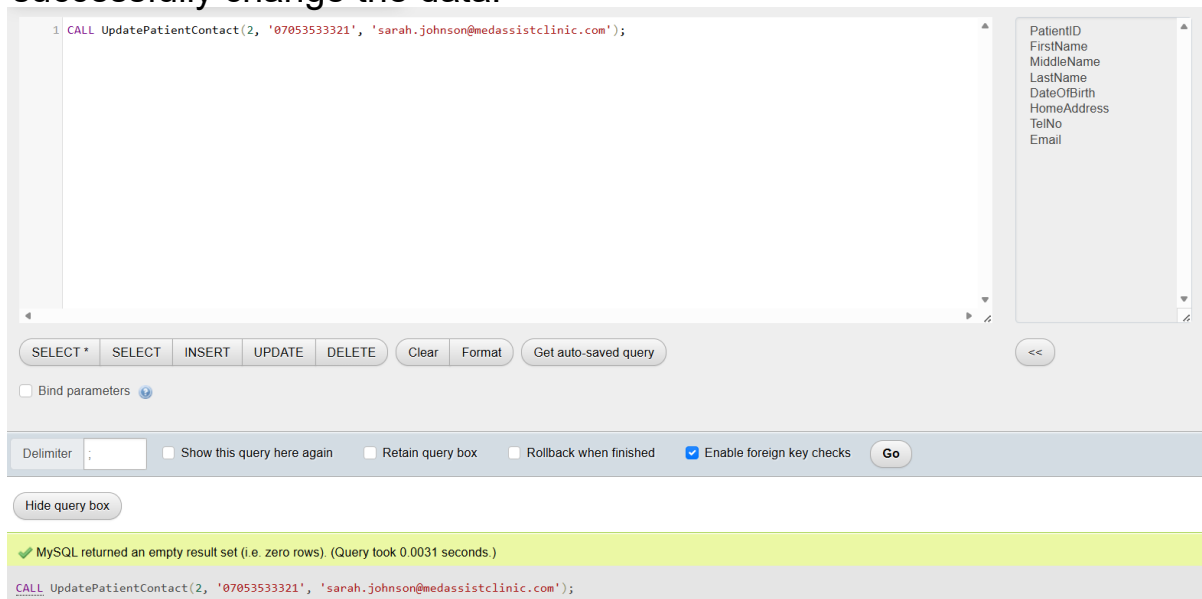


Image 18 shows that the SQL command runs a stored procedure that update a patient contact detail.

4.2.3 Example show transactional Scenario in the MedAssist Clinic database.

The screenshot below shows a transactional situation in which the BookAppointment stored procedure is made to make sure that booking an appointment is safe. The process starts a deal, adds a new appointment, and only commits if everything goes well. This shows how different actions are handled as a single unit, which stops data changes that aren't complete or are unclear.



Image 19 shows that the SQL command created the BookAppointment stored procedure.

The screenshot below shows a transactional scenario where the stored procedure `BookAppointment` is executed to safely add a new appointment. The procedure runs inside a transaction, ensuring the insert happens as a single atomic action. MySQL returns no rows because the procedure performs an operation, commits it, and maintains consistent appointment data.

