CS 432
Databases

Assignment II
Developing the DBMS

QUERY CRAFTERS

Group Members

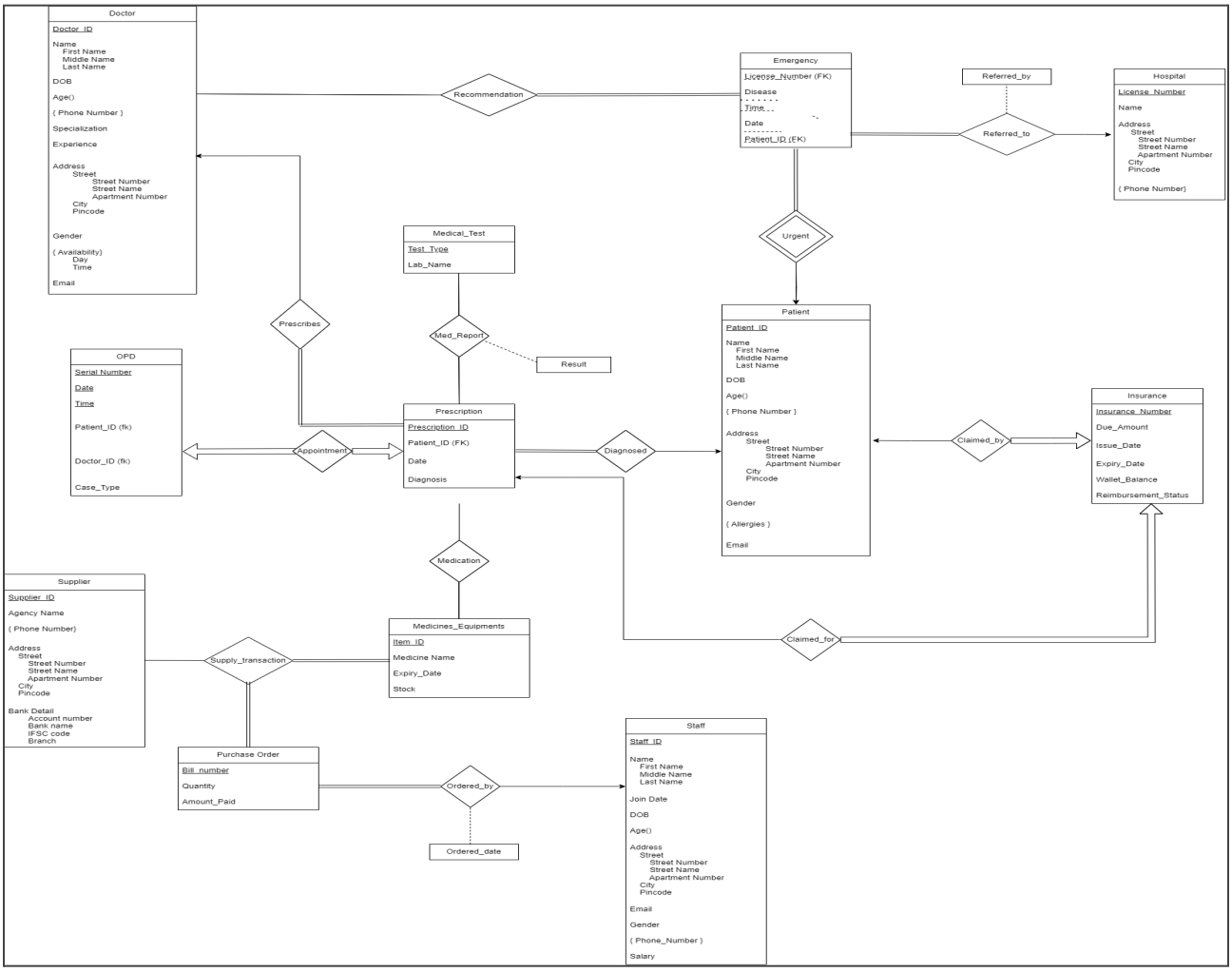
Saurabh Kumar Sah	21110188
Shreya Patel	21110155
Riya Jain	21110178
Twinkle Devda	21110228
Ishika Raj	21110081
Saumya Jaiswal	21110186
Darsh Dalal	21110049
Het Trivedi	21110226

February 15th, 2024

We have revisited Assignment 1 and implemented a few modifications to our database schema. These adjustments were made not necessarily to rectify errors but also to minimize redundancy. The alterations include:

- Removed foreign key attributes from entity set schemas wherever necessary, eliminating redundancy introduced by existing relationship sets.
- Eliminated the prescription item table, as the relationship between prescription and medication already accommodates multivalued item_id.
- Omitted first name, last name, and middle name from relationship sets where feasible, recognizing that these details can often be inferred from unique corresponding IDs.
- Added date and time of OPD as primary keys in the appointment set to ensure uniqueness and facilitate efficient retrieval of appointment data.
- Removed test_date as a descriptive attribute from Med_Report due to the complexity of synchronizing data with prescription dates.
- Defined the relationship between Insurance and Prescription (Claimed_for) as many-to-one rather than one-to-one to accurately reflect the real-world scenario.

Following is the updated ER Diagram: [ERDiagram]



3.1 Responsibility of G1:

3.1.1] Population of tables with random data with the mentioned constraints and the ACID properties.

We utilized the Faker library in Python to generate data for most of our database tables. During this process, we paid close attention to various constraints, such as ensuring data integrity with constraints like not null, unique and checks for entity sets tables, as well as considering cardinality and participation constraints for relationship sets. After generating the data, we imported it into SQL for further use. You can find the SQL dump file, tables in CSV format, and the Python code used for data generation through the provided drive link.

GOOGLE DRIVE LINK

ACID Properties on MYSQL

Atomicity:

How does mysql maintains atomicity:

MySQL follows the atomicity principle, which guarantees that a set of modifications to the database either fully takes effect or is undone. This procedure requires careful monitoring of changes because MySQL keeps a separate log to document changes prior to their being reflected in the database itself. These changes are first saved in an in-memory buffer rather than instantly changing the database. Until the user explicitly confirms or "commits" the transaction, which signifies their permanence, the commitment to apply these changes is postponed. Importantly, MySQL ensures an "all or nothing" policy, meaning that changes are either permanently incorporated into the database or not at all. Atomicity for each entity in our "Dispensary" database: In each case, atomicity helps to maintain the integrity of the database by ensuring that changes to related attributes within an entity are either fully applied or not applied at all, avoiding partial or inconsistent updates.

Atomicity for each of the entities in dispensary database:

1. **Doctor**:
    - **Atomicity**: In the context of doctors, atomicity ensures that a medical transaction involving a doctor (such as updating their specialization or availability) either occurs entirely or not at all.
    - **For example** : if a doctor's availability changes from Monday mornings to Tuesday evenings, the entire update must be executed without partial changes. This guarantees consistency in the database.
2. **Patient**:
    - **Atomicity**: When dealing with patient records, atomicity ensures that any modification (like updating allergies or contact information) happens as a complete unit.
    - **For example :** if a patient's phone number changes, the entire update (including the new number and other details) should be applied together to maintain data integrity.
3. **Medical_Test**:
    - **Atomicity**: Medical test records must be treated as indivisible units. Either the entire test entry (including test type and lab name) is added or none of it is.
    - Ensuring atomicity prevents incomplete or inconsistent test data from being stored in the database.
4. **Emergency**:
    - **Atomicity**: Emergency records, being weak entities, rely on atomicity. If an emergency entry (such as disease diagnosis and time) is created, it must be inserted completely or not at all.

- This avoids situations where only partial emergency details are recorded, which could lead to incorrect patient management.

5. **Hospital**:
   - **Atomicity**: Hospital data, including license numbers, addresses, and phone numbers, must be updated atomically. Changes to hospital details should occur as a whole.
   - **For example :** if a hospital relocates, the entire address update (street, city, and pincode) should be applied together.
6. **Insurance**:
   - **Atomicity**: Insurance records (such as issue date, wallet balance, and reimbursement status) should be treated as indivisible units.
   - **For example :** Whether an insurance policy is issued or not, the entire transaction (including patient ID and expiry date) must be committed or aborted.
7. **Prescription**:
   - **Atomicity**: Prescription entries (including diagnosis and item ID) should be handled atomically. Either the entire prescription is recorded or none of it.
   - This ensures that incomplete or inconsistent prescriptions are not stored in the database.
8. **OPD (Outpatient Department)**:
   - **Atomicity**: OPD records (such as serial numbers, patient names, and case types) must be treated as indivisible units.
   - **For example :** Whether a patient visits the OPD or not, the entire transaction (including doctor names and appointment time) should be executed or rolled back.
9. **Supplier**:
   - **Atomicity**: Supplier data (like agency names, phone numbers, and bank details) should be updated atomically.
   - **For example :** if a supplier's bank account changes, the entire update (including IFSC code and branch) should be applied together.
10. **Medicines_Equipments**:
    - **Atomicity**: Records related to medicines and equipment (such as item IDs, composition, and stock) should be treated as indivisible units.
    - **For example :** Whether a new medicine is added or stock levels are updated, the entire transaction must be committed or aborted.
11. **Purchase_Order**:
    - **Atomicity**: Purchase orders (including bill numbers, item IDs, and quantities) should be handled atomically.
    - **For example :** Whether an order is placed or not, the entire transaction (including supplier IDs and amount paid) must be executed or rolled back.
12. **Staff**:
    - **Atomicity**: Staff records (such as join dates, salaries, and contact details) should be updated atomically.
    - **For example :** if a staff member's address changes, the entire update (including street, city, and pincode) should be applied together.

Consistency:

How does MySQL maintain consistency?

MySQL makes sure the database is consistent by enforcing atomicity, upholding integrity constraints, guaranteeing ACID compliance, offering many levels of isolation, maintaining transaction logs.

Primary keys, foreign keys, unique constraints, and check constraints are few constraints that can be defined in MySQL.

Unique constraints guarantee that, for every row in a table, the values in a given column or set of columns are unique.

Usage example: email address VARCHAR(100) UNIQUE.
Check constraints are applied to a condition that needs to be true for every table row.
Usage example: Age INT CHECK (age >= 18)
A foreign key constraint creates referential integrity between tables, while a primary key constraint guarantees that every row in a table may be uniquely identified.

The isolation level also included in consistency, determines the degree to which one transaction is isolated from the effects of other concurrently running transactions. A higher isolation level provides a greater degree of consistency but may impact performance.
MySQL uses transaction logs to record changes made by transactions to ensure atomicity.
Consistency for each of the entities in the dispensary database:

1. **Doctor:**
   Consistency for the Doctor entity involves maintaining accurate details about their specialization, experience, and availability. Ensuring that the doctor's information is up-to-date and corresponds correctly with their unique ID fosters reliable healthcare services and scheduling.
2. **Patient:**
   Patient consistency revolves around maintaining accurate health records, including allergies and contact details. It ensures that patient information is current and reliable for effective treatment planning and follow-up care.
3. **Medical_Test:**
   Consistency in medical tests involves tracking test types and associated lab names accurately. This supports reliable analysis and reporting of test results, contributing to patient diagnosis and treatment.
4. **Emergency:**
   Consistency in emergencies ensures that patient names and related details are accurately linked to emergencies. It supports timely and accurate responses during critical situations and emergency care.
5. **Hospital:**
   Hospital consistency focuses on maintaining accurate details, such as license numbers and addresses, ensuring the legitimacy of healthcare institutions and reliable healthcare services.
6. **Insurance:**
   Consistency in insurance involves tracking patient insurance details accurately. It ensures reliable reimbursement processing, wallet balance management, and status tracking.
7. **Prescription:**
   Prescription consistency ensures that prescribed medications and diagnoses are accurately linked to patients. This supports effective treatment plans and medication management.
8. **OPD:**
   OPD consistency involves maintaining accurate records of patient and doctor names, ensuring correct case type categorization. It contributes to efficient outpatient department management and scheduling.
9. **Supplier:**
   Supplier consistency revolves around maintaining accurate supplier details, including contact information and bank details. It ensures reliable procurement and inventory management for medical supplies.
10. **Medicines_Equipments:**
    Consistency in medicines and equipment involves accurate tracking of item details, including expiry dates and stock levels. This guarantees the availability of safe and effective medical

resources.

11. **Purchase_Order:**
    Purchase order consistency ensures accurate records of transactions, including bill numbers and quantities. It supports reliable inventory management and financial tracking for efficient procurement processes.
12. **Staff:**
    Staff consistency involves maintaining accurate employee details, including contact information and salaries. It ensures effective human resource management and payroll processing for healthcare staff.

Isolation:

How does MySQL maintain isolation?

MySQL uses different levels of isolation to maintain data consistency during transactions. These levels strike a balance between consistency and performance. Let's break them down:

READ UNCOMMITTED: In this level, transactions can see changes made by other transactions even before they are officially committed. While it provides high performance, it can lead to inconsistencies if multiple transactions modify the same data simultaneously.
READ COMMITTED: Here, transactions only see changes that have been fully committed by other transactions. It ensures a higher level of consistency but still allows for concurrent access.
REPEATABLE READ: This level guarantees that a transaction sees a consistent snapshot of the database throughout its execution. Other transactions can't modify data that has already been read by the current transaction.
SERIALIZABLE: At the highest level, transactions are executed sequentially, one after the other. This eliminates concurrency entirely, ensuring maximum consistency but potentially impacting performance.

Isolation for each entity in our "Dispensary" database:

1. **Doctor**:
    - **Isolation**: Doctors' records should be isolated to prevent conflicts. When one doctor updates their specialization or availability, other concurrent transactions shouldn't see partial changes. Isolation ensures that a doctor's data remains consistent during simultaneous interactions.
    - **Example**: If Dr. Smith's availability changes from Monday mornings to Tuesday evenings, other transactions should not observe an inconsistent state where both schedules coexist.
2. **Patient**:
    - **Isolation**: Patient data must be isolated to maintain privacy and consistency. When patients update their allergies or contact details, other transactions should not read outdated or conflicting information.
    - **Example**: If Patient A changes their phone number, Patient B's concurrent query should not retrieve the old number.
3. **Medical_Test**:
    - **Isolation**: Test types and lab names should remain consistent during concurrent transactions. If a new test type is added, other transactions should not see it until it is committed.
    - **Example**: If Lab X adds a COVID-19 test, Lab Y's queries should not accidentally include it before Lab X commits the change.

4. **Emergency**:
    - **Isolation**: Emergency records (e.g., disease, date, time) should be isolated to avoid inconsistencies. Other transactions should not see partial or outdated data when an emergency entry is created or modified.
    - **Example**: If an ambulance logs an emergency for Patient C, other ambulances should not see this entry until it's fully recorded.
5. **Hospital**:
    - **Isolation**: Hospital details (license number, address) must remain consistent. When a hospital's address changes, other transactions should not read the old or new address until committed.
    - **Example**: If Hospital Z relocates, concurrent queries should not show old and new addresses.
6. **Insurance**:
    - **Isolation**: Insurance records (e.g., issue date, wallet balance) should be isolated. Other transactions should not see intermediate states when an insurance policy is issued or updated.
    - **Example**: If money is deducted from the Patient's insurance wallet, other transactions should not observe the renewal halfway through.
7. **Prescription**:
    - **Isolation**: Prescription data (e.g., diagnosis, item ID) should be isolated. Other transactions should not read inconsistent data when a prescription is created or modified.
    - **Example**: If Dr. Johnson prescribes medication to Patient E, other queries should not see an incomplete prescription.
8. **OPD (Outpatient Department)**:
    - **Isolation**: OPD records (serial number, patient-doctor names) must be isolated. When appointments are scheduled or rescheduled, other transactions should not observe partial changes.
    - **Example**: If the Patient's appointment time shifts, other concurrent queries should not see old and new times.
9. **Supplier**:
    - **Isolation**: Supplier data (e.g., agency name, bank details) must be isolated. Other transactions should not read inconsistent information when a supplier's bank account changes.
    - **Example**: If the Supplier updates their IFSC code, other queries should not see the old code.
10. **Medicines_Equipments**:
    - **Isolation**: Equipment details (item ID, stock) must be isolated. Other transactions should not see intermediate states when stock levels change due to purchases or usage.
    - **Example**: Other queries should not observe an incomplete update if Equipment H's stock decreases.
11. **Purchase_Order**:
    - **Isolation**: Purchase order data (bill number, quantities) should be isolated. Other transactions should not read inconsistent quantities when an order is placed or modified.
    - **Example**: If a new order is placed for item I, other queries should not see it until it is fully recorded.
12. **Staff**:
    - **Isolation**: Staff records (e.g., join date, salary) must be isolated. Other transactions should not read outdated or conflicting data when staff details change.
    - **Example**: If Staff J's salary increases, other queries should not see the old salary.

Durability:

How does MySQL maintain durability?

Durability is a critical aspect of database management. It ensures that once you commit changes (like inserting new data or updating existing records), those changes are permanently saved and won't vanish, even if something goes wrong with the system.
Here's how MySQL achieves durability:
Transaction Logs: When you commit a transaction, MySQL writes down all the changes made by that transaction in a special log called the "transaction log." Think of it as a detailed record of what happened during the transaction.
Write-Ahead Logging: Before applying the changes to the actual database files, MySQL ensures they are safely recorded in the transaction log. This step is like double-checking: even if the database server crashes suddenly, your changes are still safe in the log.
Recovery Process: Imagine the database server unexpectedly crashes (it happens!). During recovery or system restart, MySQL uses the transaction logs to replay all the committed transactions. It's like following a recipe to recreate the exact state of the database before the crash.
No Lost Commitments: Thanks to this process, no committed transactions are lost or forgotten. Your data remains intact, and the system gets back to a consistent state.

Durability for each entity in your "Dispensary" database:

1. **Doctor**:
    - **Durability**: For doctors, durability ensures that once their information (such as specialization, experience, and availability) is updated or added, it remains intact even in the face of system failures. Whether a doctor's schedule changes or their contact details are modified, the database guarantees that these changes persist permanently.
    - **Example**: If Dr. Patel's phone number is updated due to a system crash, the new number should remain saved after recovery.
2. **Patient**:
    - **Durability**: Patient data, including allergies, contact details, and addresses, must persist reliably. The changes made during patient registration or updates should survive even if the system crashes.
    - **Example**: If Patient Smith's address is modified, it should remain consistent even after a power outage.
3. **Medical_Test**:
    - **Durability**: Test types and lab names should endure. Once a new test type is added or lab details are updated, this information remains unaffected by system failures.
    - **Example**: If a COVID-19 test type is added, it should persist even during catastrophic events.
4. **Emergency**:
    - **Durability**: Emergency records (e.g., disease, date, time) must survive. Whether an emergency entry is created or modified, it remains intact even if the system crashes.
    - **Example**: If an ambulance logs an emergency for Patient Lee, this record should persist beyond system failures.
5. **Hospital**:
    - **Durability**: Hospital details (license number, address) should be durable. Changes due to relocation or updates must persist, ensuring consistency.
    - **Example**: If Hospital Mercy changes its address, the new address should survive power outages.
6. **Insurance**:
    - **Durability**: Insurance records (e.g., issue date, wallet balance) must persist. Whether an insurance policy is issued or renewed, the data remains reliable.

- **Example**: If Patient Johnson's insurance is renewed, this information should survive system crashes.

7. **Prescription**:
   - **Durability**: Prescription data (e.g., diagnosis, item ID) should endure. Once a prescription is committed, it remains intact even during unexpected failures.
   - **Example**: If Dr. Brown prescribes medication, this prescription should persist beyond system crashes.
8. **OPD (Outpatient Department)**:
   - **Durability**: OPD records (serial number, patient-doctor names) must survive. Appointments, rescheduling, or case types should persist reliably.
   - **Example**: If Patient Adams' appointment time changes, this modification should endure system failures.
9. **Supplier**:
   - **Durability**: Supplier data (e.g., agency name, bank details) should persist. Even if the system crashes, supplier information remains reliable.
   - **Example**: If Supplier XYZ updates their bank account, this change should survive unexpected events.
10. **Medicines_Equipments**:
    - **Durability**: Equipment details (item ID, stock) must endure. Whether due to purchases or usage, stock levels should persist reliably.
    - **Example**: If Equipment Alpha's stock decreases, this information remains intact during system failures.
11. **Purchase_Order**:
    - **Durability**: Purchase order data (bill number, quantities) should persist. Whether an order is placed or modified, it remains reliable.
    - **Example**: This record should survive system crashes if a new order is placed for Item Beta.
12. **Staff**:
    - **Durability**: Staff records (e.g., join date, salary) must endure. Changes due to salary updates or staff additions should persist.
    - **Example**: If Staff Garcia's salary increases, this information remains reliable even during power outages.

3.1.2] Indexing

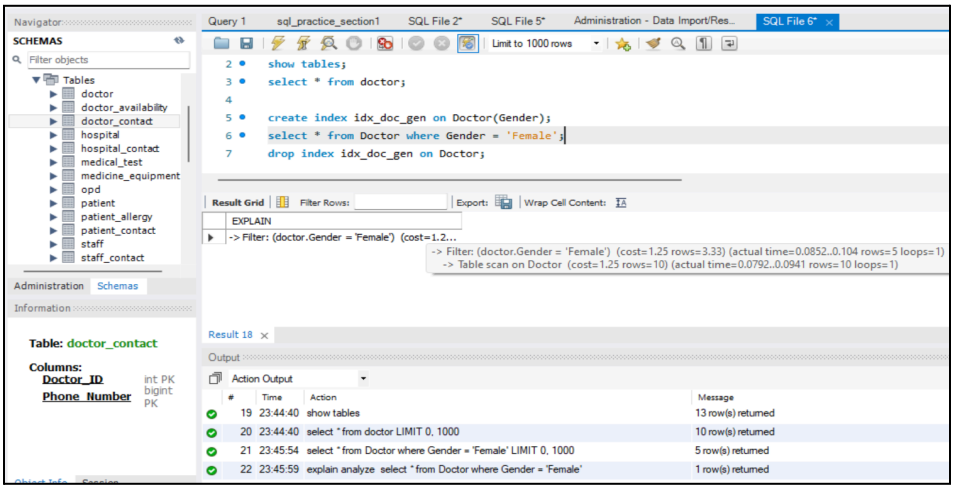
The process of building an index to speed up database retrieval is called indexing. It involves generating pointers or references to data locations within a dataset to improve filter, sort, and search functions. Indexes are usually applied to particular database columns so queries can be executed quickly by identifying relevant rows. Even though indexes enhance query performance, trade-offs are associated with them, including higher storage consumption and possible effects on insertion, update, and deletion speed. We implemented indexing in a few entities. Indexing is typically applied to columns with unique or relatively static entries, such as those with a low cardinality.

Doctor :

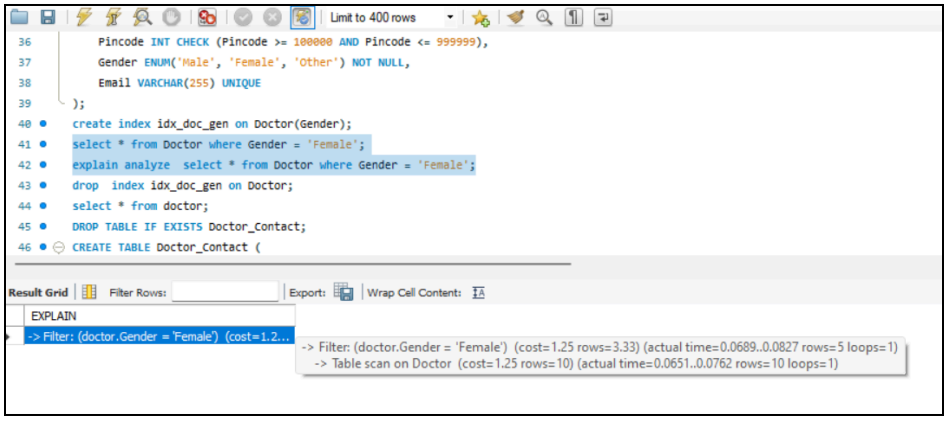
Explanation:

In the context of the doctor table, the gender attribute is a suitable candidate for indexing, especially considering it only has three distinct values. Indexing on the gender column facilitates efficient grouping or retrieval of doctors based on gender. While the impact of indexing may not be noticeable in tables with few entries, it becomes more significant as the number of entries increases.

