

## CS 432 Databases

### Assignment II Developing the DBMS

#### QUERY CRAFTERS

##### Group Members

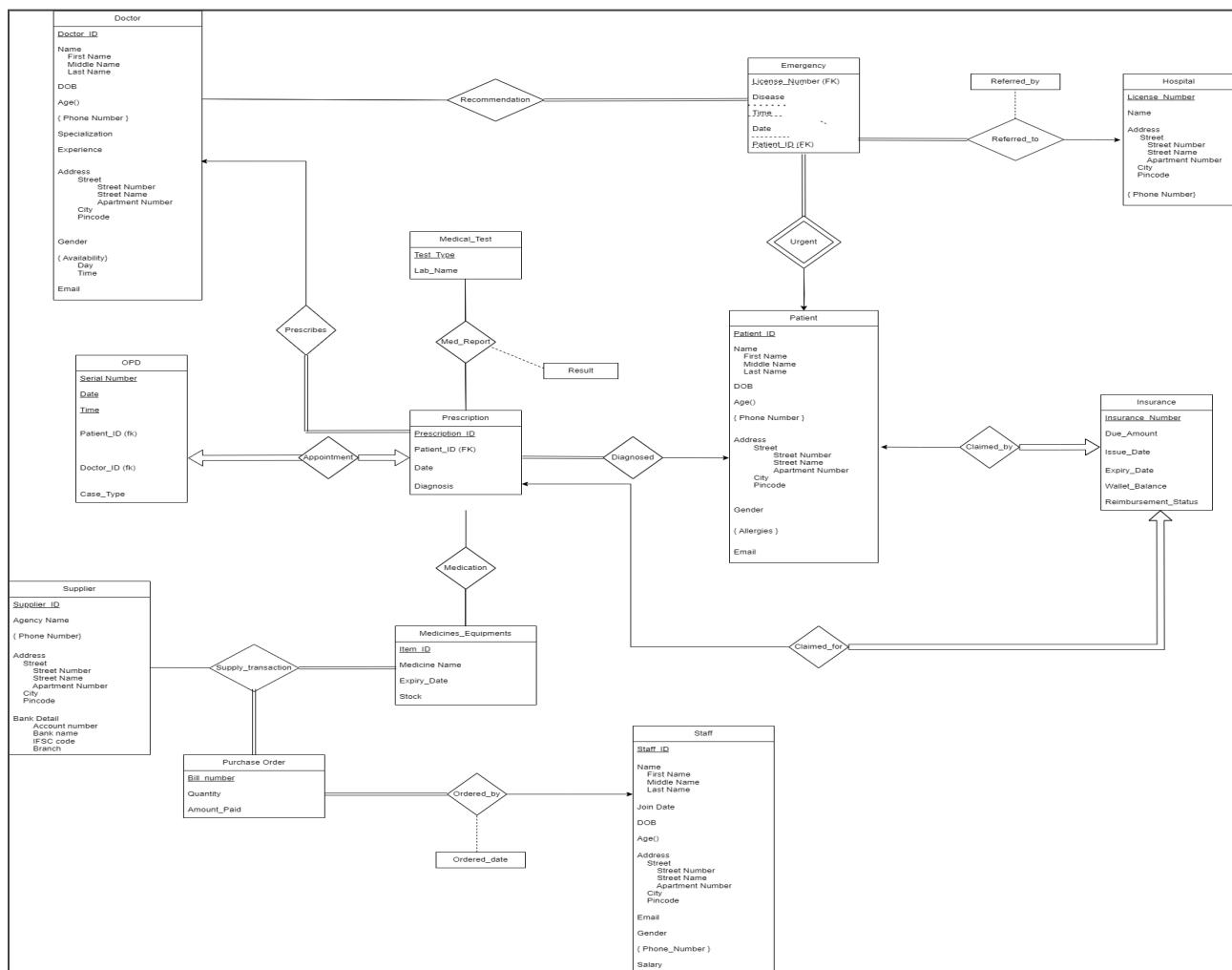
Saurabh Kumar Sah	21110188
Shreya Patel	21110155
Riya Jain	21110178
Twinkle Devda	21110228
Ishika Raj	21110081
Saumya Jaiswal	21110186
Darsh Dalal	21110049
Het Trivedi	21110226

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:

The screenshot shows the MySQL Workbench interface. In the top navigation bar, the active tab is 'Query 1' with the title 'sql\_practice\_section1'. The code pane contains the following SQL statements:

```

2 • show tables;
3 • select * from doctor;
4
5 • create index idx_doc_gen on Doctor(Gender);
6 • select * from Doctor where Gender = 'Female';
7 drop index idx_doc_gen on Doctor;

```

The 'Result Grid' pane shows the output of the 'select' statement. The 'EXPLAIN' pane displays the execution plan, which includes a 'Table scan on Doctor' operation.