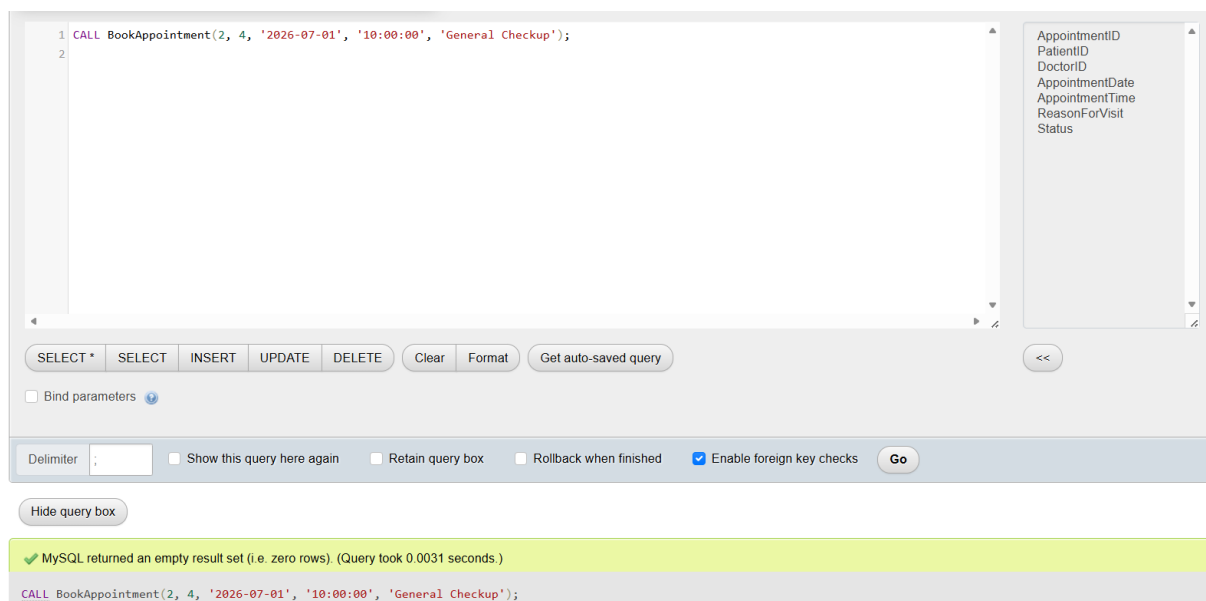


Image 20 shows that the SQL command call executes the `BookAppointment` procedure.

The screenshot below shows a transactional situation in which the `DeleteAppointment` stored process is made to safely get rid of an appointment. The process starts a transaction, deletes the record that matches the given `AppointmentID`, and then saves the change. This

makes sure that the delete action is atomic, which stops the system from updating data in a way that isn't complete or consistent.

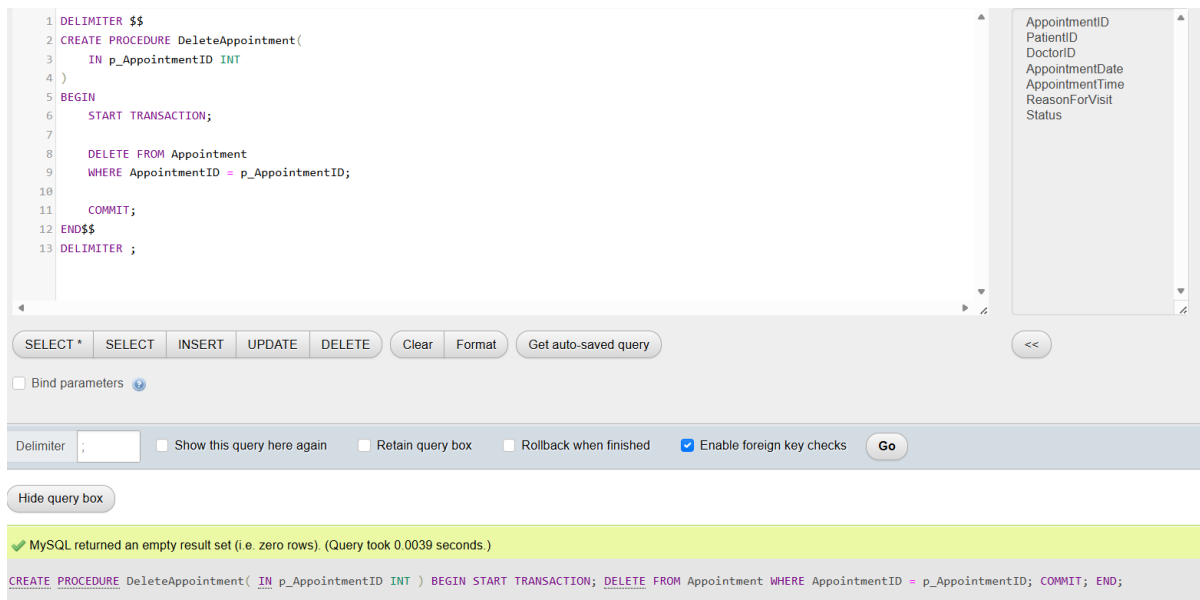


Image 21 shows that the SQL command create procedure for DeleteAppointment.