Without indexing:



With indexing:



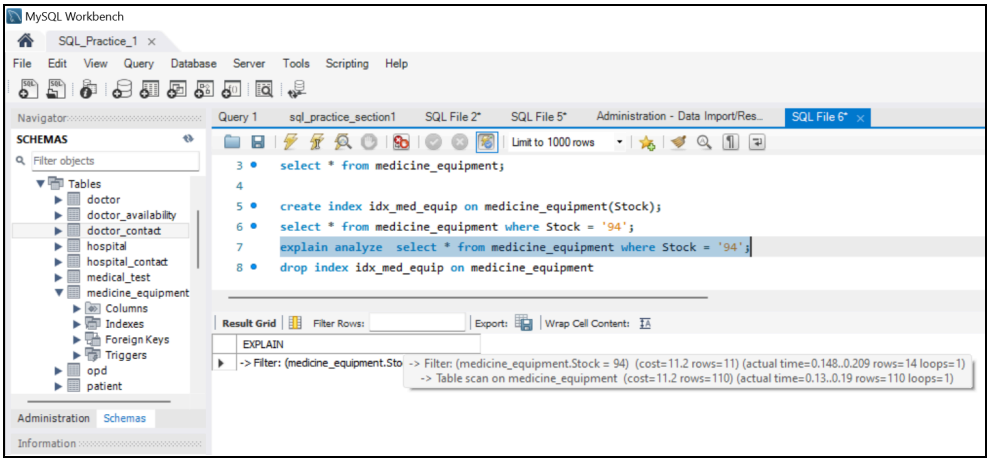
- Cost without Indexing = 1.25
- Cost with Indexing = 1.25

Medicine Equipment :

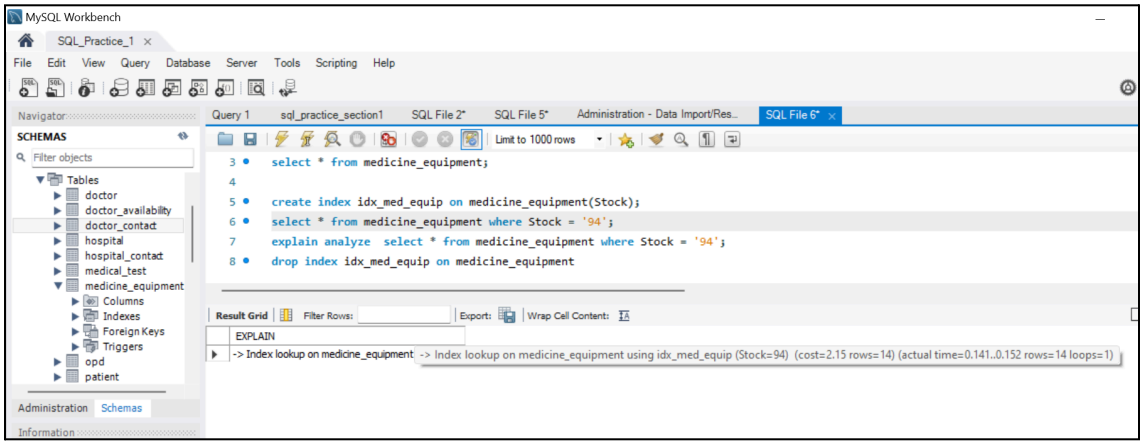
Explanation:

In the context of the medicines_equipments table, the stock attribute emerges as a fitting candidate for indexing, especially given its significance in inventory management. Considering the high volume of entries, the indexing of the stock column, particularly when the stock is set at 94, enables swift and efficient retrieval of relevant records. This optimization proves particularly impactful amidst a large dataset, as it expedites stock management and inventory tracking operations.

Without indexing:



With indexing:



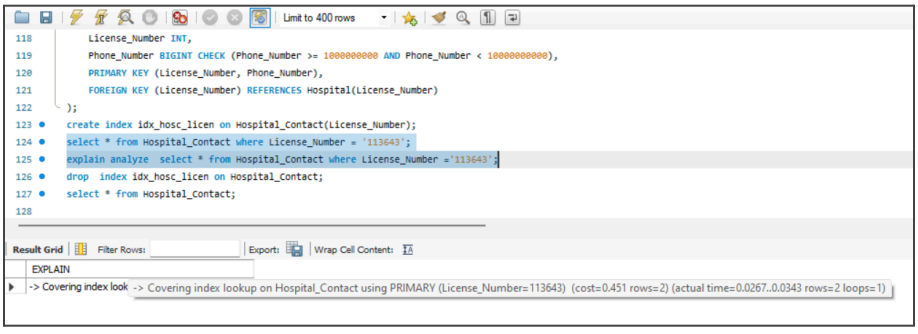
- Cost without Indexing = 11.2
- Cost with Indexing = 2.15

Hospital_Contact:

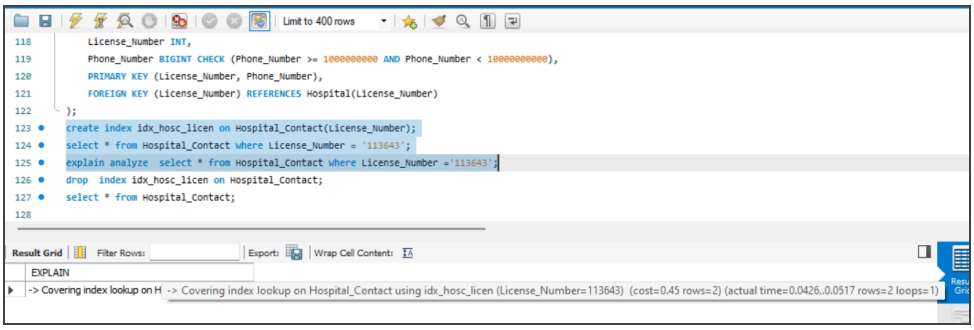
Explanation:

Within the context of the hospital_contact table, the license_number attribute emerges as a prime candidate for indexing. This is particularly relevant given the importance of license numbers in uniquely identifying hospitals. By indexing the license_number column, efficient retrieval and management of hospital records based on their license numbers are facilitated.

Without Indexing:



With Indexing:



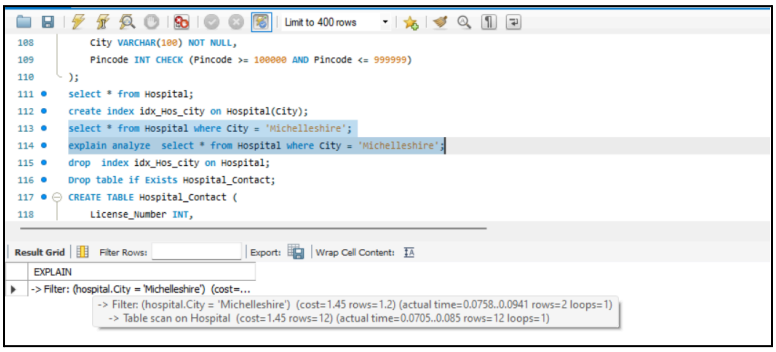
- Cost without Indexing = 0.451
- Cost with Indexing = 0.45

Hospital:

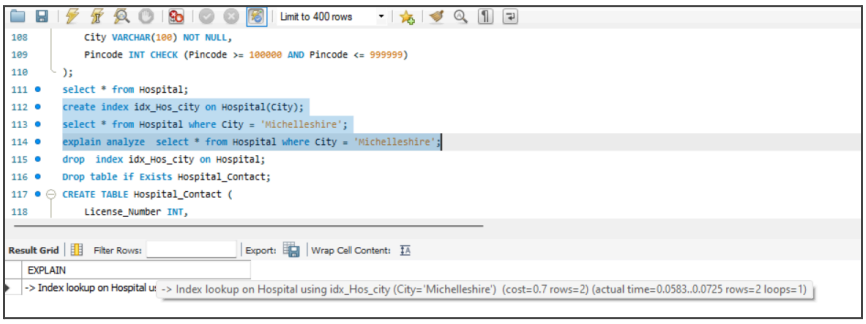
Explanation :

In the case of the hospital table, the city attribute stands out as a suitable candidate for indexing. This is particularly relevant considering the importance of efficiently retrieving hospitals based on their location. By indexing the city column, the database can swiftly identify and manage hospitals situated in specific cities. While the impact of indexing may not be immediately noticeable in tables with fewer entries, the benefits become more pronounced as the dataset grows. However, indexing still provides advantages in terms of ensuring consistent and efficient query performance, especially as the dataset expands over time.

Without indexing:



With indexing:



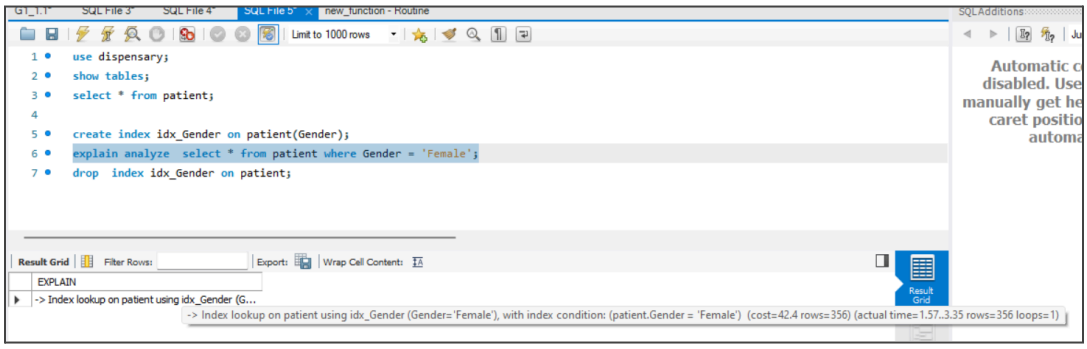
- Cost without Indexing = 1.45
- Cost with Indexing = 0.7

Patient:

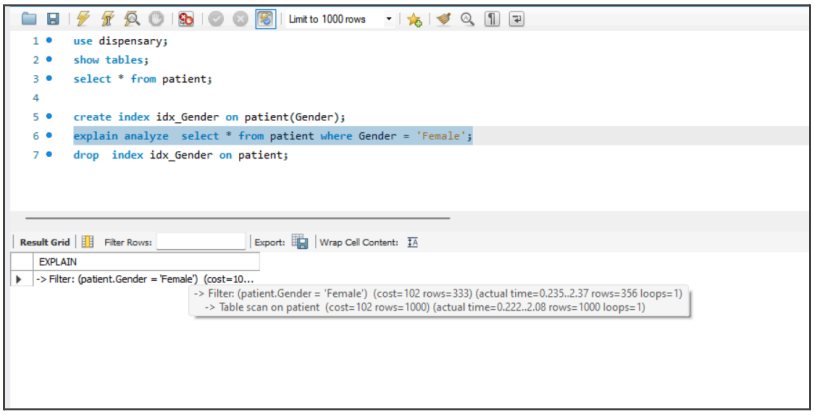
Explanation:

Indexing the gender attribute in a large patient table significantly enhances query efficiency. It allows efficient retrieval of patient records based on gender, reduces query cost, and ensures consistent performance regardless of dataset size. As a result, the explained function execution shows a significant reduction in the cost of data retrieval when compared to a non-indexed scenario.

With Indexing:



Without indexing:



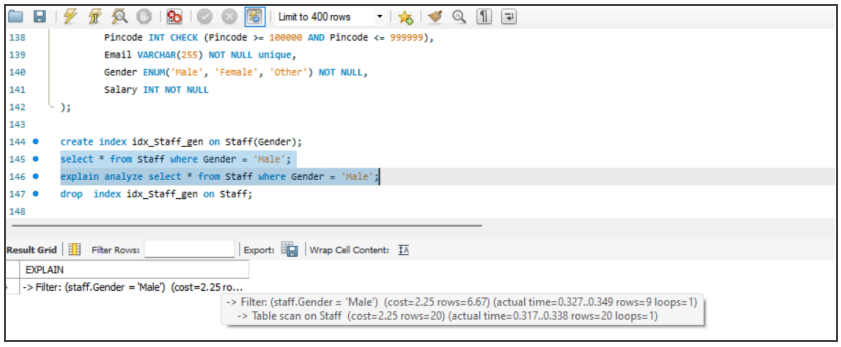
- Cost without Indexing = 102
- Cost with Indexing = 42.4

Staff:

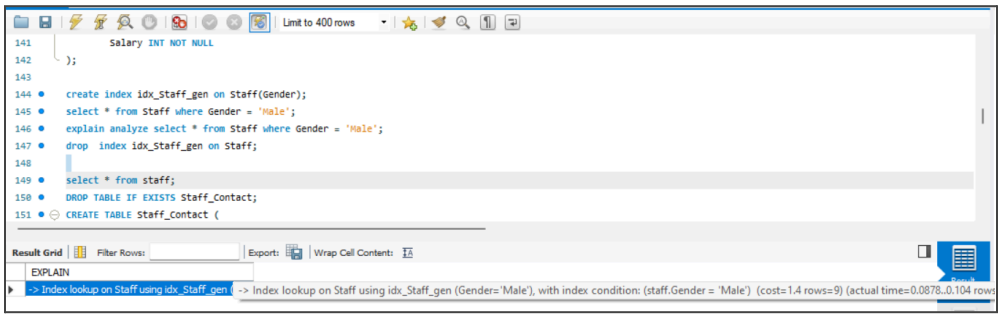
Explanation:

When considering the staff table, indexing on the gender attribute proves advantageous, particularly for queries filtering staff by gender. By leveraging indexing, database query filtering staff by gender, such as gender='male', notably improves query execution time and cost. The EXPLAIN ANALYZE function confirms this enhancement and showcases reduced query costs attributable to the efficient utilization of the index. While the benefits of indexing may be less pronounced in scenarios with limited dataset sizes, they become increasingly evident as the dataset grows, highlighting the enduring importance of indexing in optimizing database performance.

Without Indexing:



With indexing:



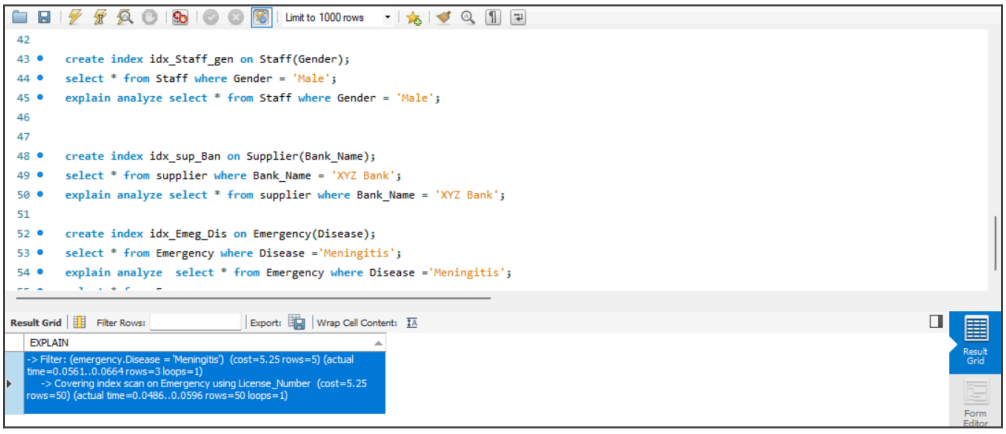
- Cost without Indexing = 2.25
- Cost with Indexing = 1.4

Emergency:

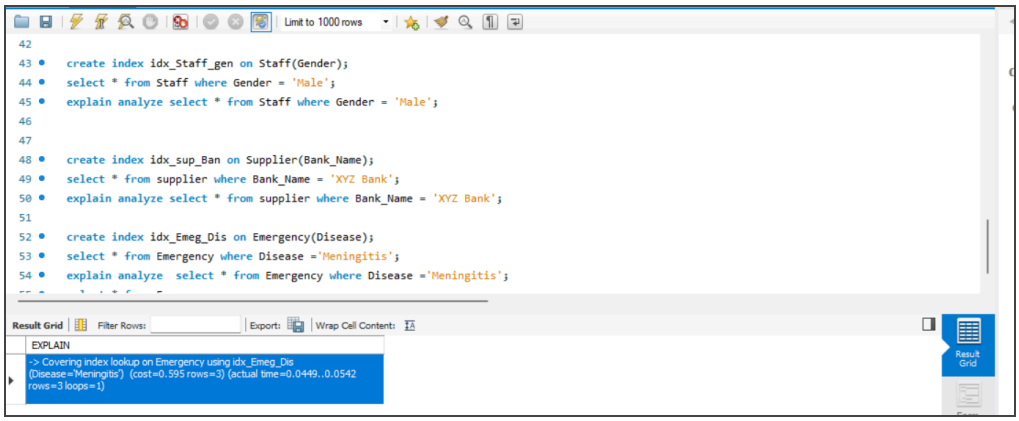
Explanation:

Indexing on the "disease" attribute within the "emergency" entity set is useful as it lowers execution time and costs by improving query performance when filtering emergencies by certain diseases. Validating the efficiency improvements, the EXPLAIN ANALYSE function highlights the ongoing importance of indexing for maximizing database performance, particularly as the dataset develops.

Without indexing:



With indexing:



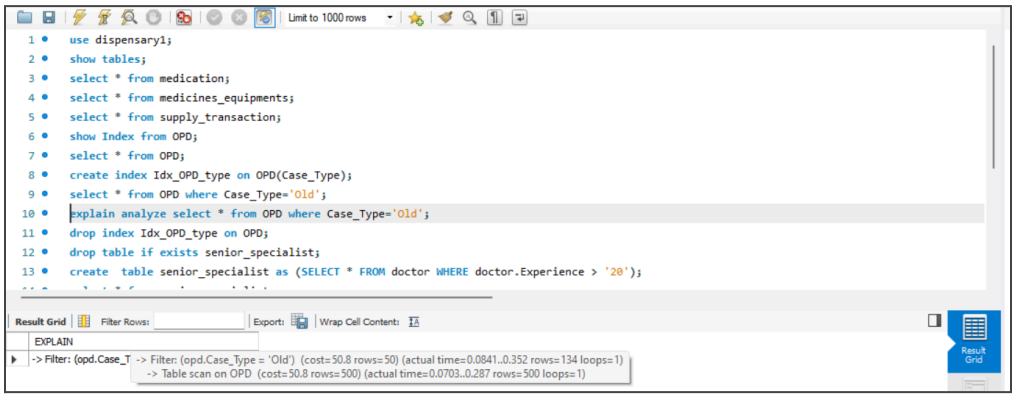
- Cost without Indexing = 5.25
- Cost with Indexing = 0.595

OPD:

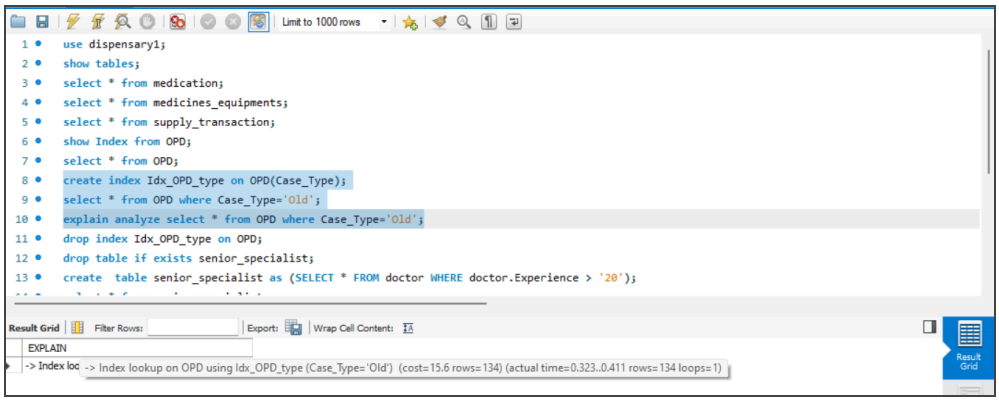
Explanation:

In the context of the OPD table, the case_type attribute emerges as a fitting candidate for indexing, particularly due to its crucial role in distinguishing between old and new cases. Given the considerable volume of entries and the significance of categorizing cases based on their type, indexing the case_type column expedites the retrieval of relevant records, especially when focusing on old or new cases. It streamlines OPD management operations, facilitating expedited access to relevant information and improving overall efficiency in patient care and record-keeping.

Without indexing:



With indexing:



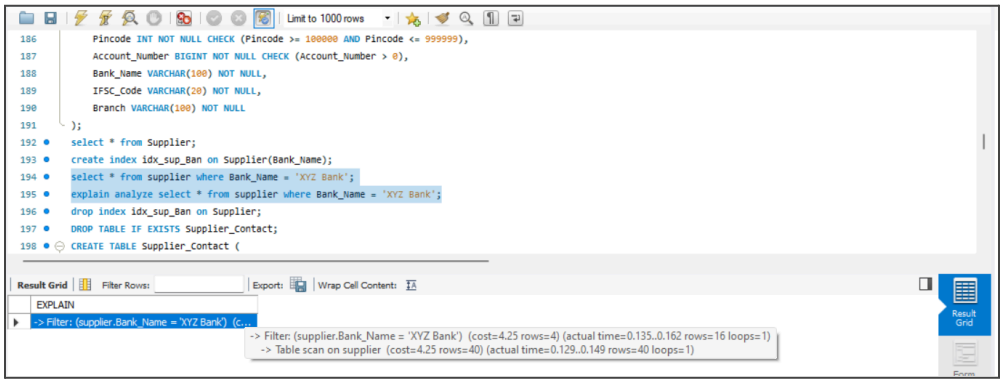
- Cost without Indexing = 50.8
- Cost with Indexing = 15.6

Supplier

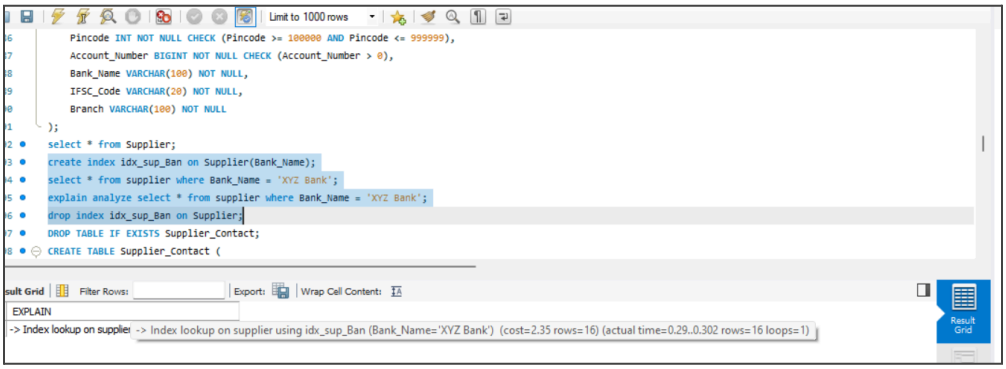
Explanation :

The bank_name attribute in the Supplier table is a prime candidate for indexing because of its crucial function in managing supplier financial transactions. This optimisation greatly speeds up the retrieval of relevant records, particularly when focusing on a particular bank. This improvement is extremely valuable in a large dataset, as it speeds up the processing of financial transactions with suppliers, improves supplier management processes, and raises the overall effectiveness of financial transaction monitoring within the supplier network.

Without indexing:



With indexing:



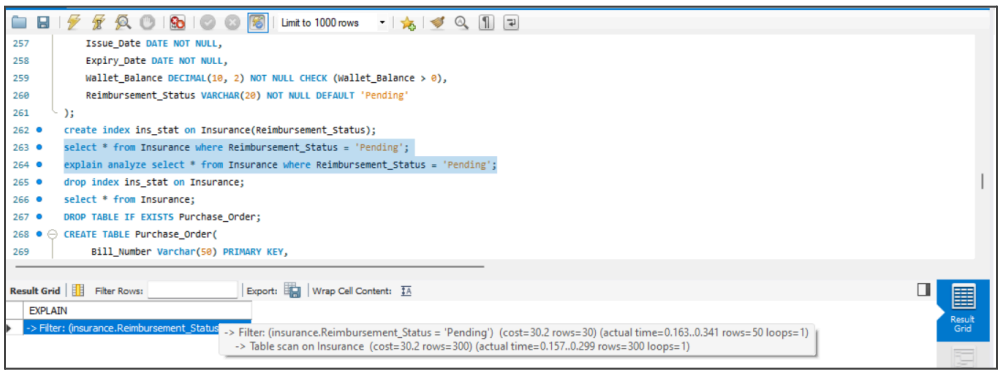
- Cost without Indexing = 4.25
- Cost with Indexing = 2.35

Insurance

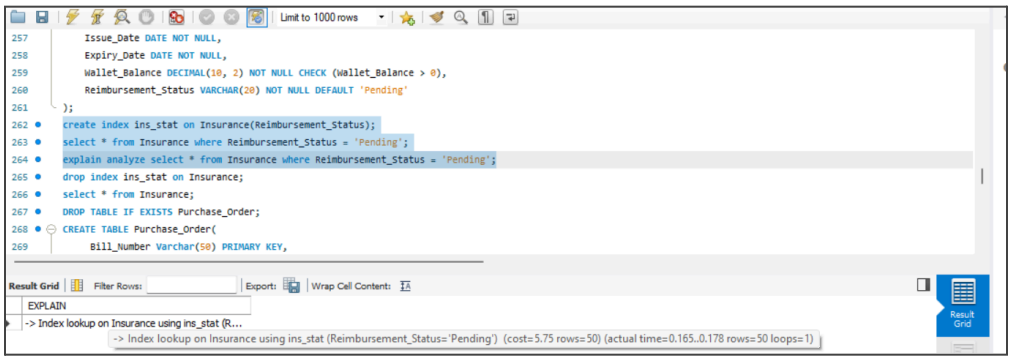
Explanation :

In the context of the insurance table, the reimbursement attribute emerges as a fitting candidate for indexing, particularly due to its pivotal role in tracking pending reimbursements. Given the substantial volume of entries and the critical nature of monitoring pending reimbursements, indexing the reimbursement column, especially when set as "pending," facilitates rapid and efficient retrieval of relevant records. This optimization is particularly impactful in a sizable dataset, as it expedites the handling of pending reimbursements, streamlining insurance management operations and enhancing overall efficiency in tracking financial transactions.