In the bottom pane, the 'Result 18' section shows the history of actions taken during the session, including the creation and dropping of the index.

## With indexing:

The screenshot shows the MySQL Workbench interface. The active tab is 'Query 1' with the title 'sql\_practice\_section1'. The code pane contains the same SQL statements as the previous screenshot, but the execution plan is different due to the presence of the index:

```

2 • show tables;
3 • select * from doctor;
4
5 • create index idx_doc_gen on Doctor(Gender);
6 • select * from Doctor where Gender = 'Female';
7 explain analyze select * from Doctor where Gender = 'Female';
8 drop index idx_doc_gen on Doctor;
9 select * from doctor;
10 DROP TABLE IF EXISTS Doctor_Contact;
11 CREATE TABLE Doctor_Contact (
    ...
);

```

The 'Result Grid' pane shows the output of the 'select' statement. The 'EXPLAIN' pane displays the execution plan, which includes a 'Filter' operation followed by a 'Table scan on Doctor' operation.

- 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:

```

MySQL Workbench
File Edit View Query Database Server Tools Scripting Help
Navigator: Schemas
SCHEMAS
Tables
doctor
doctor_availability
doctor_contact
hospital
hospital_contact
medical_test
medicine_equipment
Columns
Indexes
Foreign Keys
Triggers
opd
patient
Administration Schemas
Information

Query 1 sql_practice_section1 SQL File 2* SQL File 5* Administration - Data Import/Res... SQL File 6* x
3 • select * from medicine_equipment;
4
5 • create index idx_med_equip on medicine_equipment(Stock);
6 • select * from medicine_equipment where Stock = '94';
7 explain analyze select * from medicine_equipment where Stock = '94';
8 • drop index idx_med_equip on medicine_equipment

Result Grid Filter Rows: Export: Wrap Cell Content: 
EXPLAIN
-> Filter: (medicine_equipment.Stock = 94) (cost=11.2 rows=11) (actual time=0.148..0.209 rows=14 loops=1)
-> Table scan on medicine_equipment (cost=11.2 rows=110) (actual time=0.13..0.19 rows=110 loops=1)

```

### With indexing:

```

MySQL Workbench
File Edit View Query Database Server Tools Scripting Help
Navigator: Schemas
SCHEMAS
Tables
doctor
doctor_availability
doctor_contact
hospital
hospital_contact
medical_test
medicine_equipment
Columns
Indexes
Foreign Keys
Triggers
opd
patient
Administration Schemas
Information

Query 1 sql_practice_section1 SQL File 2* SQL File 5* Administration - Data Import/Res... SQL File 6* x
3 • select * from medicine_equipment;
4
5 • create index idx_med_equip on medicine_equipment(Stock);
6 • select * from medicine_equipment where Stock = '94';
7 explain analyze select * from medicine_equipment where Stock = '94';
8 • drop index idx_med_equip on medicine_equipment

Result Grid Filter Rows: Export: Wrap Cell Content: 
EXPLAIN
-> Index lookup on medicine_equipment -> Index lookup on medicine_equipment using idx_med_equip (Stock=94) (cost=2.15 rows=14) (actual time=0.141..0.152 rows=14 loops=1)

```

- 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:

```

118 License_Number INT,
119 Phone_Number BIGINT CHECK (Phone_Number >= 1000000000 AND Phone_Number < 1000000000),
120 PRIMARY KEY (License_Number, Phone_Number),
121 FOREIGN KEY (License_Number) REFERENCES Hospital(License_Number)
122 ;
123 • create index idx_hosc_llicen on Hospital_Contact(License_Number);
124 • select * from Hospital_Contact where License_Number = '113643';
125 • explain analyze select * from Hospital_Contact where License_Number = '113643';
126 • drop index idx_hosc_llicen on Hospital_Contact;
127 • select * from Hospital_Contact;
128

Result Grid Filter Rows: Export: Wrap Cell Content: 
EXPLAIN
-> Covering index look -> Covering index lookup on Hospital_Contact using PRIMARY (License_Number=113643) (cost=0.451 rows=2) (actual time=0.0267..0.0343 rows=2 loops=1)