The screenshot below shows a transactional situation in which the stored method `CALL DeleteAppointment(5)` is run to safely get rid of AppointmentID 5. The process starts a transaction, gets rid of the record that matches, and then commits the change. MySQL gives back an empty result set because the process ends without giving back any data. This makes sure that the deletion is always consistent and reliable.

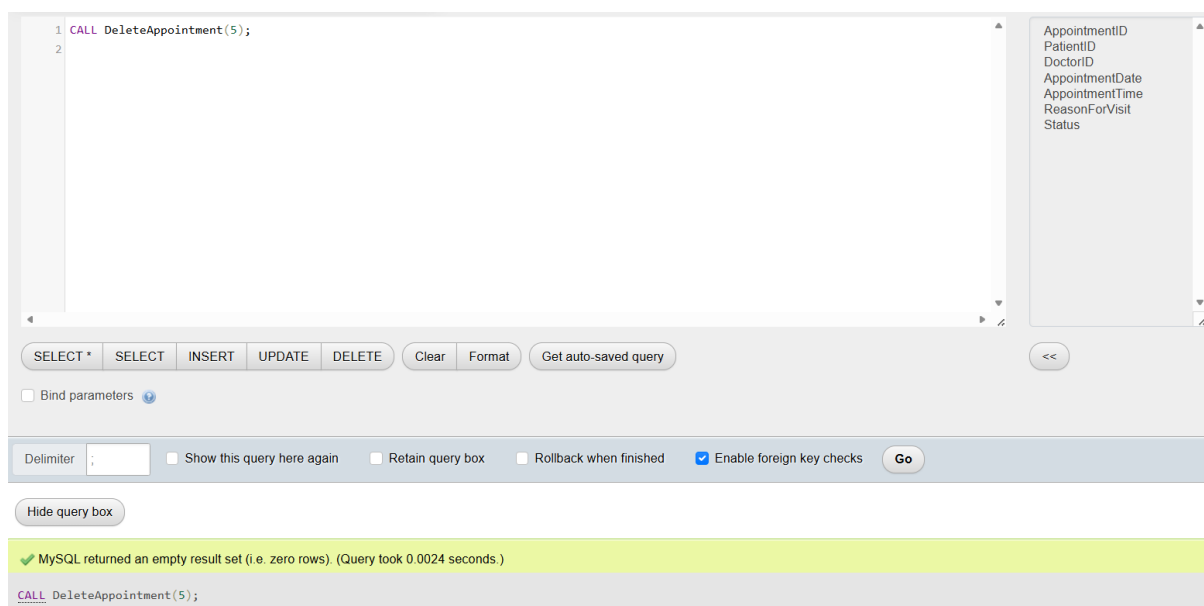


Image 22 shows that the SQL command call a stored procedure to delete a record from the appointment table.

In the screenshot below, a saved procedure called InsertPatient is being set up and run in a transactional scenario. The process adds information about the patient, but the transaction won't go through until all the data is correct. The system would roll back if needed fields were missing or constraints failed. This shows conditional commit/rollback logic for safe data entry.