Without indexing:



With indexing:



- Cost without Indexing = 30.2
- Cost with Indexing = 5.75

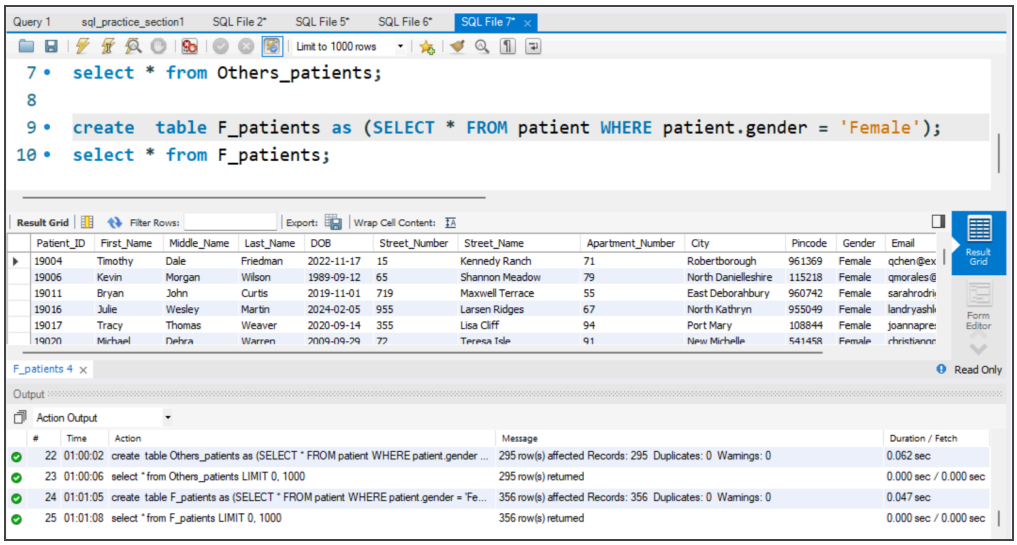
Note: In a few cases, despite implementing indexing in SQL, there are instances where the execution time might remain high, particularly when dealing with a small number of records in the table.

Table Extension

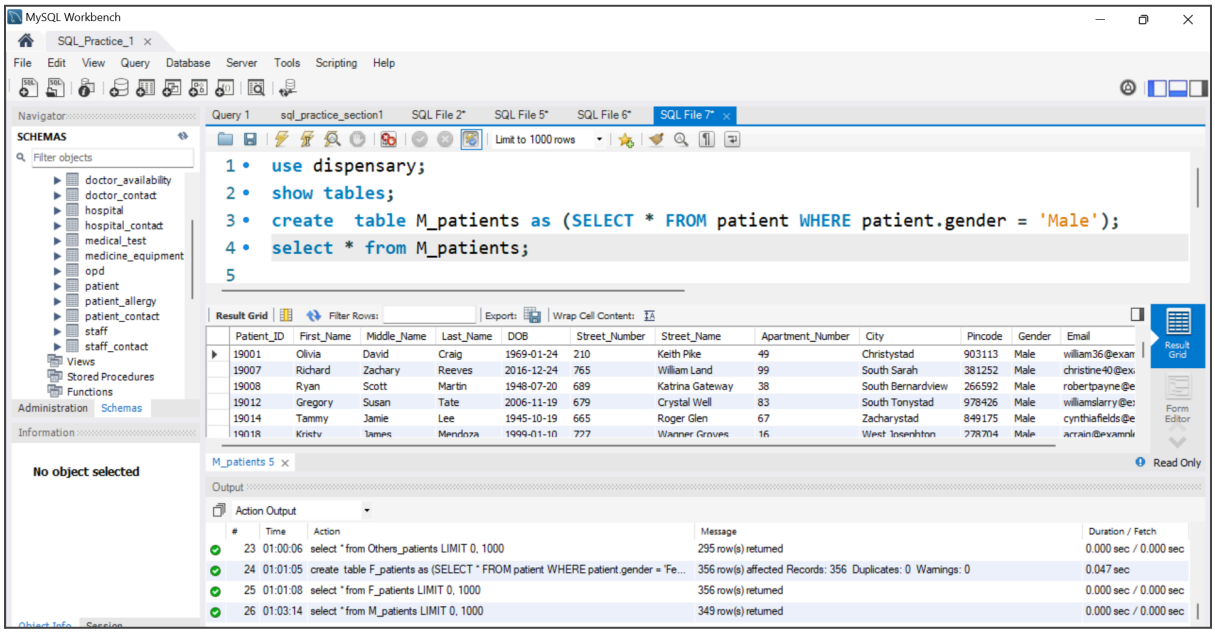
A table extension is a way to add custom fields or properties to an already-existing table without changing the original schema. It enables us to add new columns to the data model while preserving data integrity.

To separate and store the data regarding the patient's gender, whether it be male, female, or other, we implemented a table extension. We made three new tables to store gender information separately:
M_patients: Maintains information on male patients.
F_patients: Maintains information on female patients.
Others_patients: Maintains information on non-binary or alternative gender identities patients.

F_patients:


Query 1
sql_practice_section1
SQL File 2*
SQL File 5*
SQL File 6*
SQL File 7*
Limit to 1000 rows
7 • select * from Others_patients;
8
9 • create table F_patients as (SELECT * FROM patient WHERE patient.gender = 'Female');
10 • select * from F_patients;
Result Grid
Filter Rows:
Export:
Wrap Cell Content:
Patient_ID First_Name Middle_Name Last_Name DOB Street_Number Street_Name Apartment_Number City Pincode Gender Email
19004 Timothy Dale Friedman 2022-11-17 15 Kennedy Ranch 71 Robertborough 961369 Female qchen@ex
19006 Kevin Morgan Wilson 1989-09-12 65 Shannon Meadow 79 North Danielleshire 115218 Female qmorales@
19011 Bryan John Curtis 2019-11-01 719 Maxwell Terrace 55 East Deborahbury 960742 Female sarahrodri
19016 Julie Wesley Martin 2024-02-05 955 Larsen Ridges 67 North Kathryn 955049 Female landryashl
19017 Tracy Thomas Weaver 2020-09-14 355 Lisa Cliff 94 Port Mary 108844 Female joannapre
Result Grid
Form Editor
F_patients 4
Read Only
Output
Action Output
Time Action Message Duration / Fetch
22 01:00:02 create table Others_patients as (SELECT * FROM patient WHERE patient.gender ... 295 row(s) affected Records: 295 Duplicates: 0 Warnings: 0 0.062 sec
23 01:00:06 select * from Others_patients LIMIT 0, 1000 295 row(s) returned 0.000 sec / 0.000 sec
24 01:01:05 create table F_patients as (SELECT * FROM patient WHERE patient.gender = 'Fe... 356 row(s) affected Records: 356 Duplicates: 0 Warnings: 0 0.047 sec
25 01:01:08 select * from F_patients LIMIT 0, 1000 356 row(s) returned 0.000 sec / 0.000 sec

M_patients:


MySQL Workbench
SQL_Practice_1
File Edit View Query Database Server Tools Scripting Help
Navigator
SCHEMAS
Filter objects
doctor_availability
doctor_contact
hospital
hospital_contact
medical_test
medicine_equipment
opd
patient
patient_allergy
patient_contact
staff
staff_contact
Views
Stored Procedures
Functions
Administration Schemas
Information
No object selected
Query 1
sql_practice_section1
SQL File 2*
SQL File 5*
SQL File 6*
SQL File 7*
Limit to 1000 rows
1 • use dispensary;
2 • show tables;
3 • create table M_patients as (SELECT * FROM patient WHERE patient.gender = 'Male');
4 • select * from M_patients;
5
Result Grid
Filter Rows:
Export:
Wrap Cell Content:
Patient_ID First_Name Middle_Name Last_Name DOB Street_Number Street_Name Apartment_Number City Pincode Gender Email
19001 Olivia David Craig 1969-01-24 210 Keith Pike 49 Christystad 903113 Male william36@exan
19007 Richard Zachary Reeves 2016-12-24 765 William Land 99 South Sarah 381252 Male christine40@ex
19008 Ryan Scott Martin 1948-07-20 689 Katrina Gateway 38 South Bernardview 266592 Male robertpayne@e
19012 Gregory Susan Tate 2006-11-19 679 Crystal Well 83 South Tonystad 978426 Male williamslarry@e
19014 Tammy Jamie Lee 1945-10-19 665 Roger Glen 67 Zacharystad 849175 Male cynthiafields@e
Result Grid
Form Editor
M_patients 5
Read Only
Output
Action Output
Time Action Message Duration / Fetch
23 01:00:06 select * from Others_patients LIMIT 0, 1000 295 row(s) returned 0.000 sec / 0.000 sec
24 01:01:05 create table F_patients as (SELECT * FROM patient WHERE patient.gender = 'Fe... 356 row(s) affected Records: 356 Duplicates: 0 Warnings: 0 0.047 sec
25 01:01:08 select * from F_patients LIMIT 0, 1000 356 row(s) returned 0.000 sec / 0.000 sec
26 01:03:14 select * from M_patients LIMIT 0, 1000 349 row(s) returned 0.000 sec / 0.000 sec

Others_patients:

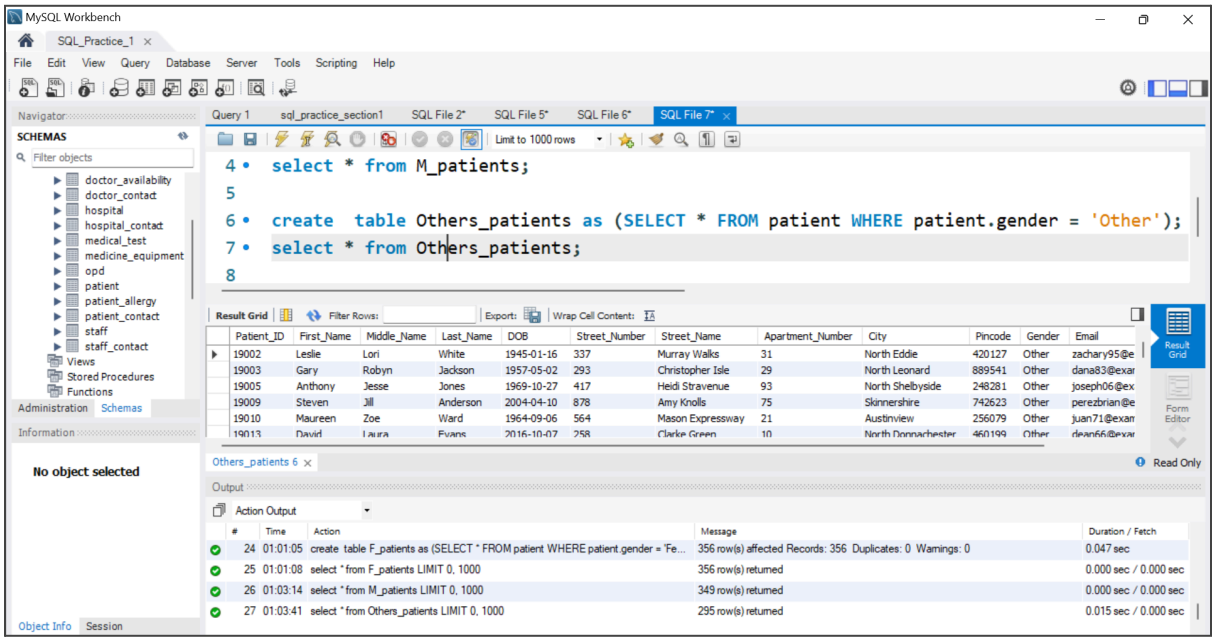


Table Extension in Doctor:

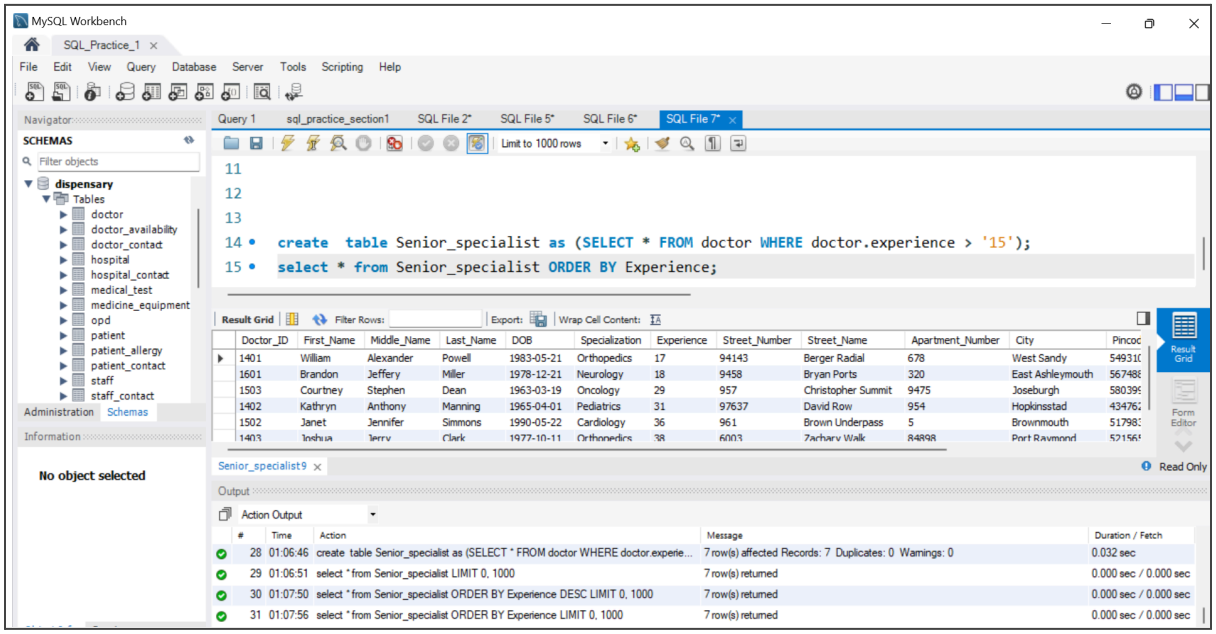
We used a table extension to categorize and separate data about medical professionals, especially senior specialists and junior doctors.

To store information unique to each role, we created three new tables:

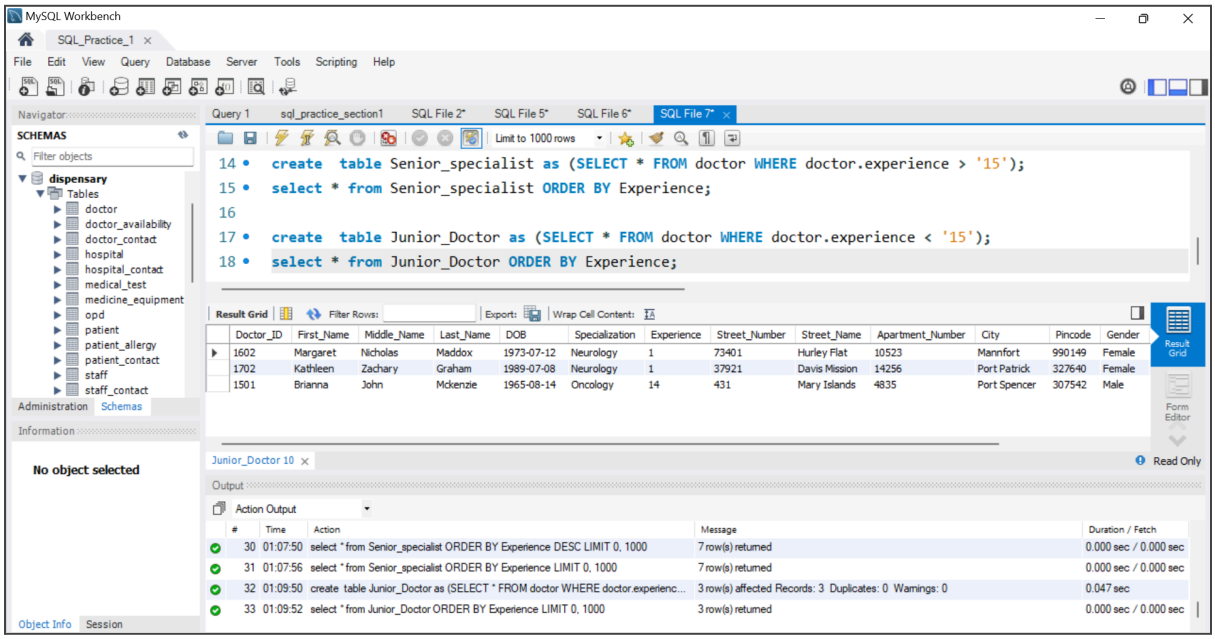
Senior_specialists: Maintains information for senior specialists. Contains information from the primary table of healthcare professionals that includes senior doctor designations.

Junior_doctors: Maintains information for junior doctors. Contains information from the primary table of healthcare professionals that includes junior doctor designations.

Senior_specialist:



Junior_doctors:

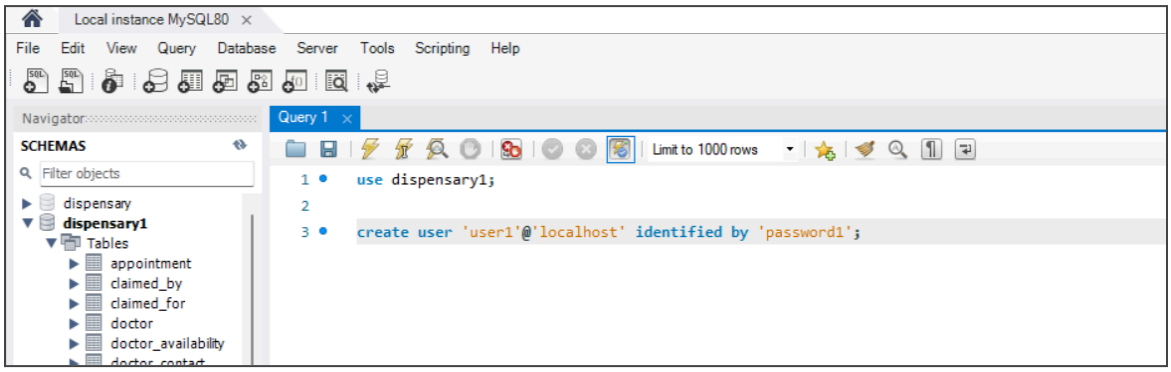


3.2 Responsibility of G2:

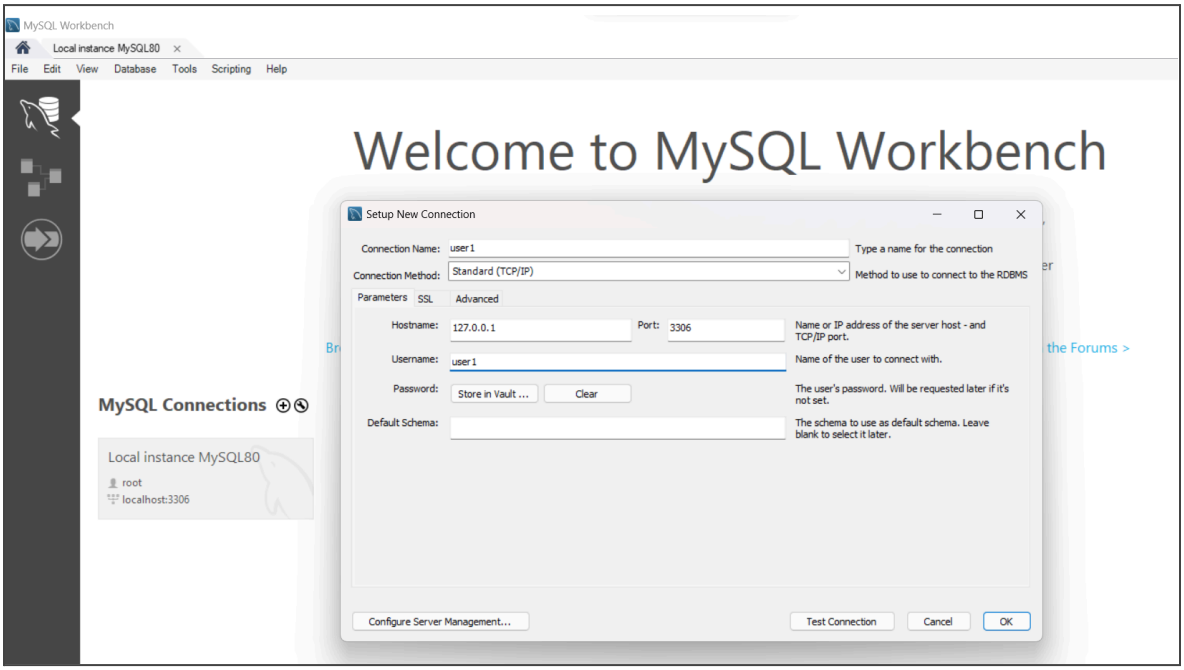
3.2.1] SQL Queries

(i)

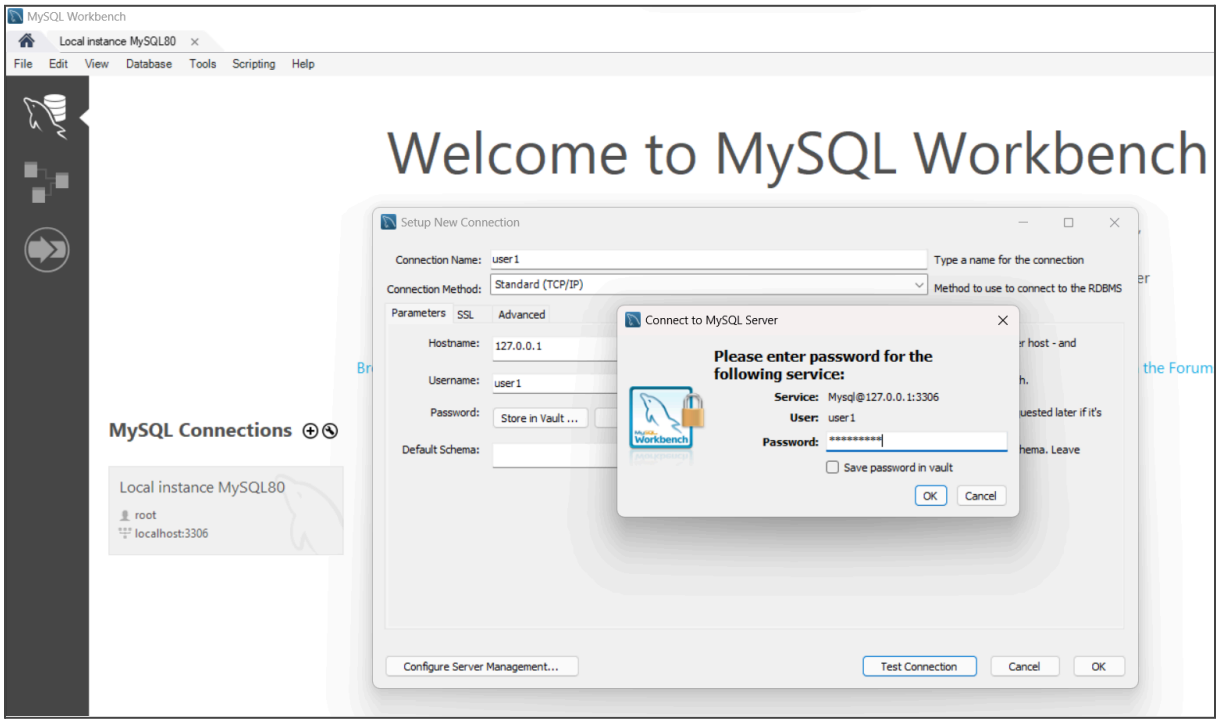
Creating user- user1 with password- password1.



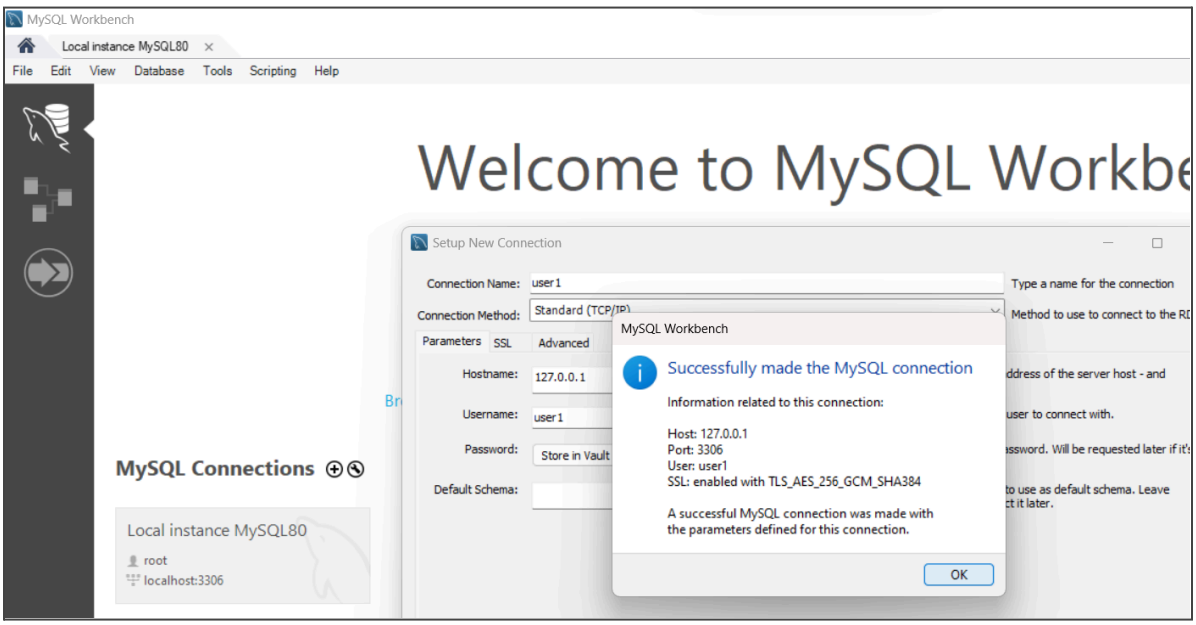
Setting up a new connection with user1.