```

#### With Indexing:

```

118     License_Number INT,
119     Phone_Number BIGINT CHECK (Phone_Number >= 1000000000 AND Phone_Number < 1000000000),
120     PRIMARY KEY (License_Number, Phone_Number),
121     FOREIGN KEY (License_Number) REFERENCES Hospital(License_Number)
122   );
123 •   create index idx_hosc_licen on Hospital_Contact(License_Number);
124 •   select * from Hospital_Contact where License_Number = '113643';
125 •   explain analyze select * from Hospital_Contact where License_Number = '113643';
126 •   drop index idx_hosc_licen on Hospital_Contact;
127 •   select * from Hospital_Contact;
128

```

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

EXPLAIN  
-> Covering index lookup on H -> Covering index lookup on Hospital\_Contact using idx\_hosc\_licen (License\_Number=113643) (cost=0.45 rows=2) (actual time=0.0426..0.0517 rows=2 loops=1)

- 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:

```

108     CITY VARCHAR(100) NOT NULL,
109     Pincode INT CHECK (Pincode >= 100000 AND Pincode <= 999999)
110   );
111 •   select * from Hospital;
112 •   create index idx_Hos_city on Hospital(City);
113 •   select * from Hospital where City = 'Michelleshire';
114 •   explain analyze select * from Hospital where City = 'Michelleshire';
115 •   drop index idx_Hos_city on Hospital;
116 •   Drop table if exists Hospital_contact;
117 •   CREATE TABLE Hospital_contact (
118     License_Number INT,

```

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

EXPLAIN  
-> Filter: (hospital.City = 'Michelleshire') (cost=...  
-> Filter: (hospital.City = 'Michelleshire') (cost=1.45 rows=12) (actual time=0.0758..0.0941 rows=2 loops=1)  
-> Table scan on Hospital (cost=1.45 rows=12) (actual time=0.0705..0.085 rows=12 loops=1)

### With indexing:

```

108     CITY VARCHAR(100) NOT NULL,
109     Pincode INT CHECK (Pincode >= 100000 AND Pincode <= 999999)
110   );
111 •   select * from Hospital;
112 •   create index idx_Hos_city on Hospital(City);
113 •   select * from Hospital where City = 'Michelleshire';
114 •   explain analyze select * from Hospital where City = 'Michelleshire';
115 •   drop index idx_Hos_city on Hospital;
116 •   Drop table if exists Hospital_contact;
117 •   CREATE TABLE Hospital_contact (
118     License_Number INT,

```

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

EXPLAIN  
-> Index lookup on Hospital u -> Index lookup on Hospital using idx\_Hos\_city (City='Michelleshire') (cost=0.7 rows=2) (actual time=0.0583..0.0725 rows=2 loops=1)

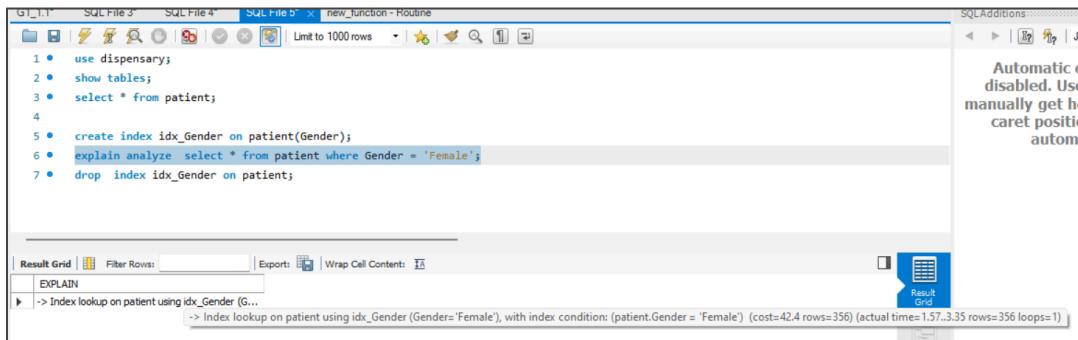
- Cost without Indexing = 1.45
- Cost with Indexing = 0.7