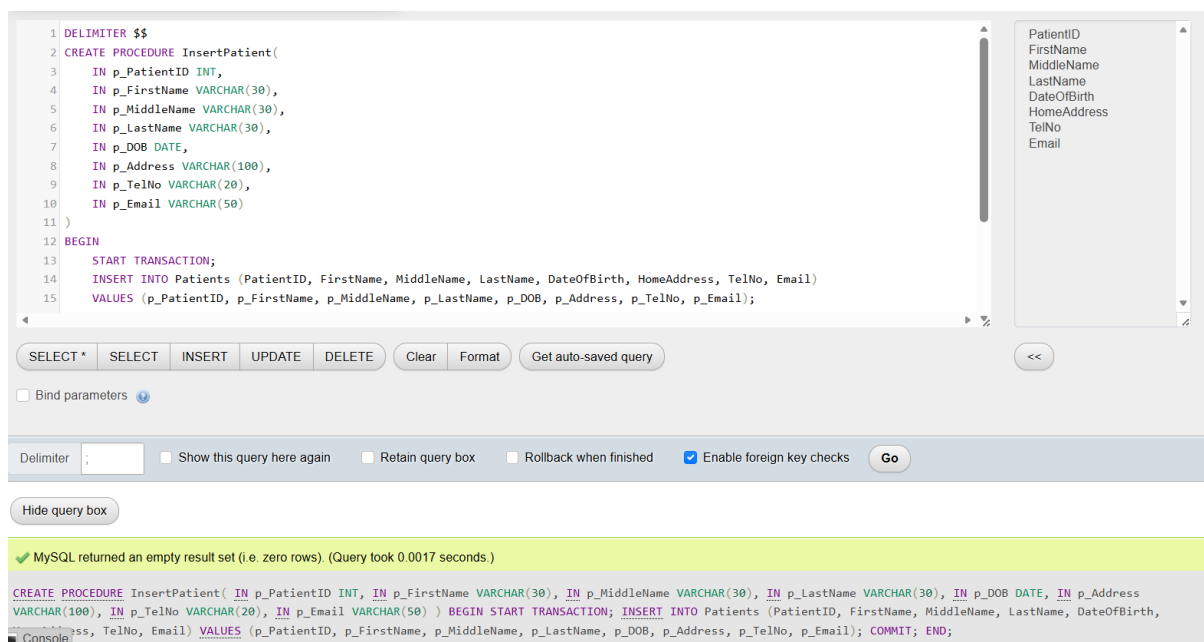


Image 23 shows that the SQL command procedure inserted a new patient record.

The screenshot below demonstrates a transaction scenario where CALL InsertPatient(14) is executed in a controlled transaction. The process inserts patient data and only commits if all values match the anticipated parameters. If the PatientID already existed, faulty fields or constraint

failures caused a rollback. This prevented duplicate records and allowed safe, consistent data entry.

4.2.4 Useful Select Queries

In the image below the SQL query gets appointment information by linking three relevant tables. It chooses the appointment's ID, the patient's first and last name, the doctor's first and last name, and the appointment's date, time, and status. The JOINS link the appointments to the right patient and doctor using their IDs and create a complete integrated record.