Entering the password.

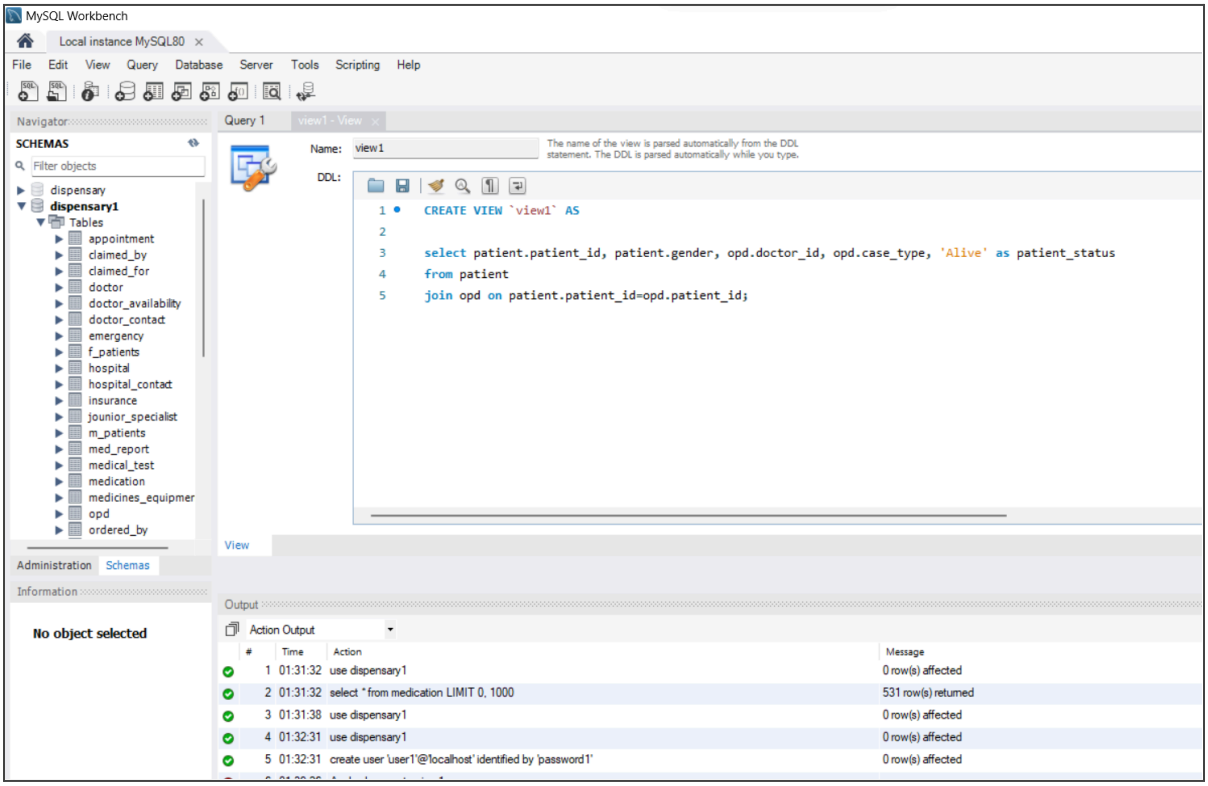


Connection of user1 successfully established.

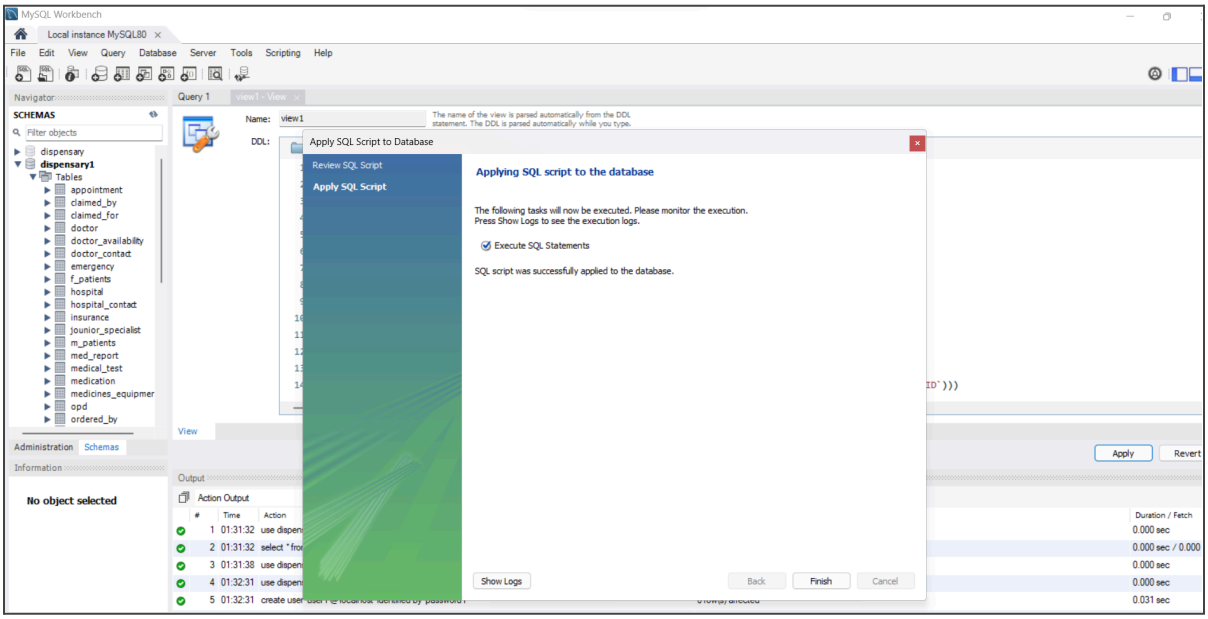


(ii)

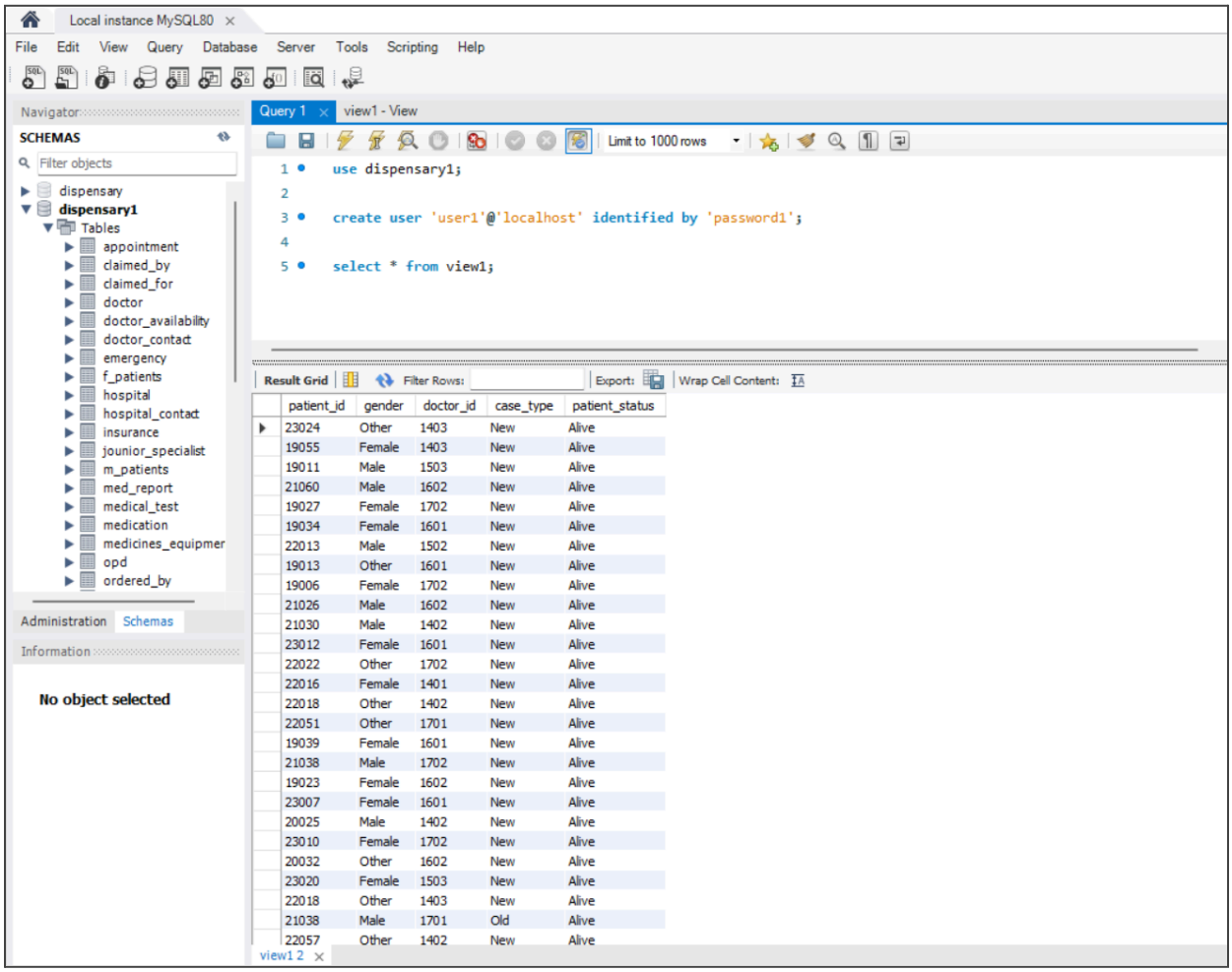
Creating a view- view1 with the Patient Table and OPD Table with patient_id, gender column from the Patient Table and case_type from the OPD Table. Patient_id is the foreign key in the OPD Table. We have also created a user-defined column, patient_status with user-defined data type.



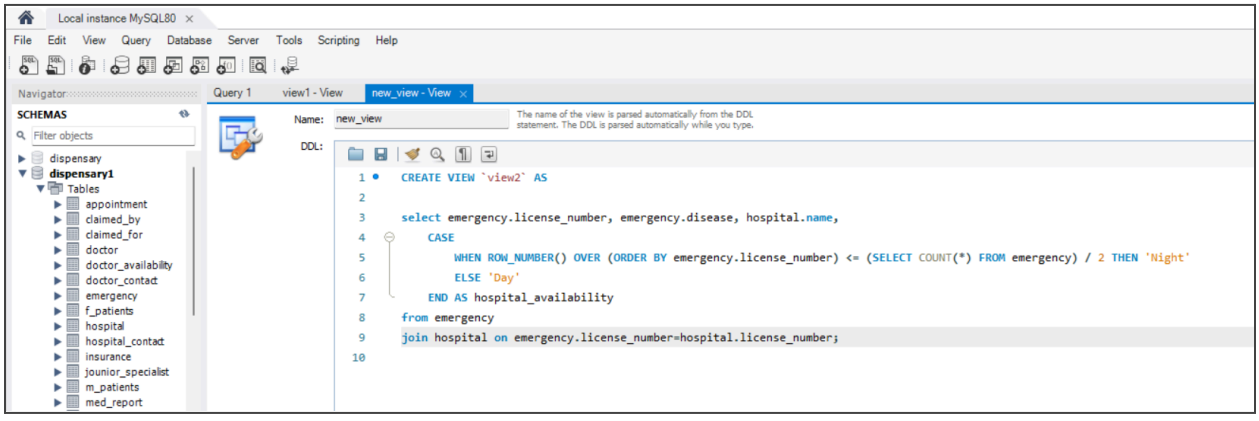
View1 created.



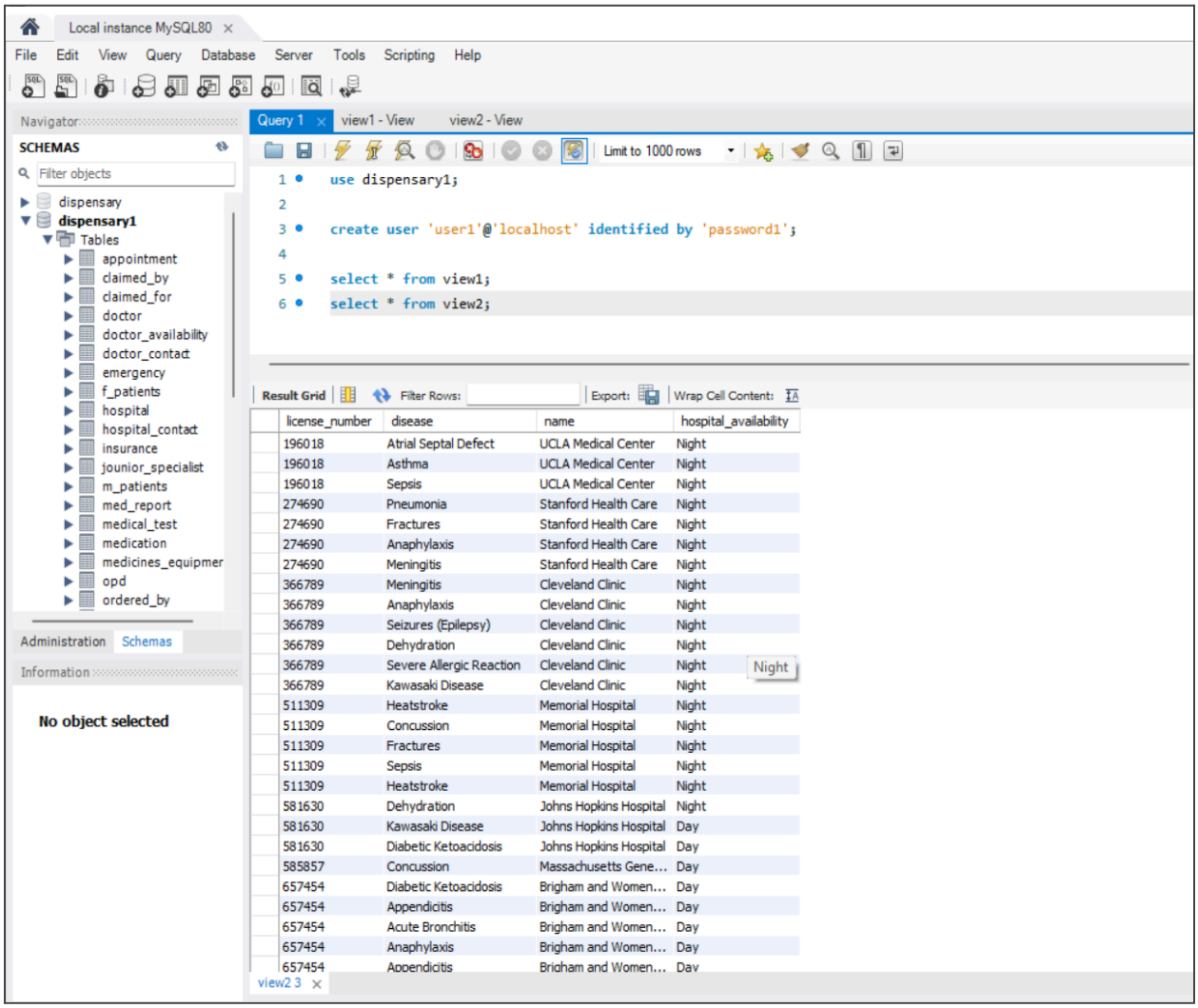
Displaying view1.



Creating a view- view2 with the Emergency Table and Hospital Table with liscence_number, disease column from the Emergency Table and hospital name from the Hospital Table. Liscence_number is the foreign key in the emergency table. We have also created a user-defined column, hospital_availibility, with a user-defined data type having two values, “Night” and “Day”.

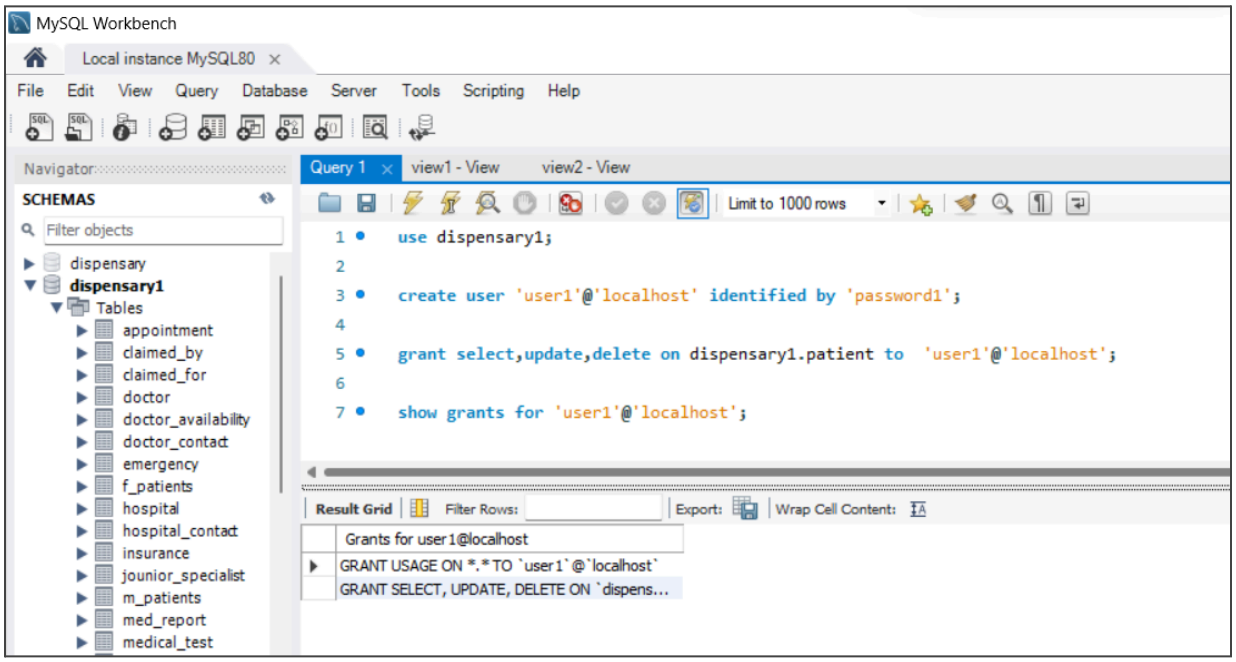


Displaying view2.



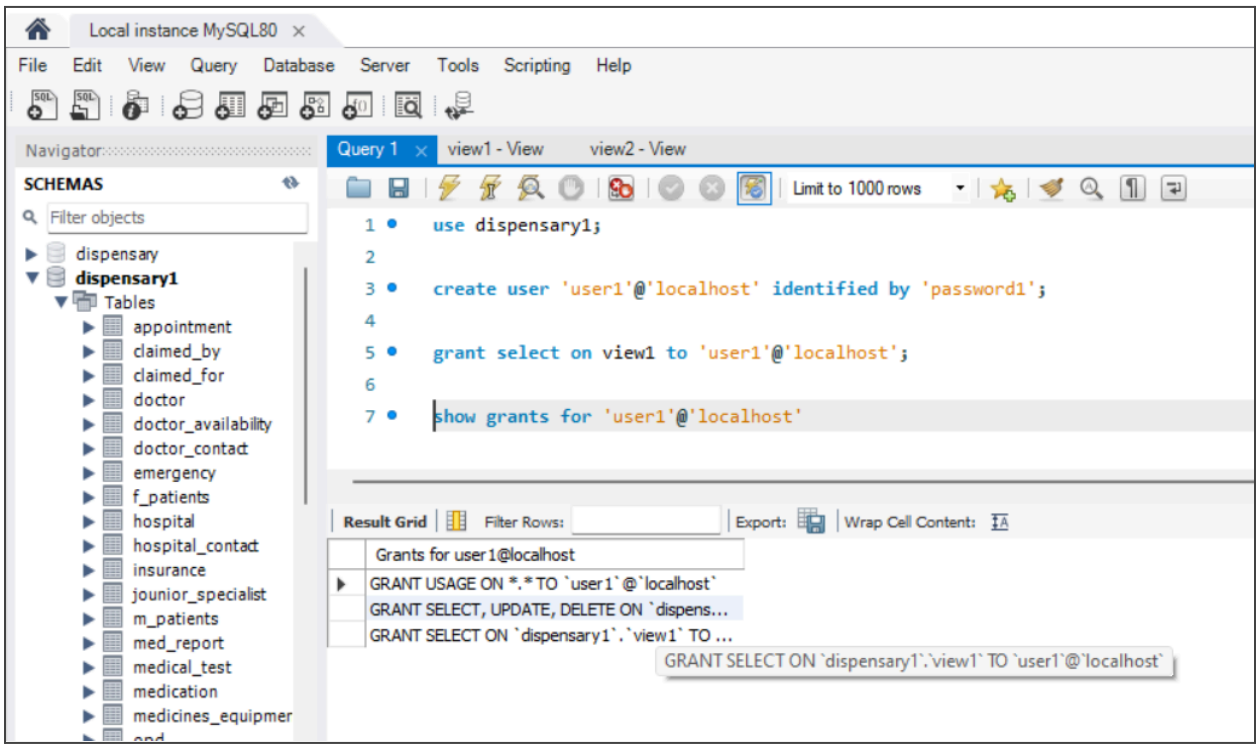
(iii)

Granting SELECT, UPDATE, DELETE permission on Patient Table to user1.



(iv)

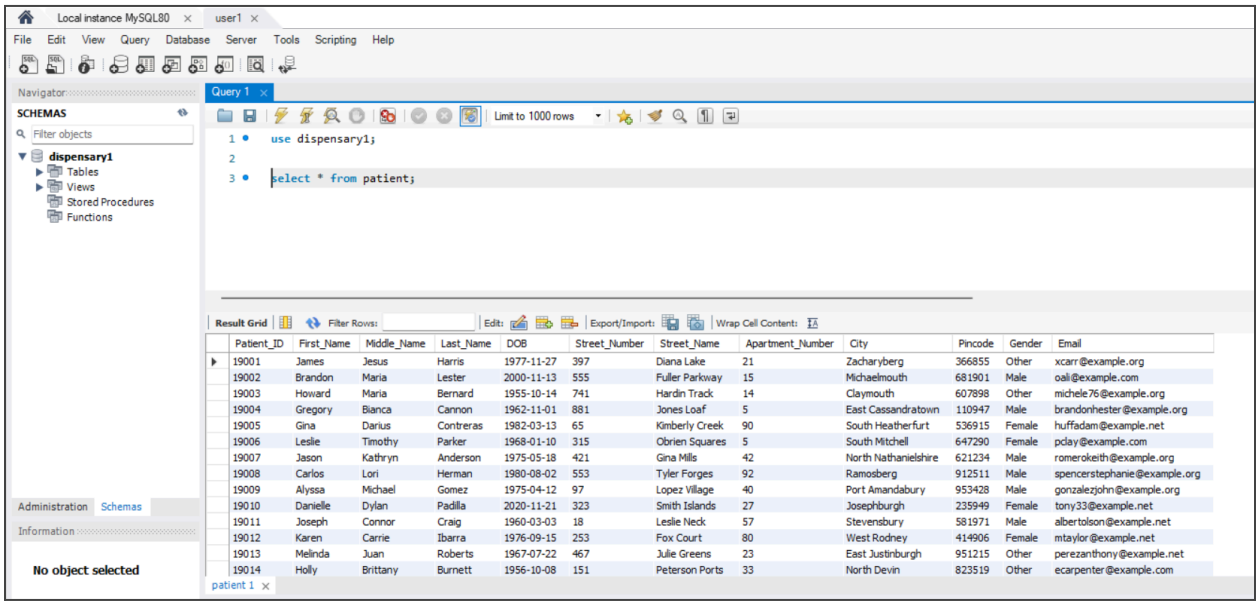
Granting SELECT permission on view1 to user1.



(v) Logging in as user1.

Table1 operations-

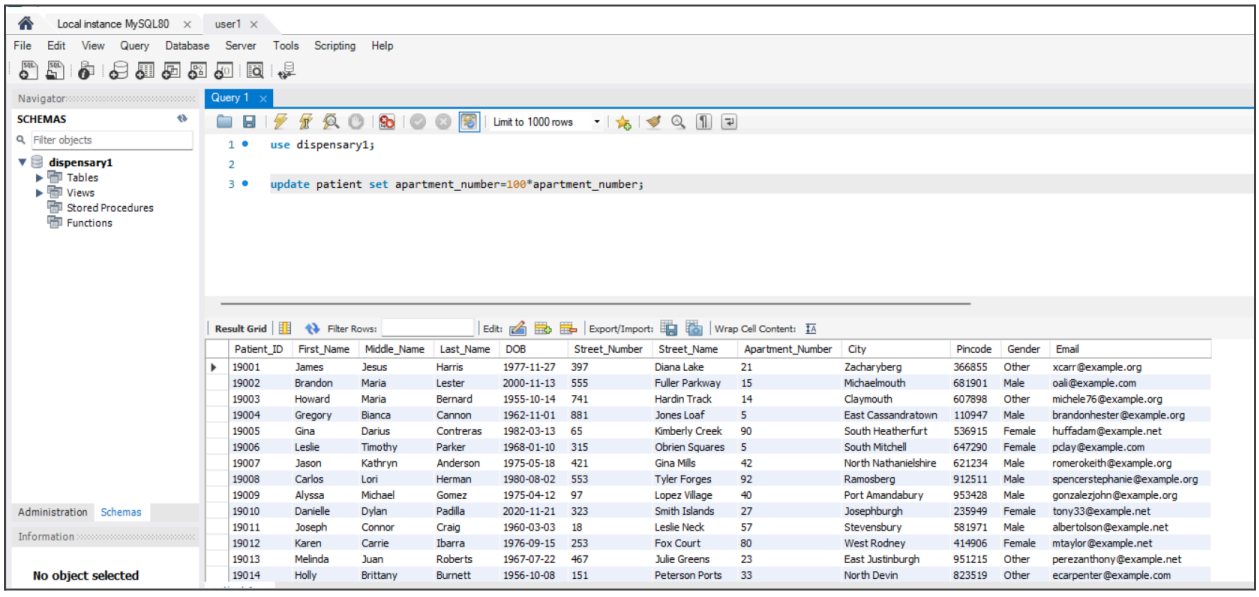
SELECT operation on the Patient Table.