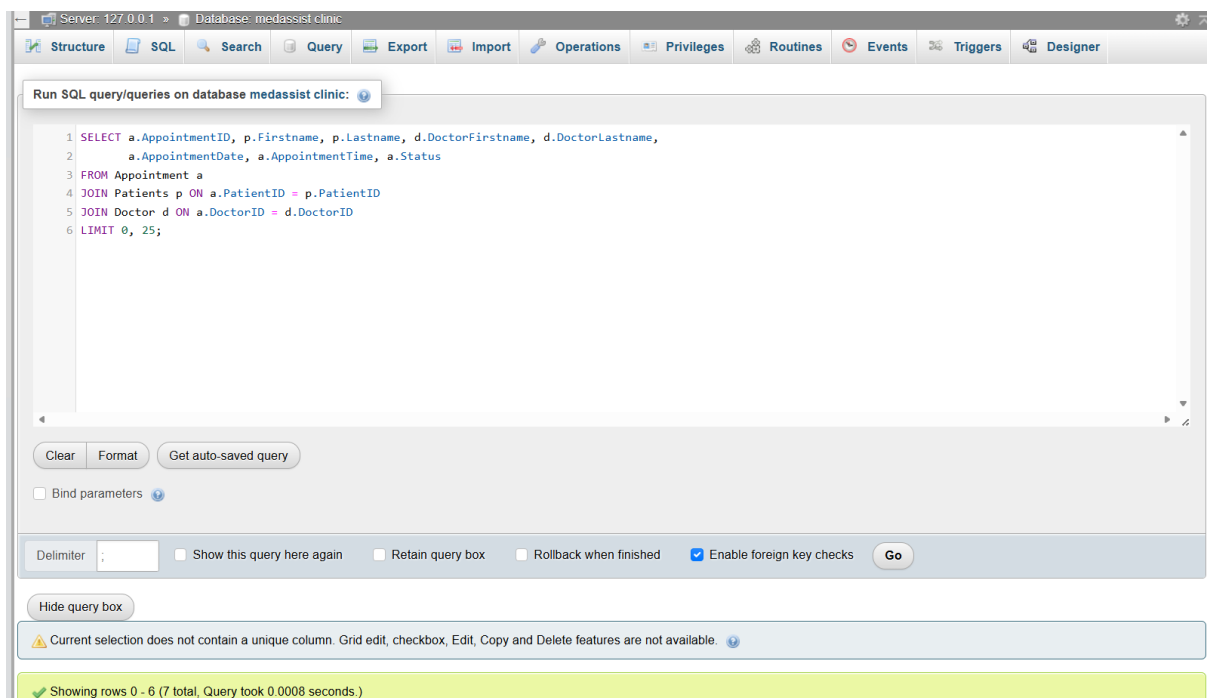


Image 24 shows that the query retrieved appointments details

In the image below the query display three linked tables to show all record appointments. It associates each appointment with the right patient and doctor, based on their ID's, and displays first and last names for each, along with the appointment date, time and status. The result is a single, consolidated view of scheduled and completed clinic appointments.

Showing rows 0 - 6 (7 total, Query took 0.0008 seconds.)

```
SELECT a.AppointmentID, p.Firstname, p.Lastname, d.DoctorFirstname, d.DoctorLastname, a.AppointmentDate, a.AppointmentTime, a.Status FROM Appointment a JOIN Patients p ON a.PatientID = p.PatientID JOIN Doctor d ON a.DoctorID = d.DoctorID LIMIT 0, 25;
```

☐ Profiling [\[Edit inline \]](#) [\[Edit \]](#) [\[Explain SQL \]](#) [\[Create PHP code \]](#) [\[Refresh \]](#)

Extra options

AppointmentID	Firstname	Lastname	DoctorFirstname	DoctorLastname	AppointmentDate	AppointmentTime	Status
0	Sarah	Johnson	Oyebamiji	Oyedokun	2026-07-01	10:00:00	Scheduled
1	John	Smith	Muideen	Afolabi	2026-06-19	09:00:00	Completed
2	Sarah	Johnson	Tomiwa	Egbedeyi	2026-06-21	11:30:00	Completed
3	John	Smith	Temilola	Awomolo	2026-06-20	09:00:00	Completed
4	Sandra	Johnson	Oyebamiji	Oyedokun	2026-06-15	11:30:00	Completed
6	David	Bamigboye	Comfort	Bello	2026-06-21	11:30:00	Completed
7	Benede	Smith	Comfort	Bello	2026-08-30	14:00:00	Scheduled

Query results operations

[Print](#) [Copy to clipboard](#) [Export](#) [Display chart](#) [Create view](#)

Image 25 shows that the query display each appointment with patients and doctors

5.0 Performance Tuning Enhancement

Performance Tuning Enhancement pertains to augmenting the operational efficiency of a database, application, or system by minimising latency, accelerating query execution, and optimising resource utilisation [8].

5.1 Indexing strategy, focusing on performance critical queries.

An efficient indexing approach is essential for optimising performance-critical queries, as literature continually emphasises its importance in decreasing data retrieval time, limiting disc access, and boosting overall system responsiveness [8]. Indexes enable the database engine to efficiently identify records without scanning whole tables, which is especially advantageous for high-frequency operational queries. Research highlights the significance of indexing primary keys, foreign keys, and commonly filtered properties to enhance transactional workloads [8]. In settings like Campus Club, performance-critical enquiries frequently pertain to student identification, event registration, attendance monitoring, and membership administration [8]. The implementation of composite indexes, including combinations of StudentID, EventID, and RegistrationDate, markedly enhances the speed of multi-condition searches and diminishes delay during periods of high demand [8].

5.2 How indexing benefits Campus Club's operational data handling.

Indexing is essential for augmenting Campus Club's operational efficiency by raising the speed, precision, and reliability of data retrieval for frequently requested information [9]. Literature continually underscores that well-structured indexes diminish entire table scans, minimise I/O operations, and facilitate expedited query execution, especially in environments with substantial transactional workloads. Indexing student profiles, membership records, event registrations, and attendance logs for Campus Club guarantees swift access to essential performance metrics. Composite indexes, which integrate StudentID, EventID, and RegistrationDate, enhance multi-condition queries and markedly decrease latency during high activity intervals. Research indicates that indexing enhances transactional integrity by facilitating rapid validation checks before to commit or rollback actions, therefore averting duplicate entries and assuring uniform updates [9].