Patient_ID	First_Name	Middle_Name	Last_Name	DOB	Street_Number	Street_Name	Apartment_Number	City	Pincode	Gender	Email
19001	James	Jesus	Harris	1977-11-27	397	Diana Lake	21	Zacharyberg	366855	Other	xcarr@example.org
19002	Brandon	Maria	Lester	2000-11-13	555	Fuller Parkway	15	Michaelmouth	681901	Male	oali@example.com
19003	Howard	Maria	Bernard	1955-10-14	741	Hardin Track	14	Claymouth	607898	Other	michele76@example.org
19004	Gregory	Bianca	Cannon	1962-11-01	881	Jones Loaf	5	East Cassandratown	110947	Male	brandonhester@example.org
19005	Gina	Darius	Contreras	1982-03-13	65	Kimberly Creek	90	South Heatherfurt	536915	Female	huffadam@example.net
19006	Leslie	Timothy	Parker	1968-01-10	315	Obrien Squares	5	South Mitchell	647290	Female	pclay@example.com
19007	Jason	Kathryn	Anderson	1975-05-18	421	Gina Mills	42	North Nathanielshire	621234	Male	romerokeith@example.org
19008	Carlos	Lori	Herman	1980-08-02	553	Tyler Forges	92	Ramosberg	912511	Male	spencerstephanie@example.org
19009	Alyssa	Michael	Gomez	1975-04-12	97	Lopez Village	40	Port Amandabury	953428	Male	gonzalezjohn@example.org
19010	Danielle	Dylan	Padilla	2020-11-21	323	Smith Islands	27	Josephburgh	235949	Female	tony33@example.net
19011	Joseph	Connor	Craig	1960-03-03	18	Leslie Neck	57	Stevensbury	581971	Male	albertolson@example.net
19012	Karen	Carrie	Ibarra	1976-09-15	253	Fox Court	80	West Rodney	414906	Female	mtaylor@example.net
19013	Melinda	Juan	Roberts	1967-07-22	467	Julie Greens	23	East Justinburgh	951215	Other	perezanthony@example.net
19014	Holly	Brittany	Burnett	1956-10-08	151	Peterson Ports	33	North Devin	823519	Other	ecarpenter@example.com

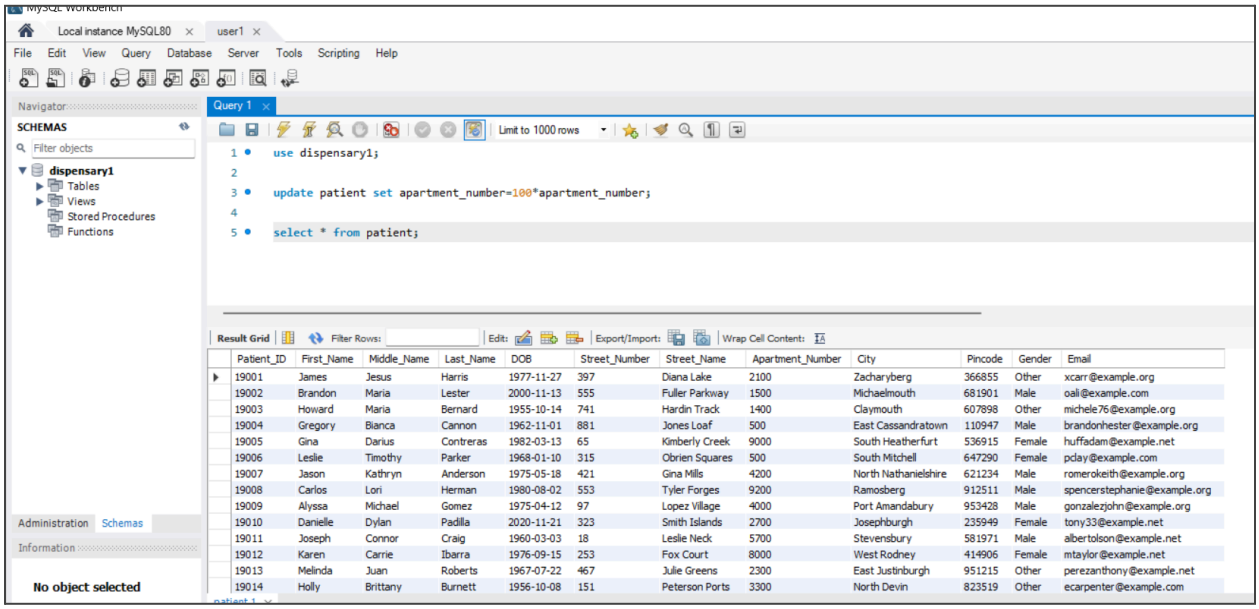
UPDATE operation on the Patient Table.

Before updating the apartment number column.



Patient_ID	First_Name	Middle_Name	Last_Name	DOB	Street_Number	Street_Name	Apartment_Number	City	Pincode	Gender	Email
19001	James	Jesus	Harris	1977-11-27	397	Diana Lake	21	Zacharyberg	366855	Other	xcarr@example.org
19002	Brandon	Maria	Lester	2000-11-13	555	Fuller Parkway	15	Michaelmouth	681901	Male	oali@example.com
19003	Howard	Maria	Bernard	1955-10-14	741	Hardin Track	14	Claymouth	607898	Other	michele76@example.org
19004	Gregory	Bianca	Cannon	1962-11-01	881	Jones Loaf	5	East Cassandratown	110947	Male	brandonhester@example.org
19005	Gina	Darius	Contreras	1982-03-13	65	Kimberly Creek	90	South Heatherfurt	536915	Female	huffadam@example.net
19006	Leslie	Timothy	Parker	1968-01-10	315	Obrien Squares	5	South Mitchell	647290	Female	pclay@example.com
19007	Jason	Kathryn	Anderson	1975-05-18	421	Gina Mills	42	North Nathanielshire	621234	Male	romerokeith@example.org
19008	Carlos	Lori	Herman	1980-08-02	553	Tyler Forges	92	Ramosberg	912511	Male	spencerstephanie@example.org
19009	Alyssa	Michael	Gomez	1975-04-12	97	Lopez Village	40	Port Amandabury	953428	Male	gonzalezjohn@example.org
19010	Danielle	Dylan	Padilla	2020-11-21	323	Smith Islands	27	Josephburgh	235949	Female	tony33@example.net
19011	Joseph	Connor	Craig	1960-03-03	18	Leslie Neck	57	Stevensbury	581971	Male	albertolson@example.net
19012	Karen	Carrie	Ibarra	1976-09-15	253	Fox Court	80	West Rodney	414906	Female	mtaylor@example.net
19013	Melinda	Juan	Roberts	1967-07-22	467	Julie Greens	23	East Justinburgh	951215	Other	perezanthony@example.net
19014	Holly	Brittany	Burnett	1956-10-08	151	Peterson Ports	33	North Devin	823519	Other	ecarpenter@example.com

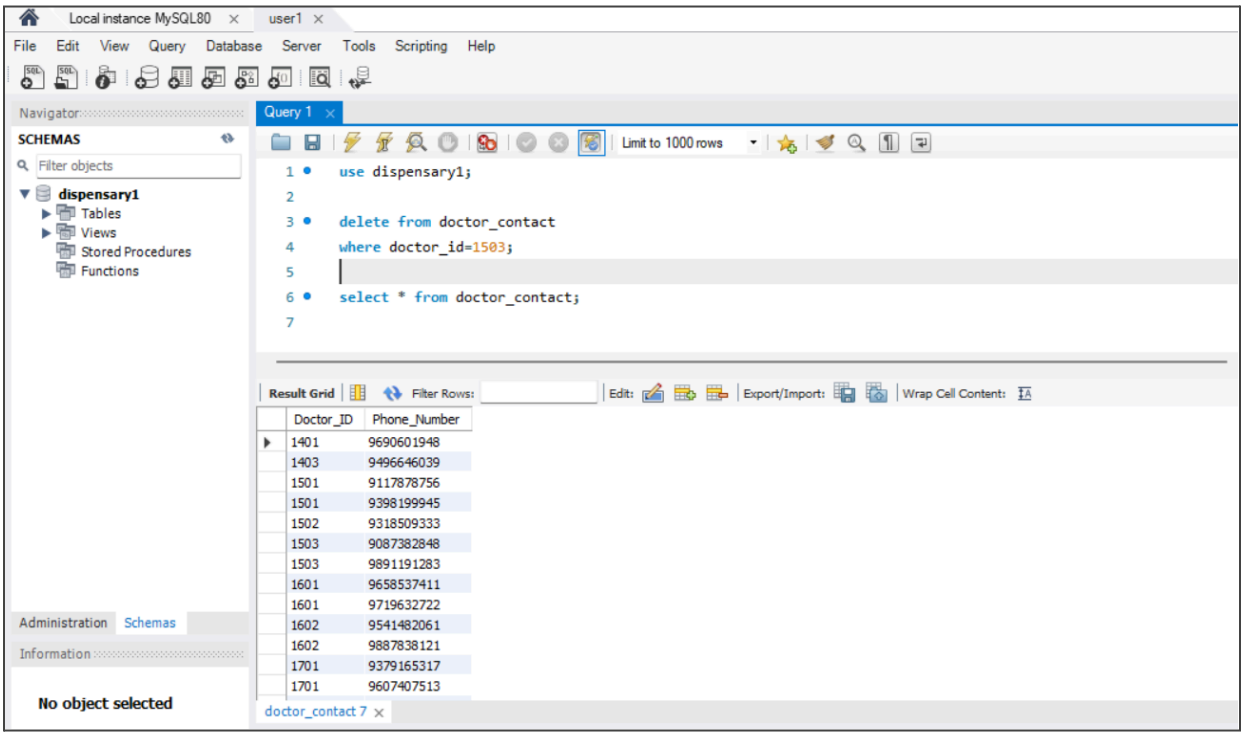
After updating the apartment number column.



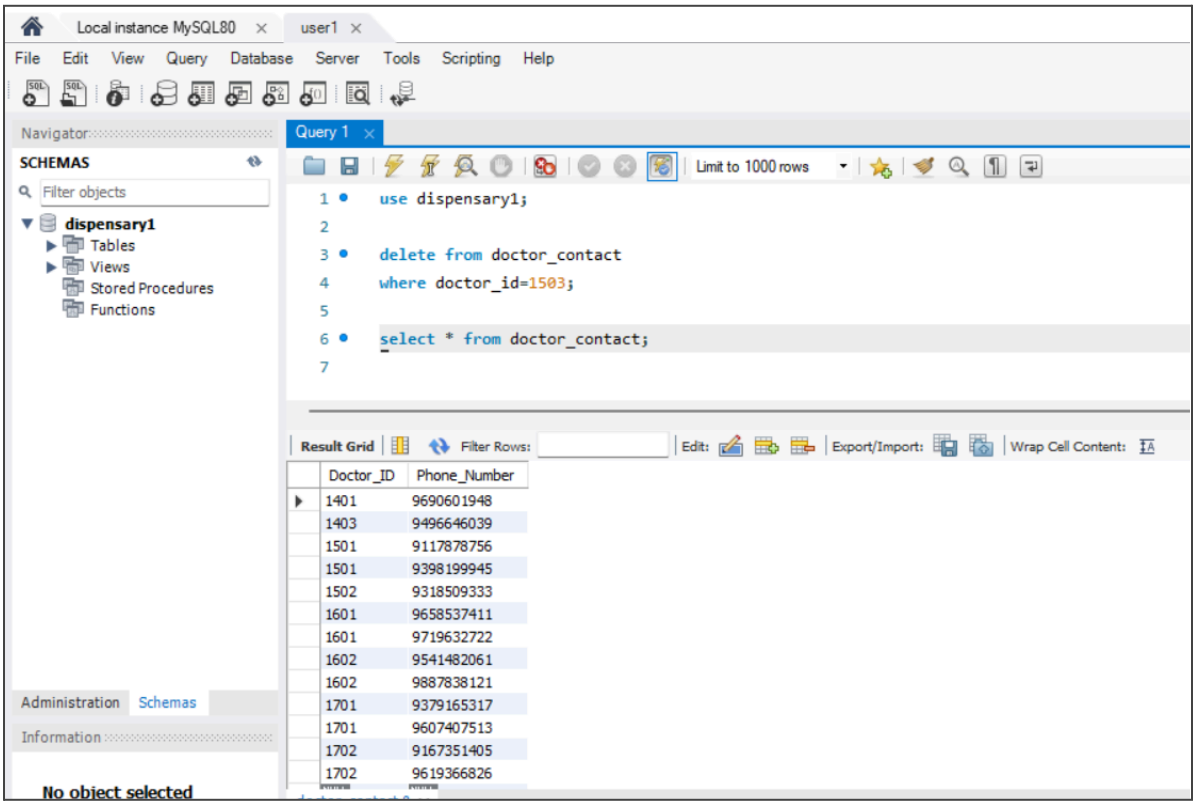
DELETE operation on the Patient Table.

We gave permission for the table of Doctor_contact to user1 for this operation because, in the Patient Table, we could not perform the delete operation as patient_id is the foreign key for some other table, so it would violate referential integrity.

Before deleting records with 1503 doctor_id.

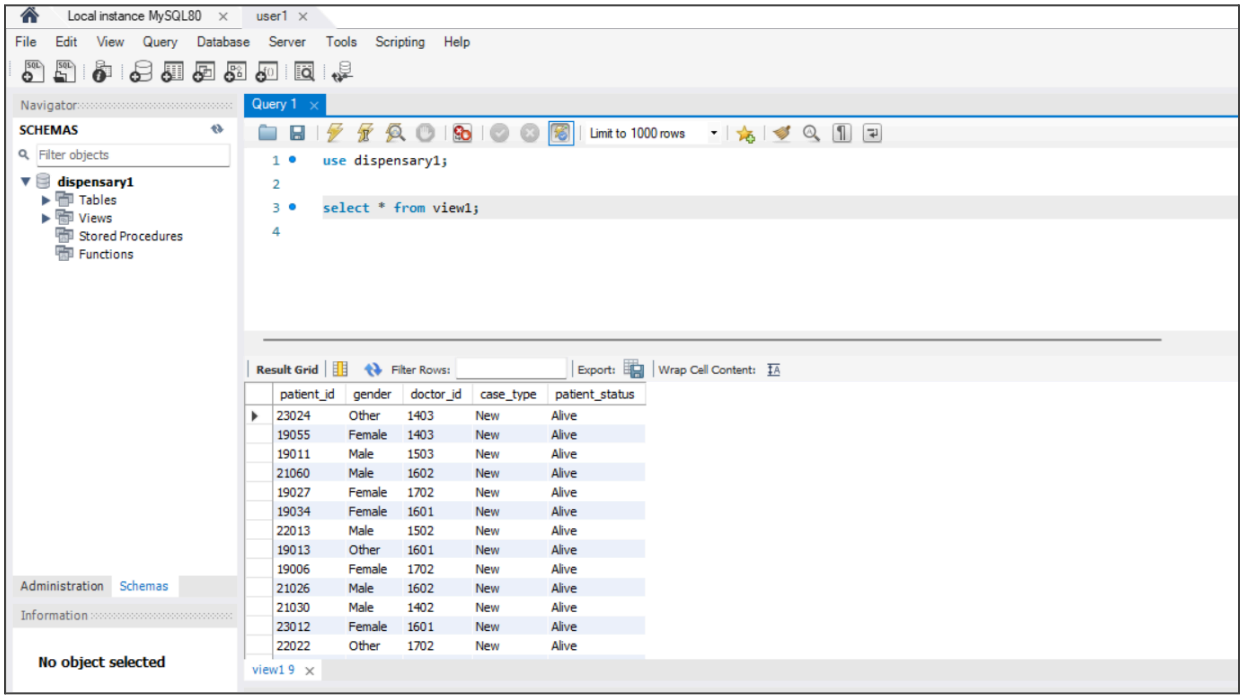


After deleting records with 1503 doctor_id.

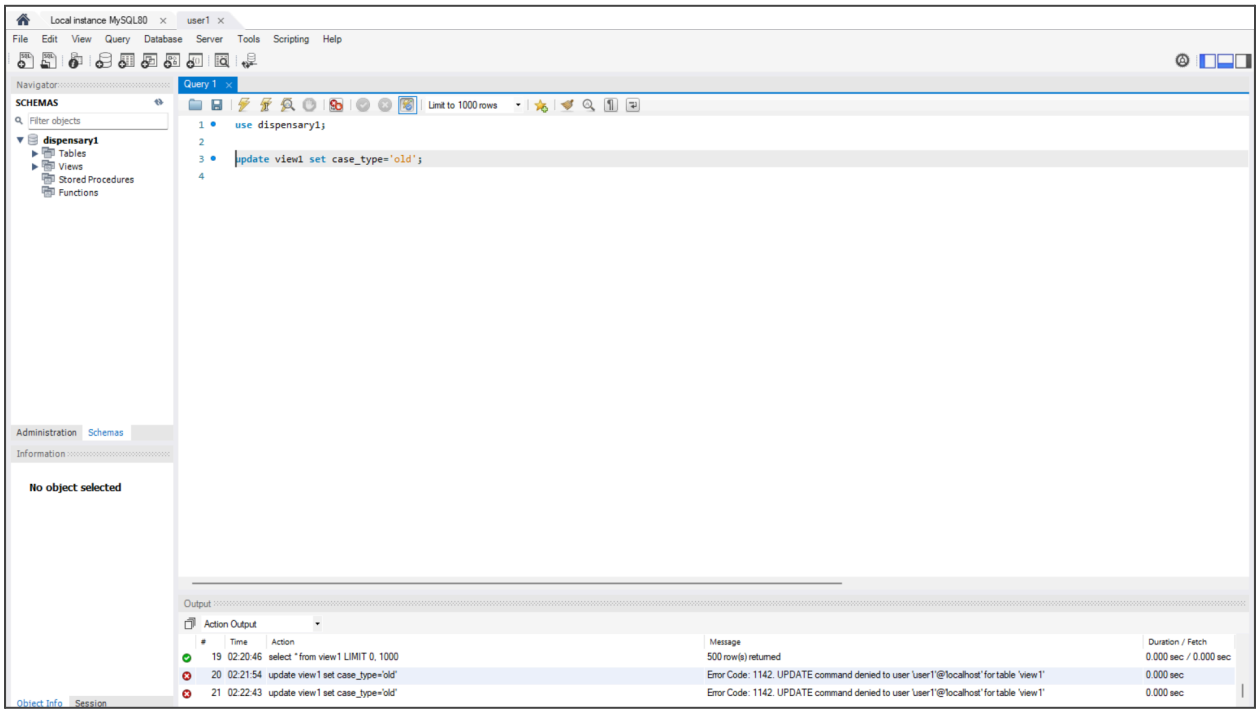


View1 operations-

SELECT operation on the view1.

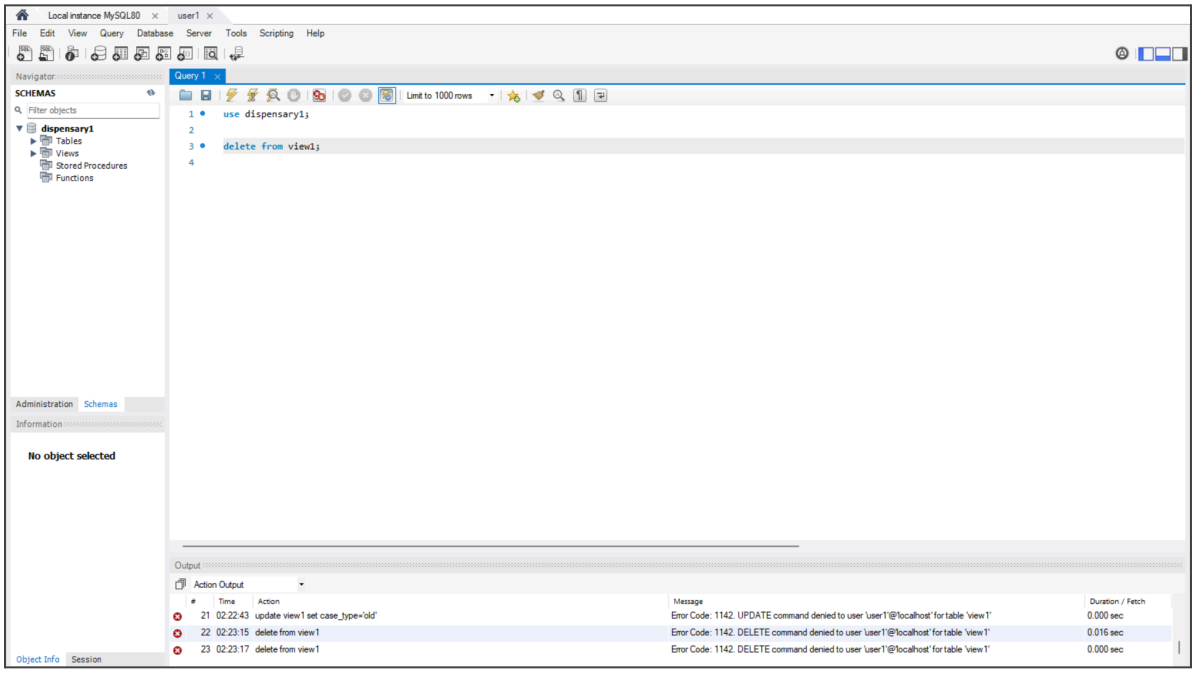


UPDATE operation on view1.



We can see that the update operation in the error log has failed because we have not granted user1 permission to update on view1.

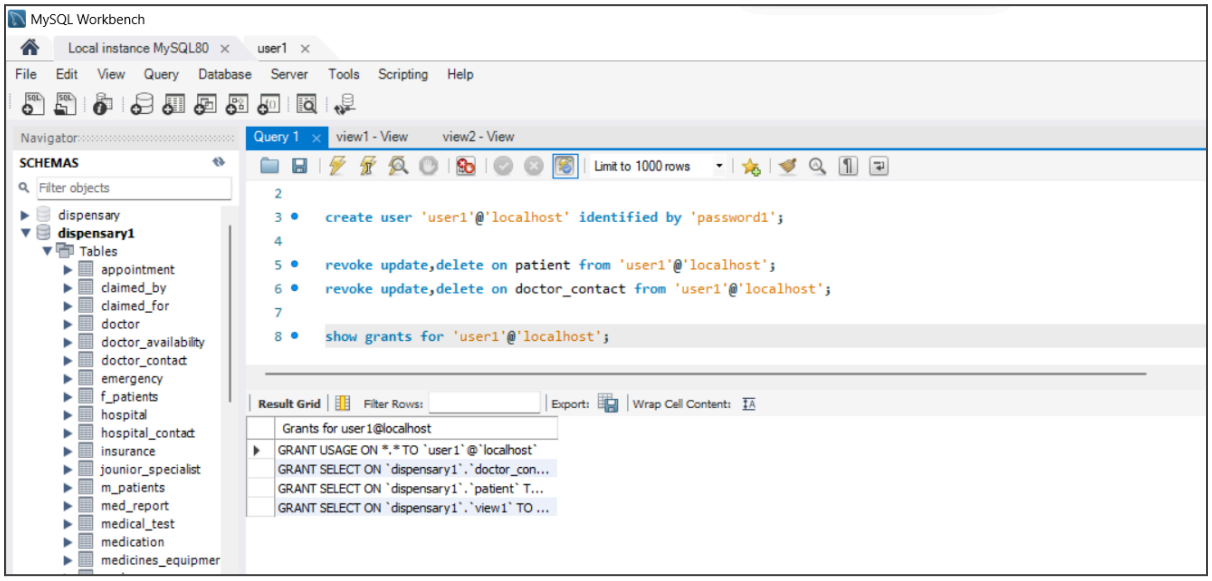
DELETE operation on view1.



We can see that the delete operation in the error log has failed because we have not granted user1 permission to delete on view1.

(vi)

Revoking the permissions of UPDATE and DELETE operation on Patient, Doctor_contact Table for user1.

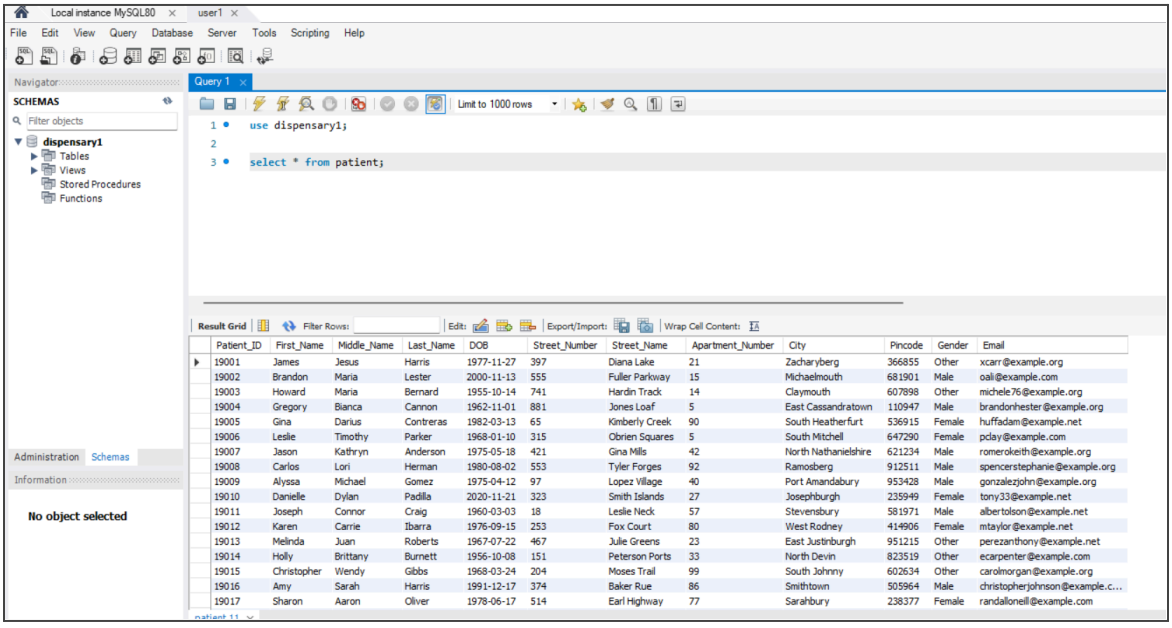


Here, we can see that user1 only has permission to SELECT on Patient Table, Doctor_contact and view1.

(vii)

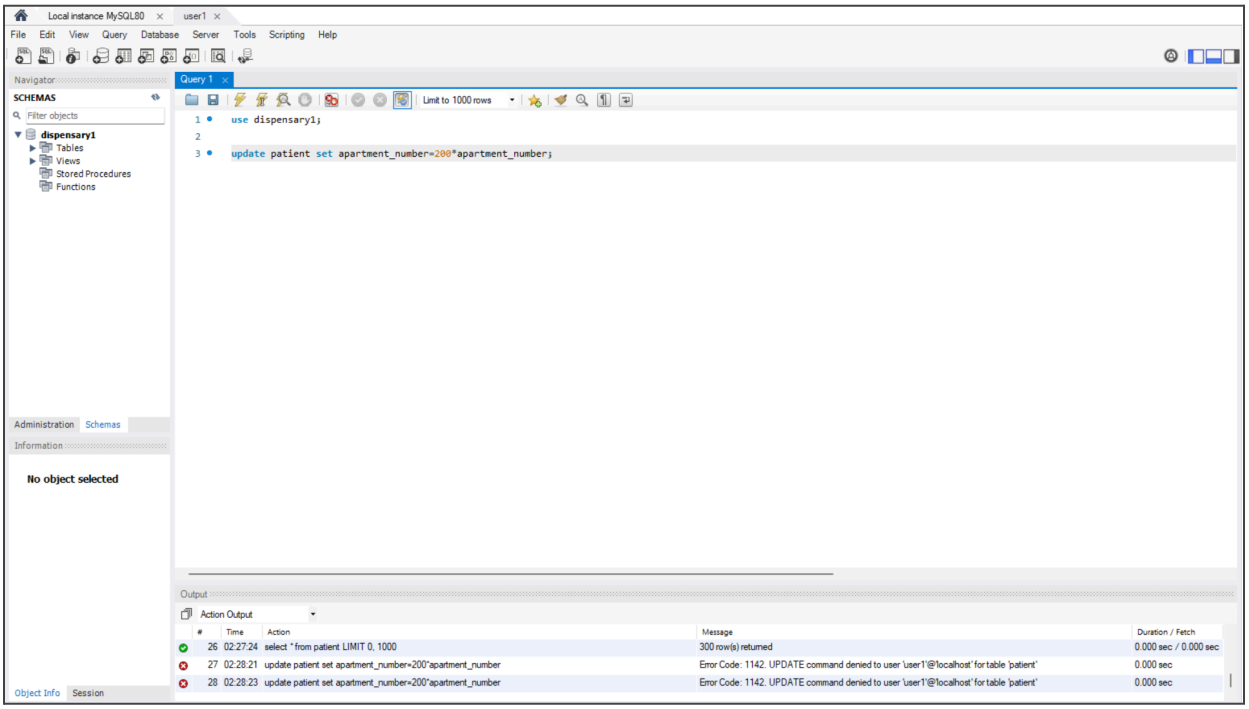
Table1 operations-

SELECT operation on the Patient Table.



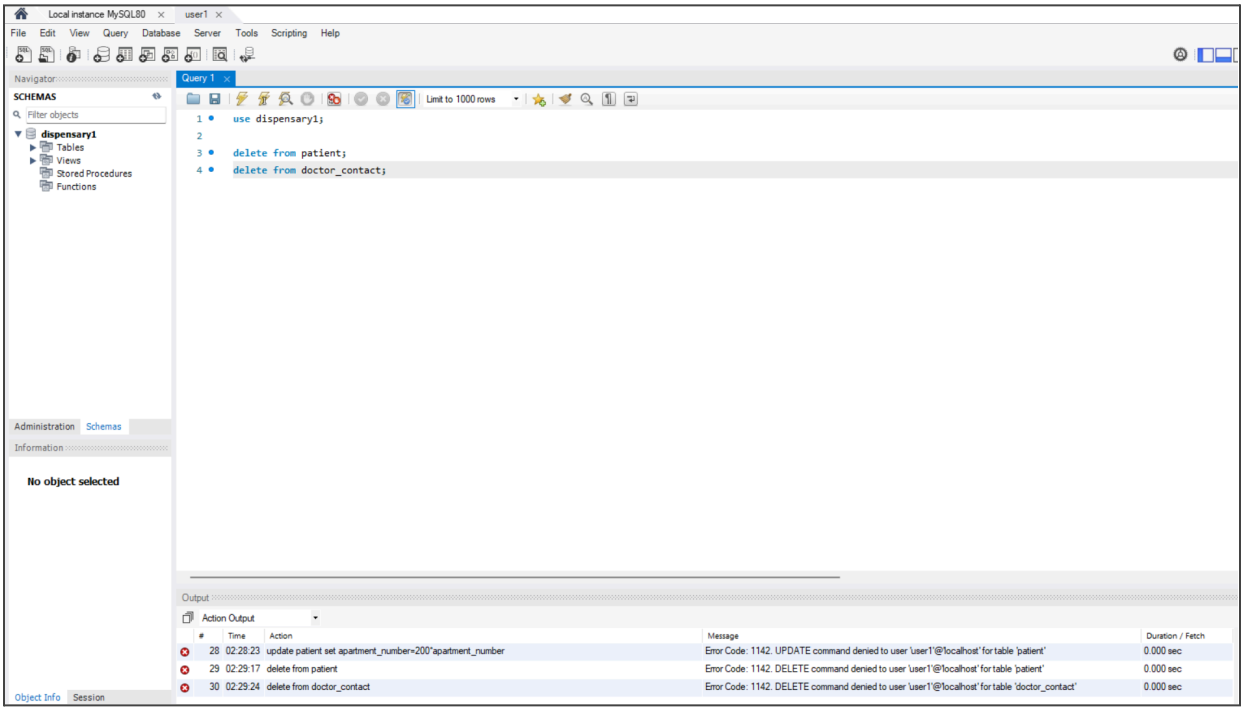
The SELECT operation works.

UPDATE operation on the Patient Table.



User1 cannot update because we have revoked the permissions to update on the Patient Table.

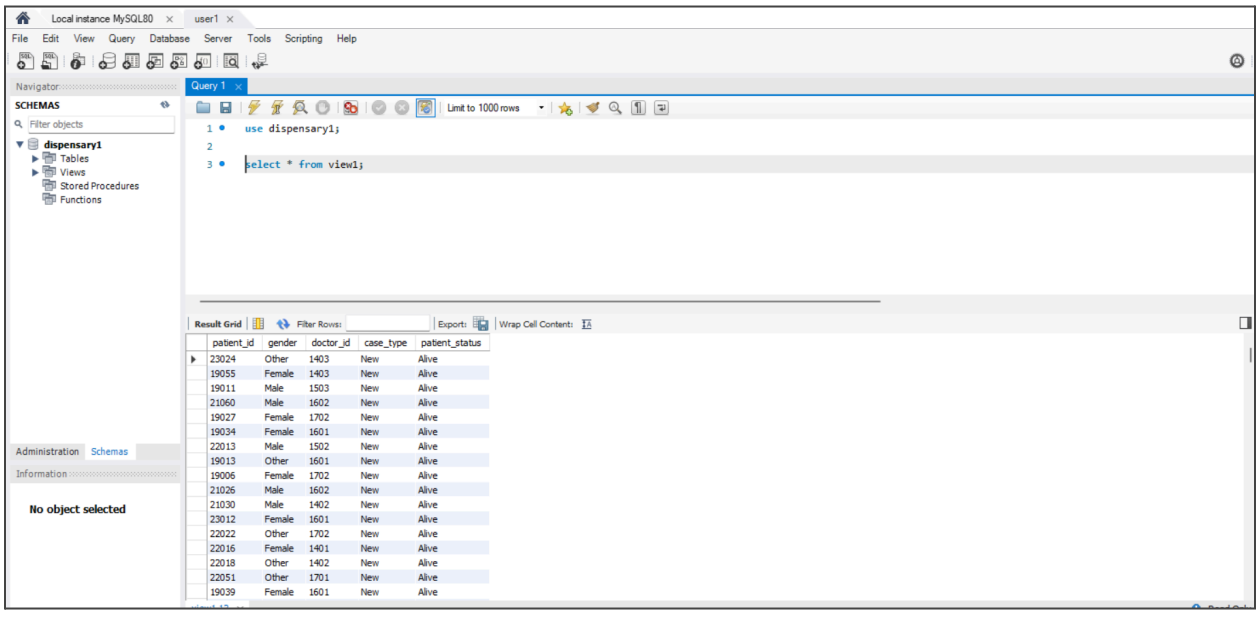
DELETE operation on the Patient Table.
We performed the delete operation on the doctor_contact table because of the referential integrity violation for Patient Table.



We can see that the delete operation in the error log has failed because we had revoked the permissions to delete on the doctor_contact table.

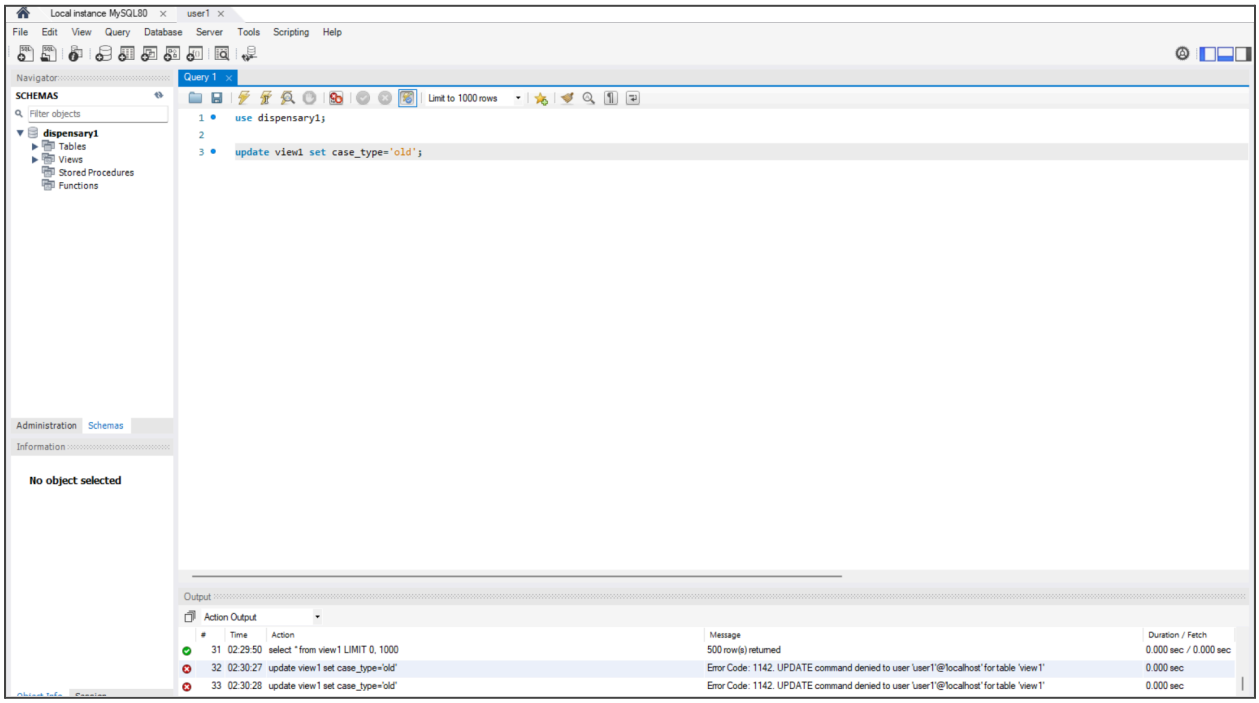
View1 operations-

SELECT operation on view1.



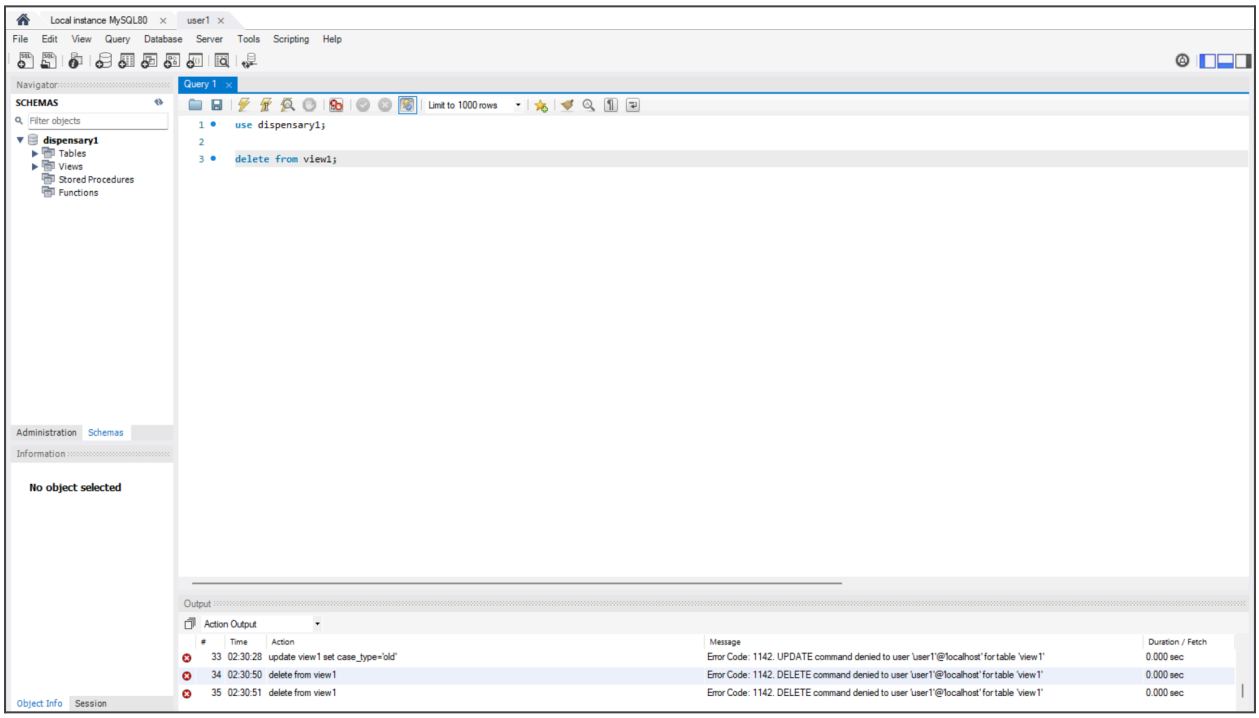
The SELECT operation works.

UPDATE operation on view1.



We can see that the update operation in the error log has failed because we have not granted user1 permission to update on view1.

DELETE operation on view1.



We can see that the delete operation in the error log has failed because we have not granted user1 permission to delete on view1.

3.2.2] Referential Integrity

What is referential integrity, and why is it important?

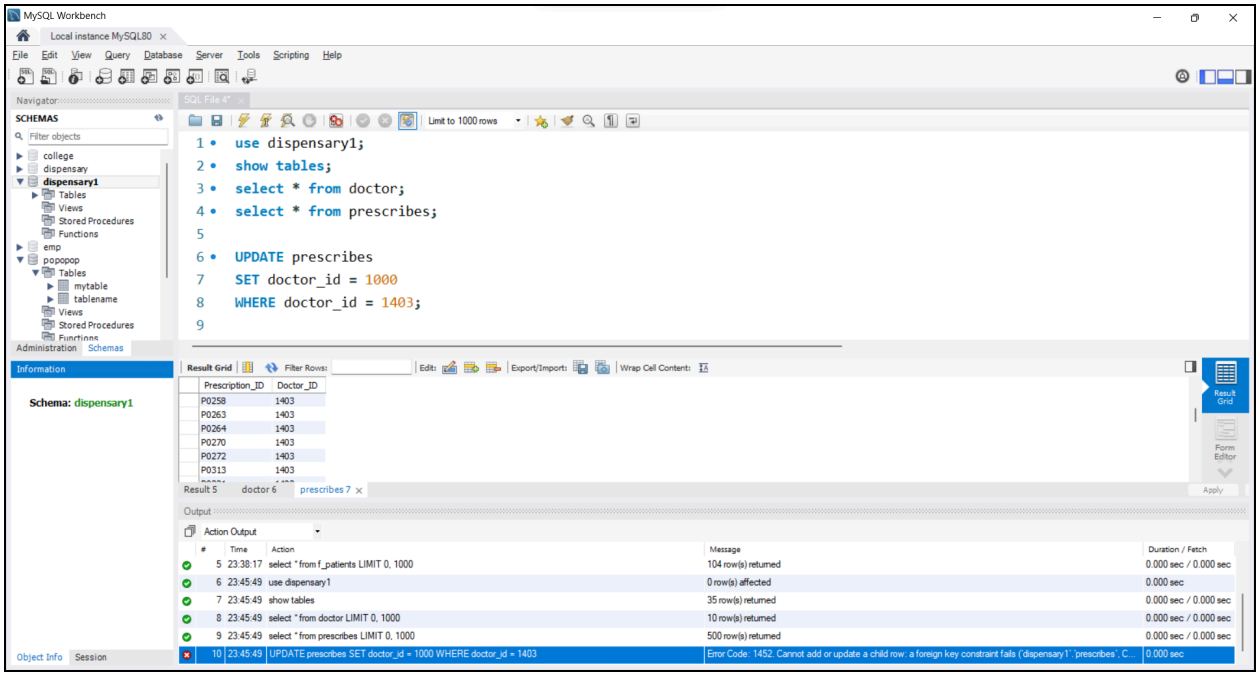
Referential integrity is a fundamental concept in relational database systems that ensures the consistency and accuracy of relationships between tables. It dictates that foreign key values in a child table must always reference valid primary key values in the parent table. In essence, it guarantees that relationships between tables remain intact, preventing orphaned or invalid references. Referential integrity constraints enforce rules that govern the insertion, deletion, and modification of data, maintaining the integrity of the database structure. By upholding referential integrity, databases can ensure data consistency, accuracy, and reliability, facilitating efficient data management and reliable query operations.

Situations that may violate referential integrity:

1] Altering Foreign Key Value:

If an existing record undergoes alteration with a new foreign key value that lacks correspondence to any primary key value in the referenced table, it undermines the integrity of the relationship between the tables. This prompts an error to uphold the database structure's integrity.

Example in our case:



Error message that appeared:

Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails (`dispensary1`.`prescribes`, CONSTRAINT `prescribes_ibfk_2` FOREIGN KEY (`Doctor_ID`) REFERENCES `doctor` (`Doctor_ID`))

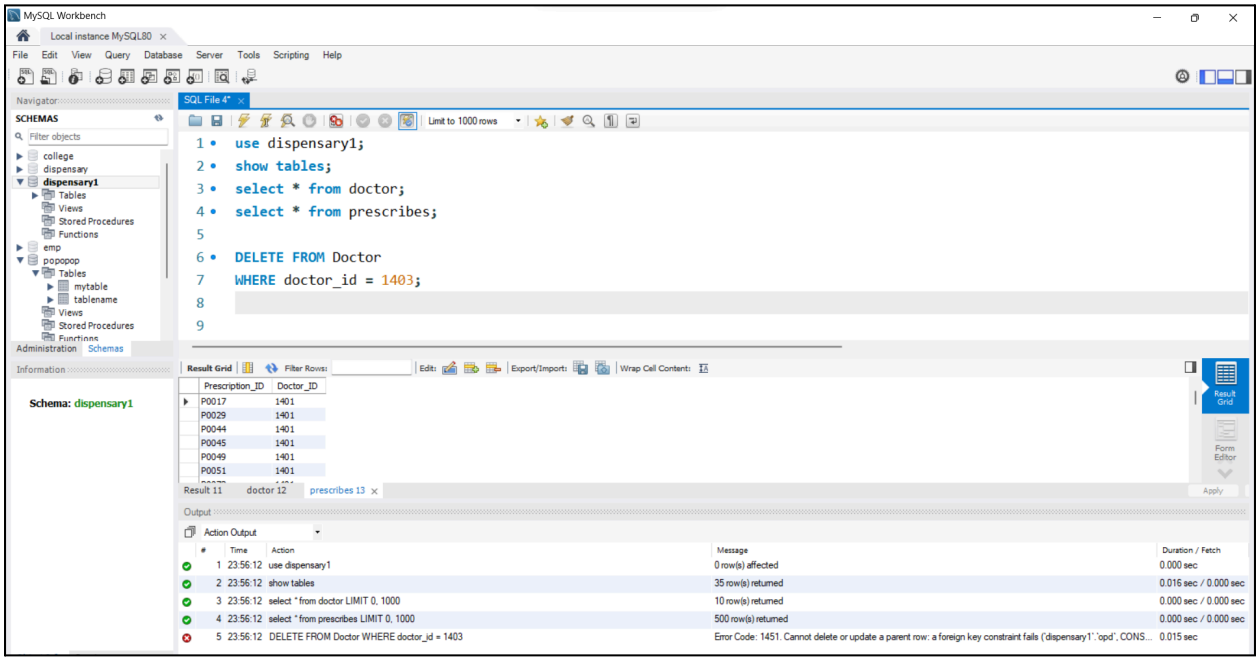
Why this error?

In the provided scenario, the SQL query attempts to update the doctor_id column in the "prescribes" table from 1403 to 1000. However, the prescribed table has a foreign key constraint that references the doctor_id column to the doctor table. This constraint ensures that every doctor_id value in the prescribed table corresponds to a valid doctor_id in the doctor table. Since there is no doctor_id with the value 1000 in the doctor table, the foreign key constraint is violated. As a result, the database prevents the update operation from being executed, and an error is raised, indicating that a child row (in the prescribed table) references a non-existent parent row (in the doctor table).

2] Deleting Referenced Primary Key Value

Ensuring referential integrity becomes essential when removing entries from a referenced table. Deleting a primary key value from a referenced table without propagating the deletion to related records in the referencing table results in integrity breaches. The referencing table still preserves references to the deleted primary key value, leading to errors and data inconsistencies.

Example in our case:



Error message that appeared:

Error Code: 1451. Cannot delete or update a parent row: a foreign key constraint fails (`dispensary1`.`opd`, CONSTRAINT `opd_ibfk_2` FOREIGN KEY (`Doctor_ID`) REFERENCES `doctor` (`Doctor_ID`))

Why this error?

The error occurs because there is a foreign key constraint defined in the prescribed table, specifically on the Doctor_ID column, which references the doctor table's Doctor_ID column. This constraint ensures referential integrity by enforcing that every Doctor_ID value in the prescribed table must exist in the doctor table. Similarly, other tables may have foreign key constraints referencing the Doctor_ID column in the doctor table like opd as shown in the error message.

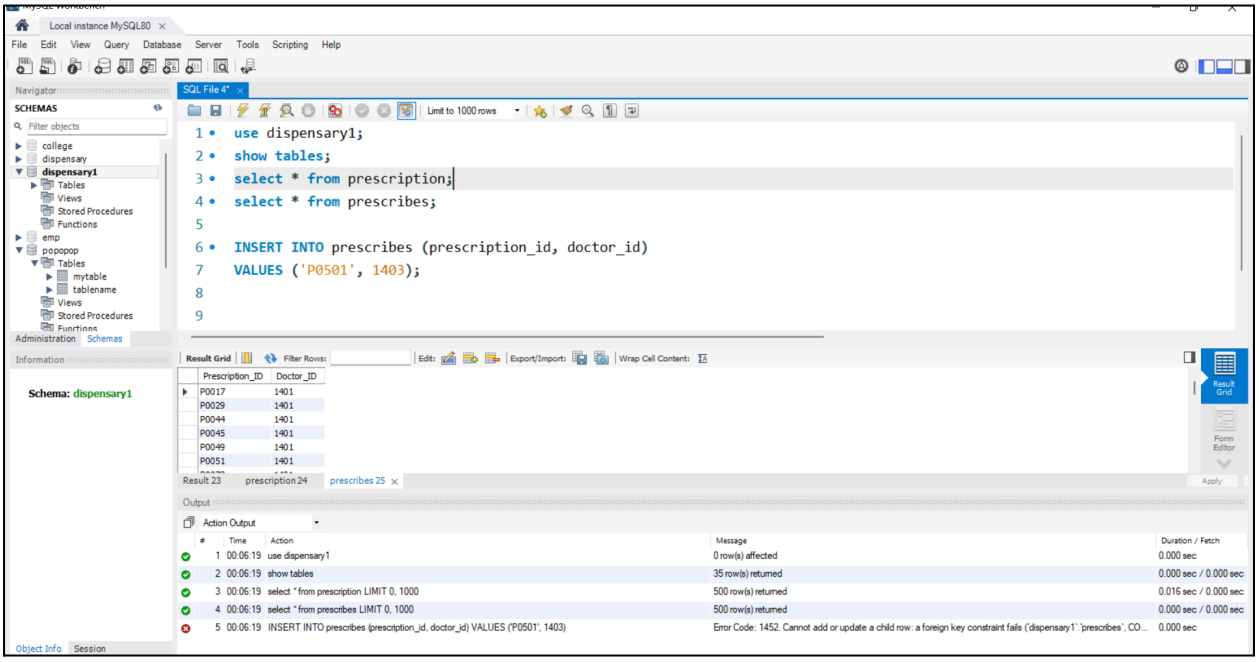
When we attempt to delete a row from the doctor table where doctor_id is 1403, the foreign key constraint prevents it because there are corresponding records in the prescribed table that reference this doctor_id. Deleting the row in the doctor table would leave orphaned references in the prescribed table, violating referential integrity.