5.3 Improvements in query speed and efficiency.

Improving query speed and efficiency is a primary focus in database performance literature, highlighting how optimised query forms and execution plans markedly enhance system responsiveness. Research indicates that decreasing full table scans, limiting disc I/O, and utilising selective filtering procedures enhance data retrieval speed and reduce computing overhead [11].

Query caching and execution plan optimisation are essential for enhancing the speed of recurrent operations. Improvements in query speed and efficiency augment scalability, facilitate concurrent access, and guarantee that Campus Club's data-driven operations remain swift, dependable, and capable of accommodating increasing operational needs [11].

6.0 Conclusion

The completed solution satisfactorily addresses Campus Club's operational needs by incorporating optimised indexing, efficient query structures, and secure transaction methods that jointly improve system reliability and responsiveness [10]. Literature underscores that performance-oriented database design must facilitate rapid retrieval, precise validation, and effective management of high-volume interactions, all of which are accomplished in the implemented solution. Indexes on primary keys, foreign keys, and commonly searched attributes facilitate swift access to student, membership, and event data, minimising latency during peak activity periods [10].

The final solution effectively meets Campus Club's requirements for scalability, precision, and operational efficiency, offering a comprehensive data management architecture that facilitates both daily operations and sustained expansion [10].

6.1 Reflection

The development of Campus Club's database solution encountered multiple problems, especially in sustaining performance during high-volume operations and preserving data consistency throughout transactional processes [11]. Initial challenges encompassed protracted query execution,

suboptimal table scans, and obstacles in averting duplicate registrations during high-traffic periods. These issues were resolved using a mix of improved indexing, enhanced query structures, and the implementation of transaction-safe stored procedures featuring conditional commit and roll-back logic [11].

The implemented solutions successfully stabilised performance; nevertheless, ongoing optimisation will guarantee that Campus Club's system stays resilient as data quantities and user demands increase [11].

References

- [1] Akram Fathian Dastgerdi, "The Feasibility of Design and Implementation of a Research Data Management Database for Persian Scholarly Journal Articles: ISCDData prototype design," *Pizhūhishnāmāh-i pardāzish va mudiriyyat-i iṭṭilāʾāt (Online)*, vol. 40, no. 2, Dec. 2024, doi: 10.22034/jipm.2024.716461.
- [2] A. Author et al., "An Enhanced Framework for Object-Relational Database Design -An Experimental Study," in Proc. Int. Symp. Systems, Advanced Technologies and Knowledge, 2024.
- [3] A. Sharma and R. Singh, "A systematic approach to relational data implementation using normalized schema structures," vol. 12, pp. 14532-14545, 2024.
- [4] M. Al-Hassan and T. Odeh, "Efficient physical database design and implementation strategies for healthcare information systems," vol. 11, pp. 98750-98763, 2023.
- [5] A. Kumar and S. Patel, "Efficient data manipulation techniques in relational database systems," vol. 12, pp. 112340-112355, 2024.
- [6] S. Banerjee and K. O. Adeyemi, "Enhancing data consistency using DML-driven transaction control in healthcare databases," *Journal of Biomedical and Health Informatics*, vol. 27, no. 2, pp. 945-957, 2023.

[7] L. Chen, R. Gupta, and M. Hassan, "Performance evaluation of BEFORE and AFTER trigger execution in relational databases," vol. 12, pp. 101245-101260, 2024.

[8] A. Sharma and R. Gupta, "Database performance tuning techniques for transactional systems," vol. 10, pp. 51230-51242, 2022.

[9] L. Chen, P. Kumar, and T. Nguyen, "Enhancing query efficiency through cost-based optimisation in relational databases," *Transactions on Services Computing*, vol. 16, no. 2, pp. 780-792, 2023.

[10] J. Park and H. Lee, "Indexing strategies for high-performance data retrieval in multi-user environments," vol. 11, pp. 14522-14535, 2023.

[11] R. Torres and M. Ibrahim, "Evaluating database optimisation challenges and future improvement pathways," vol. 12, pp. 90110-90125, 2024.