To resolve this error, we need to first delete the related records from the prescribed table where Doctor_ID is 1403, and then we can proceed with deleting the row from the doctor table. This ensures that referential integrity is maintained throughout the database.

3] Inserting Non-existent Foreign Key Value

Within a relational database system, referential integrity may be breached during data insertion into a table containing foreign key references. If a new entry is added with a foreign key value absent in the corresponding referenced table, it triggers an error. This arises because the foreign key constraint mandates that all references must align with valid primary key values in the referenced table.

Example in our case:



Error message that appeared:

Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails (`dispensary1`.`prescribes`, CONSTRAINT `prescribes_ibfk_1` FOREIGN KEY (`Prescription_ID`) REFERENCES `prescription` (`Prescription_ID`))

Why this error?

The error occurs because there is a foreign key constraint defined in the prescribed table, specifically on the Prescription_ID column, which references the prescription table's Prescription_ID column. This constraint ensures referential integrity by enforcing that every Prescription_ID value in the “prescribes” table must exist in the prescription table.

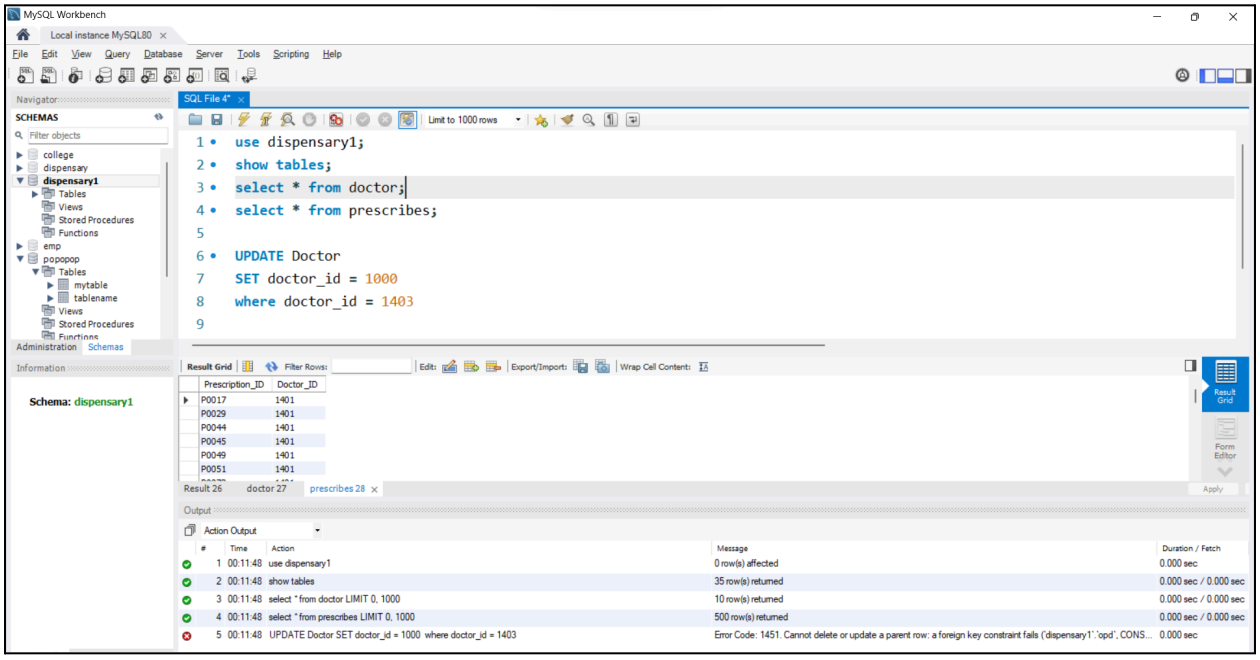
When we attempt to insert a new row into the “prescribes” table with prescription_id “P0501”, the foreign key constraint prevents it because there is no corresponding record with Prescription_ID “P0501” in the prescription table. The database expects that any prescription_id referenced in the prescribed table must already exist in the prescription table to maintain referential integrity.

To resolve this error, we would need to first ensure that a record with Prescription_ID “P0501” exists in the prescription table. Then we can proceed with inserting the new row into the “prescribes” table, referencing the existing Prescription_ID. This ensures that referential integrity is maintained throughout the database.

4] Updating Referenced Primary Key Value

Updating a primary key value in the referenced table without synchronizing updates with corresponding references in the referencing table can compromise integrity. If the table that's supposed to refer to the updated value still points to the old value, it messes up the data. This causes errors to prevent problems and keep the data consistent and correct.

Example in our case:



Error message that appeared:
Error Code: 1451. Cannot delete or update a parent row: a foreign key constraint fails (`dispensary1`.`opd`, CONSTRAINT `opd_ibfk_2` FOREIGN KEY (`Doctor_ID`) REFERENCES `doctor` (`Doctor_ID`))

Why this error?
The error occurs because there is a foreign key constraint defined in the prescribed table, specifically on the Doctor_ID column, which references the doctor table's Doctor_ID column. Similarly, other tables may have foreign key constraints referencing the Doctor_ID column in the doctor table, like opd as shown in the error message.

When we attempted to update a row in the doctor table, for example, changing the doctor_id from 1403 to 1000, the foreign key constraints in the prescribed table (and potentially other tables) prevented it. This is because there may be records in the prescribed table (and other tables) that reference the doctor_id 1403.

Updating the doctor_id would cause a mismatch between the referenced and referencing tables, violating referential integrity. To resolve this error, we need to first update or delete the related records in the referencing tables (such as prescribed) to ensure that the foreign key constraints are not violated. Then, we can proceed with updating the doctor's table. This ensures that referential integrity is maintained throughout the database.

What if we are okay with disregarding referential integrity constraints?

If we're okay with temporarily disregarding referential integrity constraints, we can disable foreign key checks. In MySQL, this can be achieved by setting the foreign_key_checks variable to 0. Here's how it can be done:

SET foreign_key_checks = 0;

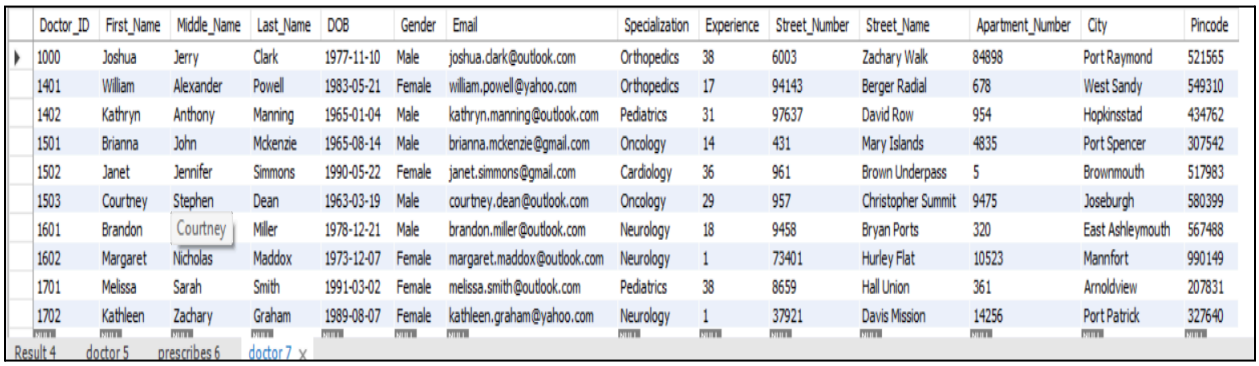
By setting foreign_key_checks to 0, MySQL will no longer enforce foreign key constraints, allowing us to perform operations that may violate referential integrity. However, it's crucial to exercise caution when disabling foreign key checks, as it can lead to data inconsistencies and integrity issues if not managed carefully.

Here's an example:

Code:

```
use dispensary1;
show tables;
set foreign_key_checks = 0;
select * from doctor;
select * from prescribes;
UPDATE Doctor
SET doctor_id = 1000
where doctor_id = 1403;
select * from doctor;
```

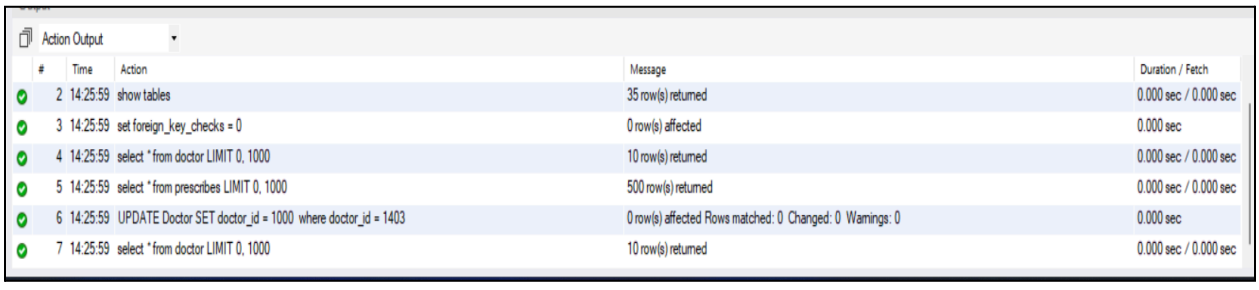
Entry with Doctor_id = 1000:



Doctor_ID	First_Name	Middle_Name	Last_Name	DOB	Gender	Email	Specialization	Experience	Street_Number	Street_Name	Apartment_Number	City	Pincode
1000	Joshua	Jerry	Clark	1977-11-10	Male	joshua.clark@outlook.com	Orthopedics	38	6003	Zachary Walk	84898	Port Raymond	521565
1401	William	Alexander	Powell	1983-05-21	Female	william.powell@yahoo.com	Orthopedics	17	94143	Berger Radial	678	West Sandy	549310
1402	Kathryn	Anthony	Manning	1965-01-04	Male	kathryn.manning@outlook.com	Pediatrics	31	97637	David Row	954	Hopkinsstad	434762
1501	Brianna	John	Mckenzie	1965-08-14	Male	brianna.mckenzie@gmail.com	Oncology	14	431	Mary Islands	4835	Port Spencer	307542
1502	Janet	Jennifer	Simmons	1990-05-22	Female	janet.simmons@gmail.com	Cardiology	36	961	Brown Underpass	5	Brownmouth	517983
1503	Courtney	Stephen	Dean	1963-03-19	Male	courtney.dean@outlook.com	Oncology	29	957	Christopher Summit	9475	Joseburgh	580399
1601	Brandon	Courtney	Miller	1978-12-21	Male	brandon.miller@outlook.com	Neurology	18	9458	Bryan Ports	320	East Ashleymouth	567488
1602	Margaret	Nicholas	Maddox	1973-12-07	Female	margaret.maddox@outlook.com	Neurology	1	73401	Hurley Flat	10523	Mannfort	990149
1701	Melissa	Sarah	Smith	1991-03-02	Female	melissa.smith@outlook.com	Pediatrics	38	8659	Hall Union	361	Arnoldview	207831
1702	Kathleen	Zachary	Graham	1989-08-07	Female	kathleen.graham@yahoo.com	Neurology	1	37921	Davis Mission	14256	Port Patrick	327640

Result 4 doctor 5 prescribes 6 doctor 7

No error message!



Action Output

#	Time	Action	Message	Duration / Fetch
2	14:25:59	show tables	35 row(s) returned	0.000 sec / 0.000 sec
3	14:25:59	set foreign_key_checks = 0	0 row(s) affected	0.000 sec
4	14:25:59	select * from doctor LIMIT 0, 1000	10 row(s) returned	0.000 sec / 0.000 sec
5	14:25:59	select * from prescribes LIMIT 0, 1000	500 row(s) returned	0.000 sec / 0.000 sec
6	14:25:59	UPDATE Doctor SET doctor_id = 1000 where doctor_id = 1403	0 row(s) affected Rows matched: 0 Changed: 0 Warnings: 0	0.000 sec
7	14:25:59	select * from doctor LIMIT 0, 1000	10 row(s) returned	0.000 sec / 0.000 sec

Explanation:

Here we tried to update the primary key value which is Doctor_id = 1403 value to 1000. Technically, it should have thrown an error, but here it didn't, as we have set the foreign_key_constraints to 0, so it will disregard any violation of referential integrity. Also we can see that tuples in prescribes table are still unaffected, ie the tuples having Doctor_id = 1403 are still present, clearly showing inconsistencies:

Prescription_ID	Doctor_ID
P0156	1403
P0168	1403
P0170	1403
P0171	1403
P0177	1403
P0178	1403
P0185	1403
P0207	1403
P0219	1403
P0228	1403

Solutions attempted to avoid the error message and avoiding the breach of referential integrity:

Firstly, we need to set the foreign_key_checks as 1 so that it throws an error whenever referential integrity is breached. But to avoid the error message, we can make use of cascading actions in case of deleting and updating of primary key value. Exactly, we can use the following instructions:

- ON DELETE CASCADE
- ON UPDATE CASCADE

Here's an example:

If we want to allow deletions and updations in the doctor table having its primary key, Doctor_ID, without violating foreign key constraints of all the referencing tables that have "Doctor_ID" has as their FOREIGN KEY, then we need to make these changes during its table creation:

(For example, we have shown the behavior of "Prescribes" table in which "Doctor_ID" is the Foreign Key)

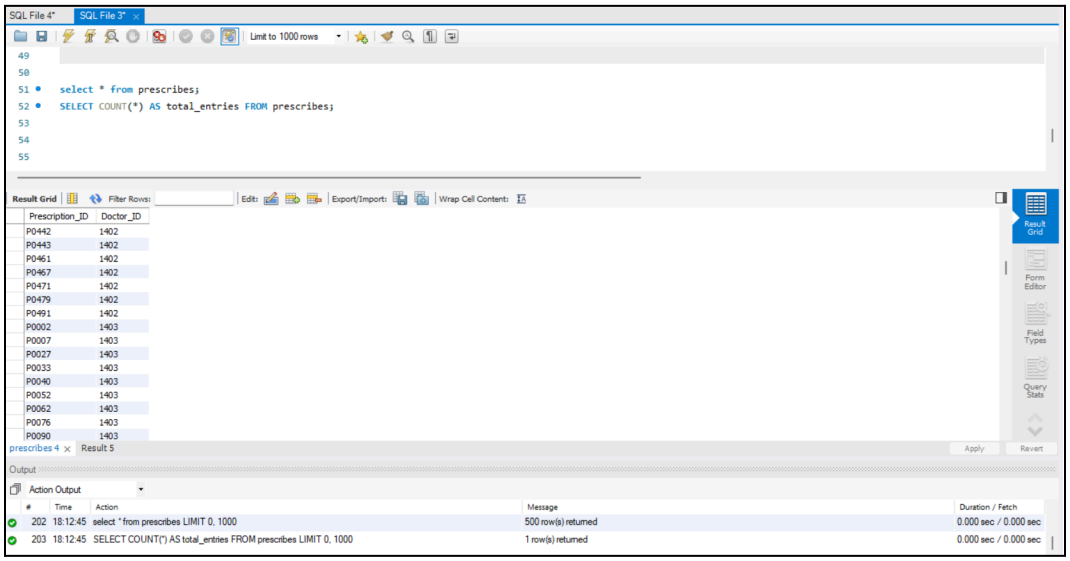
Code:

```
33
34 •  DROP TABLE IF EXISTS `prescribes`;
35 •  CREATE TABLE `prescribes` (
36      `Prescription_ID` varchar(20) NOT NULL,
37      `Doctor_ID` int DEFAULT NULL,
38      PRIMARY KEY (`Prescription_ID`),
39      KEY `Doctor_ID` (`Doctor_ID`),
40      CONSTRAINT `prescribes_ibfk_1` FOREIGN KEY (`Prescription_ID`) REFERENCES `prescription` (`Prescription_ID`)
41      ON DELETE CASCADE
42      ON UPDATE CASCADE,
43      CONSTRAINT `prescribes_ibfk_2` FOREIGN KEY (`Doctor_ID`) REFERENCES `doctor` (`Doctor_ID`)
44      ON DELETE CASCADE
45      ON UPDATE CASCADE
46    );
```

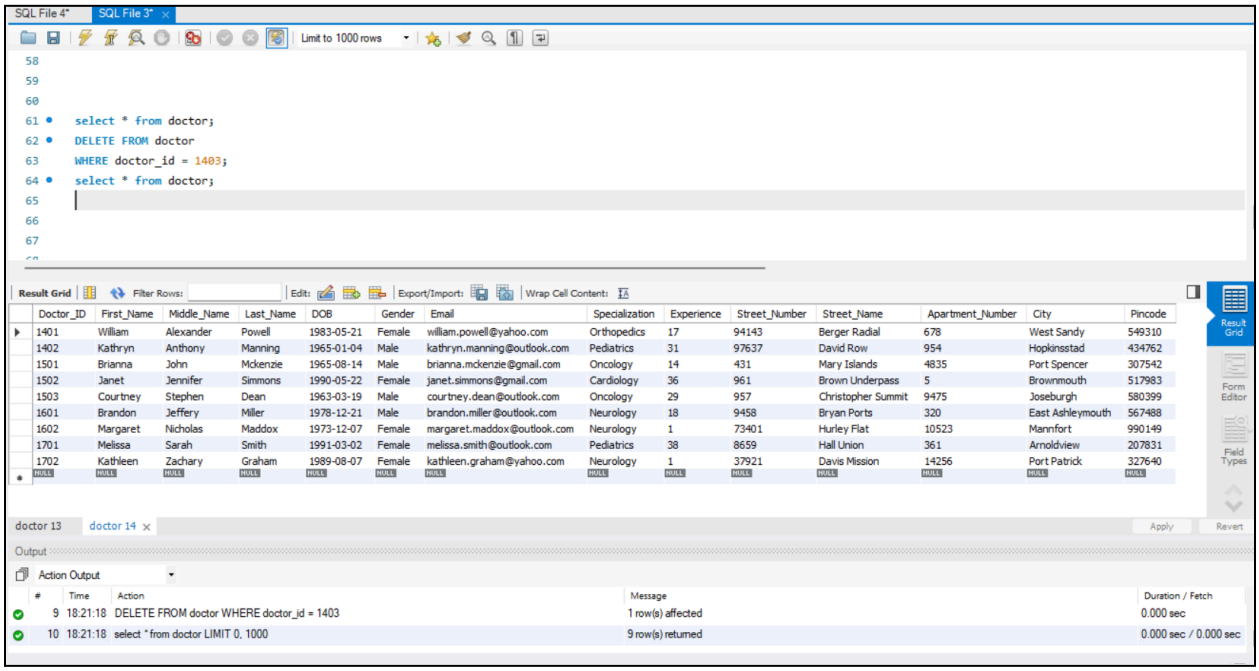
Now we tried to do these operations and checked the results.

Deleting Referenced Primary Key Value

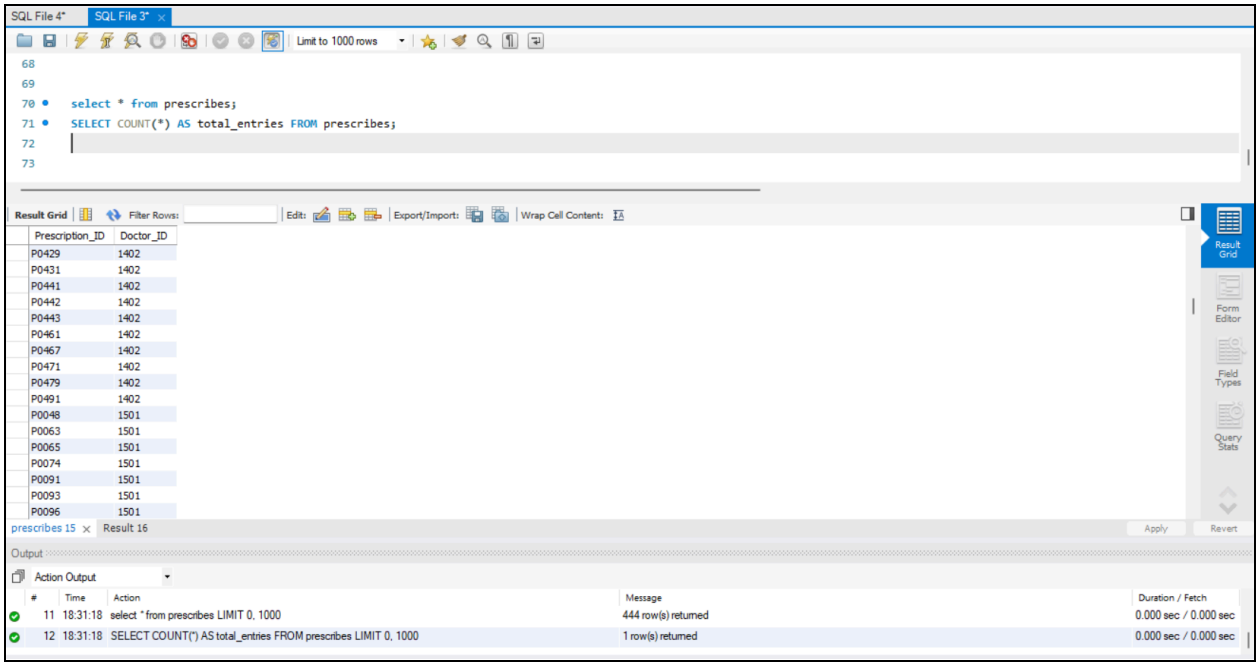
Initially, there are 500 entries in the ‘prescribes’ table, where Doctor_ID is the Foreign Key.



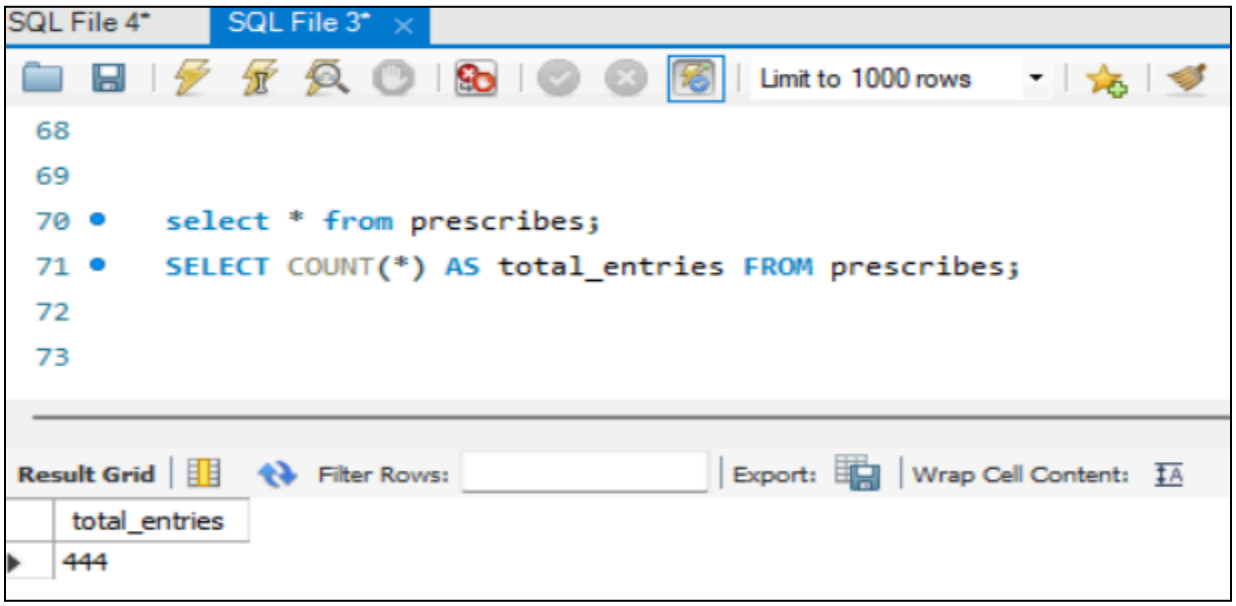
Then, we deleted the tuple with Doctor_ID = 1403 from the Parent ‘Doctor’ table, the operation successfully gets executed without any breaches/errors related to Foreign Key Constraints.



Since we have used the ‘ON DELETE CASCADE & ON UPDATE CASCADE’ in ‘prescribes’ table, all the tuples having ‘Doctor_ID = 1403’ get deleted from ‘prescribes’ table too and thus total entries got reduced to ‘444’ from ‘500’.

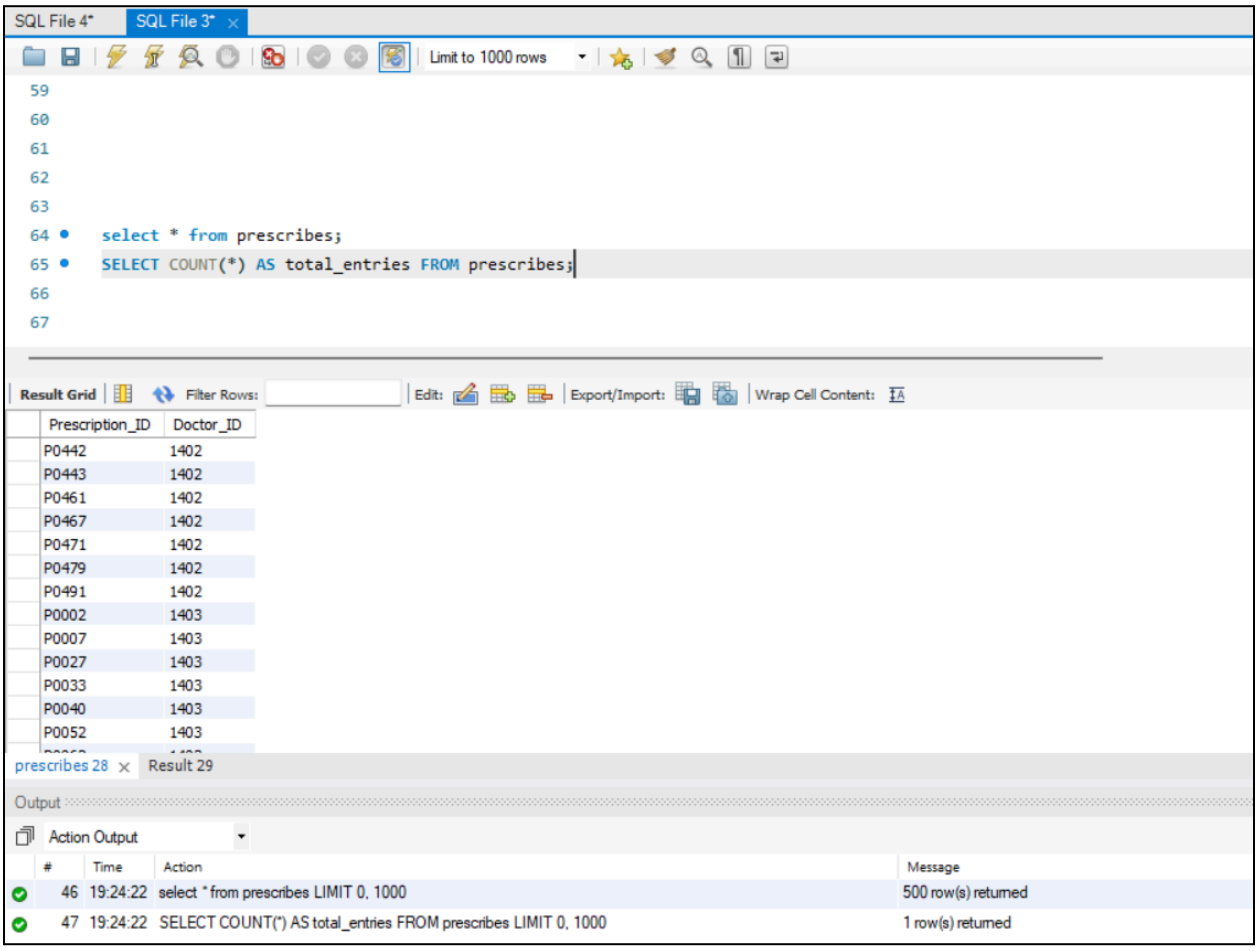


Total entries also get reduced to 444.

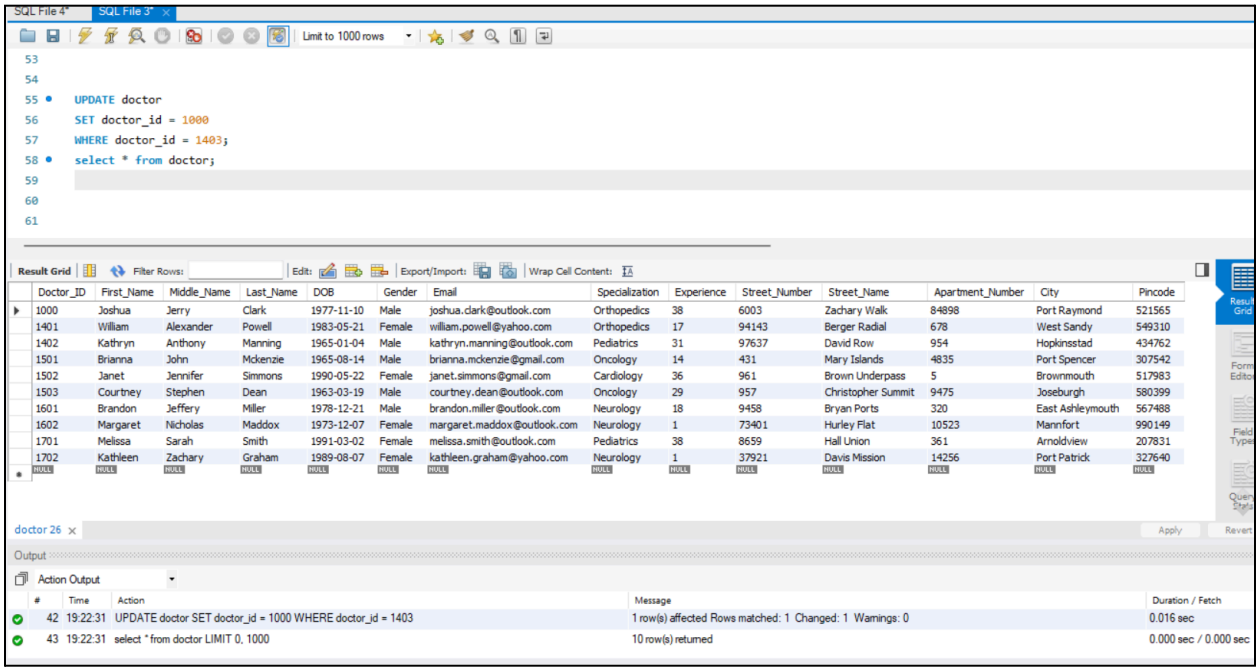


Updating Referenced Primary Key Value

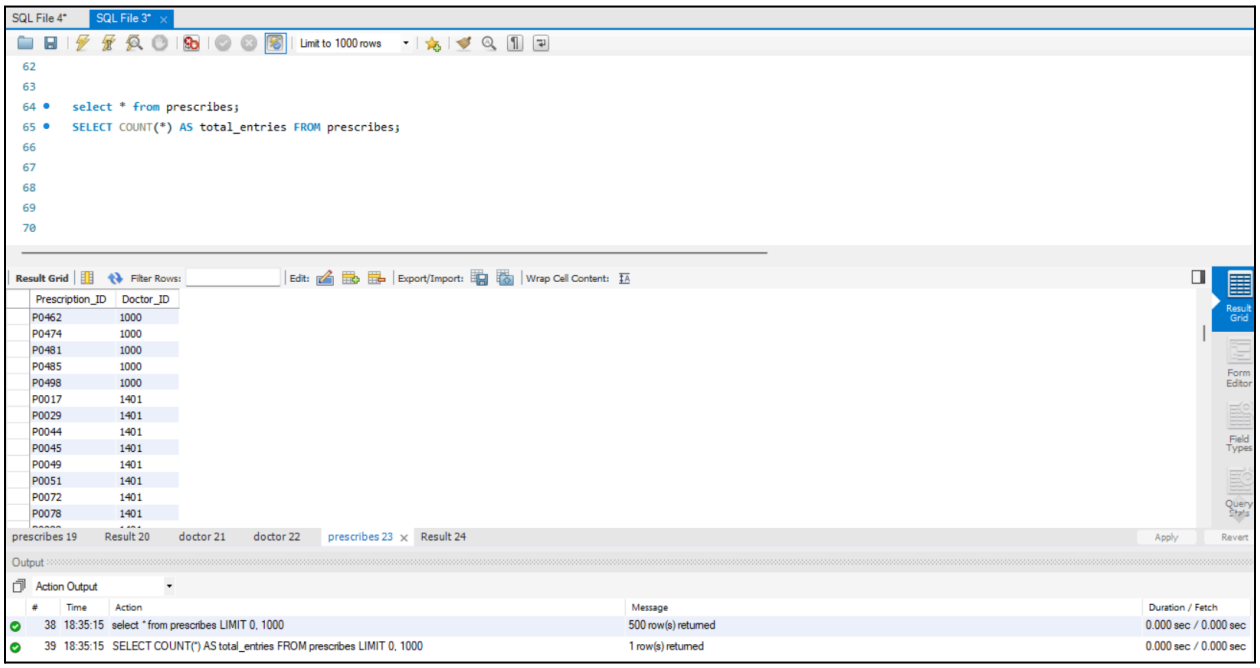
Initially, the ‘prescribes’ table has all the entries aligned with the ‘doctor’ table.



Then, we updated the value of an entry in the parent ‘Doctor’ table i.e.,
replaced Doctor_ID = 1403 with 1000. This operation gets executed without breaching referential integrity or throwing any errors.



The new Doctor_ID gets updated in the ‘Prescribes’ table too.



3.3 Responsibility of G1 & G2:

Operation - 1

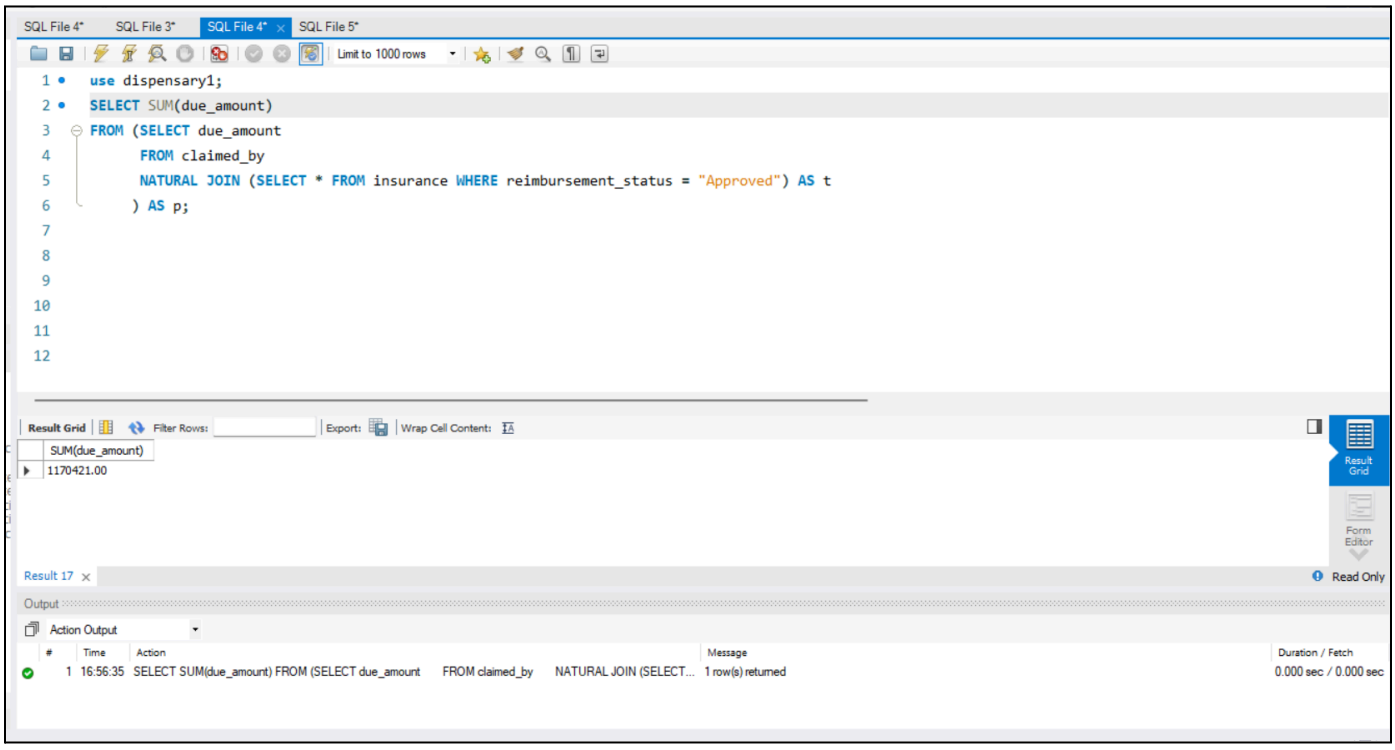
SQL Query:
SELECT SUM(due_amount)
FROM (SELECT due_amount FROM claimed_by
NATURAL JOIN (SELECT * FROM insurance WHERE reimbursement_status = "Approved") AS t) AS p;

Function:
It gives the sum of all the approved reimbursement amounts claimed by patients for insurance using natural join

Relational algebra:

$$t \leftarrow \sigma_{reimbursement\_status = "Approved"}(insurance)$$
$$p \leftarrow t \bowtie claimed\_by$$
$$\gamma_{sum(due\_amount)}(\pi_{due\_amount}(p))$$

Screenshot:



Operation - 2

SQL Query:

insert into prescribes(prescription_ID, Doctor_ID) values ('P0501',1050);

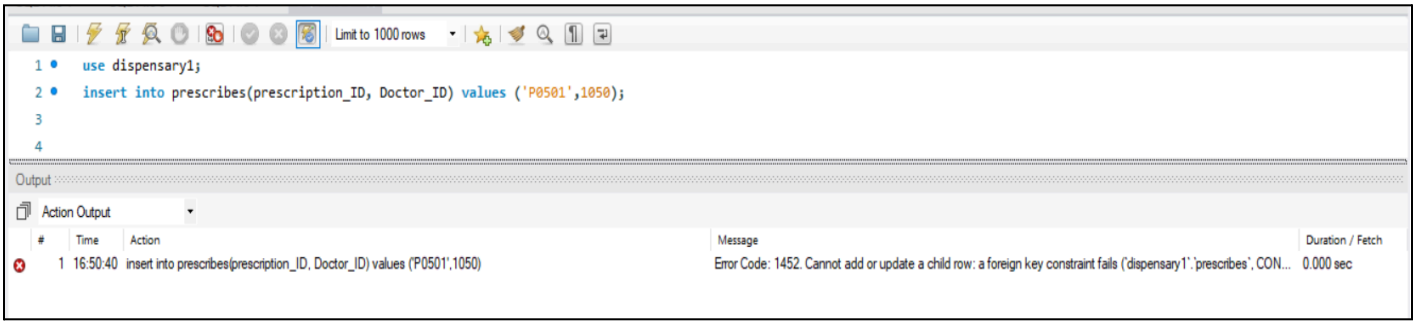
Function:

This query is executed to demonstrate that inserting a record into the prescribes table with a Doctor_ID value (1050 in this case) that does not exist in the parent table (Doctor) would lead to a foreign key constraint violation error. This illustrates the importance of ensuring data integrity through foreign key constraints.

Relational algebra:

prescribes ← prescribes ∪ {('P0501', 1050)}

Screenshot:



Operation - 3

SQL Query:

select o.doctor_id, count(o.patient_id) as number_of_patient_checked from
doctor as d
left outer join opd as o
on d.doctor_id = o.doctor_id where o.date>'2023-11-01' and o.date <'2023-12-01' group by(o.doctor_id) ;

Function: This function counts the number of patients checked by the doctor between '2023-11-01' and '2023-12-01' these dates.

Relational algebra:

left_join ← ρ_d(doctor) ⋈ ρ_o(opd)
filter ← $\sigma_{o.date > '2023-11-01' \wedge o.date < '2023-12-01'}$(left_join)
grouped ← $\gamma_{o.doctor\_id,\ count(o.patient\_id)\ AS\ number\_of\_patients\_checked}$(filter)
$\pi_{o.doctor\_id,\ number\_of\_patients\_checked}$(grouped)

Screenshot:

```
14
15 •  select o.doctor_id, count(o.patient_id) as number_of_patient_checked from
16    doctor as d
17    left outer join opd as o
18    on d.doctor_id = o.doctor_id where o.date>'2023-11-01' and o.date <'2023-12-01'
19    group by(o.doctor_id) ;
```

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

doctor_id	number_of_patient_checked
1601	36
1401	35
1502	31
1602	39
1701	41
1702	26

Operation - 4

SQL Query:

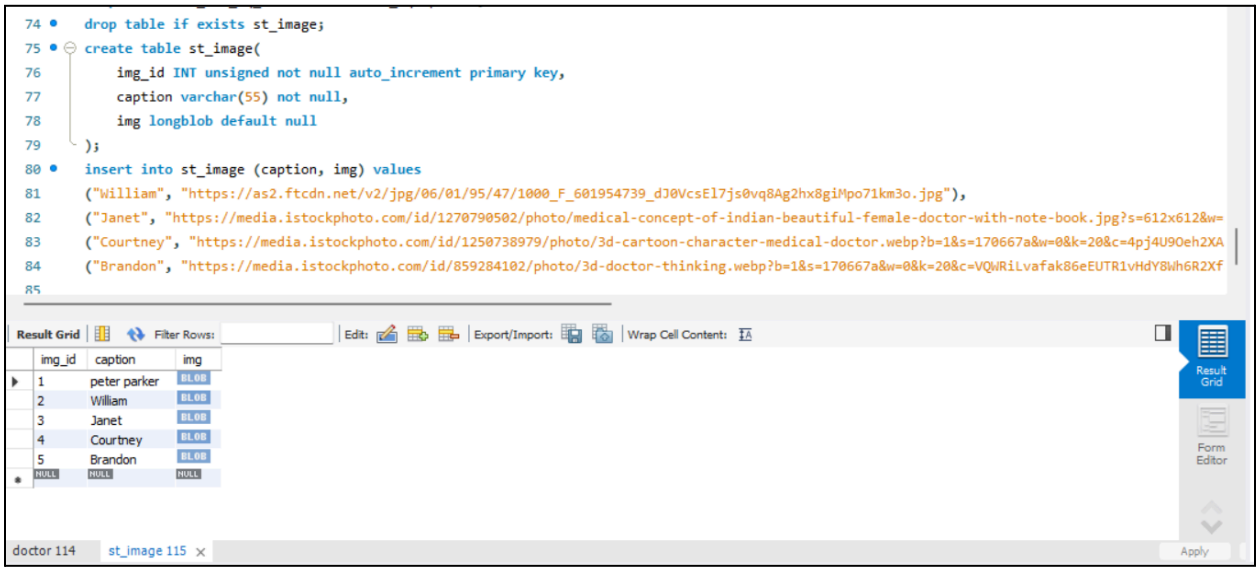
```
create table st_image(
   img_id INT unsigned not null auto_increment primary key,
    caption varchar(55) not null,
    img longblob default null
    );
insert into st_image(caption,img) values
("peter
parker","https://as2.ftcdn.net/v2/jpg/06/01/95/47/1000_F_601954739_dJ0VcsEl7js0vq8Ag2hx8giMpo71k
m3o.jpg"),
("William","https://media.istockphoto.com/id/1270790502/photo/medical-concept-of-indian-beautiful-fema
le-doctor-with-note-book.jpg?s=612x612&w=is&k=20&c=HIoD2eo-sfO53FsvZsboE37N-I2RYmB5yloCr7
jNBm0="),
("Janet","https://media.istockphoto.com/id/1363852481/photo/3d-render-cartoon-character-young-caucasian
-woman-doctor-holds-clipboard-with-blank-page.jpg?s=1024x1024&w=is&k=20&c=qEwRxuAiej4QzcT9c
ULV1Z5qcwevLM6Rds0Bifr-iHM="),
("Courtney","https://media.istockphoto.com/id/1314984668/photo/3d-render-character-doctor-attention-doc
tor-of-medicine-woman-wearing-mask-gloves-points-up.webp?b=1&s=170667a&w=0&k=20&c=8BPObPa
LSHvDbyp45TK27CaeT3bKvUKVJAYr_IU0nOc="),
("Brandon","https://media.istockphoto.com/id/1250738979/photo/3d-cartoon-character-medical-doctor.web
p?b=1&s=170667a&w=0&k=20&c=4pj4U9Oeh2XAJIUBVz4IQ_gFcLMmsVn6Vj2LC8_UCZs=");
```

Function: For storing images we have a table named st_images and it will store the images of the doctor in the table.

Relational algebra:

$$st\_images \leftarrow st\_images \cup \{("william", "...")\} \cup \{("Janet", "...")\} \cup \{("Courtney", "...")\} \cup \{("Brandon", "...")\}$$

Screenshot:



Operation - 5

SQL Query:

```
UPDATE staff
SET salary =
CASE
   when join_date =  (select min(join_date) from staff) then salary*1.5
   when join_date < '2018-12-31' then salary * 1.1
    when join_date < '2020-12-31' then salary * 1.05
    ELSE salary
END;
```

Function: It checks the join_date of staff, if it's less than 2018-12-31 then salary is incremented by 10%, else if it's less than 2020-12-31 the salary is incremented by 5% and salary of most senior staff members is incremented by 15%. This functions throws an error because we can not refer to the same table in the case statement which is referred as error code: 1093

Relational algebra:

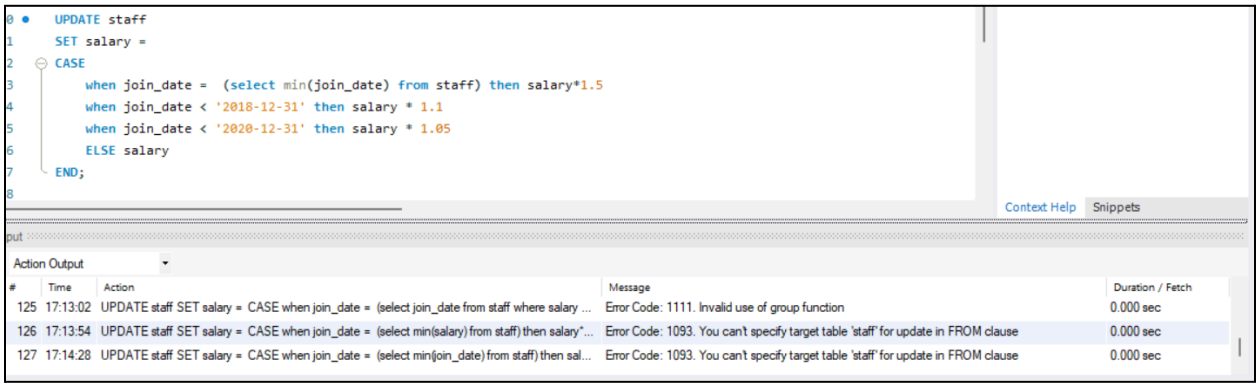
original_staff ← $\pi_{staff\_id, salary, join\_date}$

min_join_date ← $\pi_{join\_date}(\sigma_{min(join\_date)}(original\_staff))$

updated_salary ← $\pi_{staff\_id,}$ (CASE
WHEN join_date = min_join_date THEN salary * 1.5
WHEN join_date < '2018-12-31' THEN salary * 1.1
WHEN join_date < '2020-12-31' THEN salary * 1.05
ELSE salary
END) AS salary
(original_staff)

final_staff ← (staff − salary) ∪ updated_salary

Screenshot:



Contributions

Name	Groups	Contribution
Darsh Dalal	G2	Data population of entity sets and relationship sets tables using faker library in python; took care of cardinality and participation constraints while populating relationship sets tables and also checked for aforementioned constraints for entity sets tables. Wrote the answer for referential integrity explaining the situations where the same was breached and also gave possible solutions to keep the integrity of the database intact. Wrote Relational algebra for operations of G1+G2.

Ishika Raj	G2	Data population for entity sets and relationship sets using python code. Wrote SQL queries for creating user, giving select, update and delete permissions to user. Created views with the required constraints.
Twinkle Devda	G2	Populated data for entity sets and relationship sets using the python code while taking care of the various constraints. Formed the tables for all the sets in mySQL. Worked on writing the sql queries for creating users, and giving permissions and revoking them for tables and view and did documentation for the same. Wrote relational algebra for operations in G1+G2.
Saumya Jaiswal	G2	Populated the tables for the entity sets and the relationship sets using python code keeping the constraints in check. Formed the tables for all the sets in mySQL, worked on the solution of the referential integrity question, implementing different methods and verifying through different examples using mySQL. Wrote relational algebra for operations in G1+G2.
Shreya Patel	G1	The data population complied with the assignment's stated restrictions, provided examples to clarify the maintenance of ACID properties using MySQL, participated in indexing entity sets,wrote case statement, and created and described queries that threw errors and stored images,helped in documentation.
Riya Jain	G1	Populated the data following all the constraints given, explained and implemented the indexing of entity sets and table extensions, wrote the explanation for ACID properties along with the examples, implemented mysql queries which will throw error and store images.
Saurabh Kumar Sah	G1	Created tables and Populated them using CSV file, Provided example on clarification for ACID property, Did indexing of the tables, created the SQL queries for inner and left outer join, aggregation and query which threw an error and store image. Helped in the documentation.
Het Trivedi	G1	Wrote SQL code for creating tables and populated them with CSV files, Did indexing for the table and explained them, created SQL queries for inner and left outer join, wrote case statement, wrote a query which threw an error and stored the image, Wrote the SQL queries for updating the table, Made updated ER diagram, the Helped in the documentation.