## 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:**



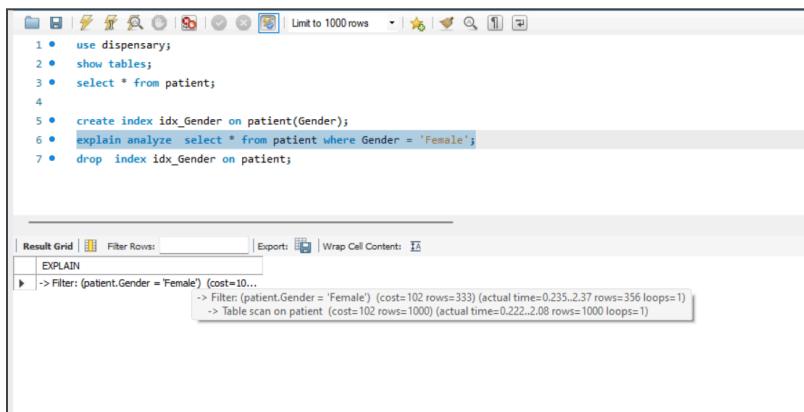
A screenshot of MySQL Workbench showing a SQL editor window. The code is as follows:

```
1 • use dispensary;
2 • show tables;
3 • select * from patient;
4
5 • create index idx_Gender on patient(Gender);
6 • explain analyze select * from patient where Gender = 'Female';
7 • drop index idx_Gender on patient;
```

The EXPLAIN ANALYZE output shows:

```
EXPLAIN
-> Index lookup on patient using idx_Gender (G...
-> Index lookup on patient using idx_Gender (Gender='Female'), with index condition: (patient.Gender = 'Female') (cost=42.4 rows=356) (actual time=1.57..3.35 rows=356 loops=1)
```

### **Without indexing:**



A screenshot of MySQL Workbench showing a SQL editor window. The code is identical to the one above:

```
1 • use dispensary;
2 • show tables;
3 • select * from patient;
4
5 • create index idx_Gender on patient(Gender);
6 • explain analyze select * from patient where Gender = 'Female';
7 • drop index idx_Gender on patient;
```

The EXPLAIN ANALYZE output shows:

```
EXPLAIN
-> Filter: (patient.Gender = 'Female') (cost=102 rows=333) (actual time=0.235..2.37 rows=356 loops=1)
  -> Filter: (patient.Gender = 'Female') (cost=102 rows=333) (actual time=0.235..2.37 rows=356 loops=1)
  -> Table scan on patient (cost=102 rows=1000) (actual time=0.222..2.08 rows=1000 loops=1)
```

- 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:**

```

138     Pincode INT CHECK (Pincode >= 100000 AND Pincode <= 999999),
139     Email VARCHAR(255) NOT NULL unique,
140     Gender ENUM('Male', 'Female', 'Other') NOT NULL,
141     Salary INT NOT NULL
142   );
143
144 •   create index idx_Staff_gen on Staff(Gender);
145 •   select * from Staff where Gender = 'Male';
146 •   explain analyze select * from Staff where Gender = 'Male';
147 •   drop index idx_Staff_gen on Staff;
148

```

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

EXPLAIN

```

-> Filter: (staff.Gender = 'Male') (cost=2.25 rows=...
-> Filter: (staff.Gender = 'Male') (cost=2.25 rows=6.67) (actual time=0.327..0.349 rows=9 loops=1)
-> Table scan on Staff (cost=2.25 rows=20) (actual time=0.317..0.338 rows=20 loops=1)

```

### With indexing:

```

141     Salary INT NOT NULL
142   );
143
144 •   create index idx_Staff_gen on Staff(Gender);
145 •   select * from Staff where Gender = 'Male';
146 •   explain analyze select * from Staff where Gender = 'Male';
147 •   drop index idx_Staff_gen on Staff;
148
149 •   select * from staff;
150 •   DROP TABLE IF EXISTS Staff_Contact;
151 •   CREATE TABLE Staff_Contact (

```

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

EXPLAIN

```

-> Index lookup on Staff using idx_Staff_gen (-> Index lookup on Staff using idx_Staff_gen (Gender='Male'), with index condition: (staff.Gender = 'Male') (cost=1.4 rows=9) (actual time=0.0878..0.104 rows=9)

```

- 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:

```

42
43 •   create index idx_Staff_gen on Staff(Gender);
44 •   select * from Staff where Gender = 'Male';
45 •   explain analyze select * from Staff where Gender = 'Male';
46
47
48 •   create index idx_sup_Ban on Supplier(Bank_Name);
49 •   select * from supplier where Bank_Name = 'XYZ Bank';
50 •   explain analyze select * from supplier where Bank_Name = 'XYZ Bank';
51
52 •   create index idx_Emeg_Dis on Emergency(Disease);
53 •   select * from Emergency where Disease = 'Meningitis';
54 •   explain analyze select * from Emergency where Disease = 'Meningitis';

```

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

EXPLAIN

```

-> Filter: (emergency.Disease = 'Meningitis') (cost=5.25 rows=5) (actual time=0.056 L..0.064 rows=3 loops=1)
-> Covering index scan on Emergency using License_Number (cost=5.25 rows=50) (actual time=0.0486..0.0596 rows=50 loops=1)

```

### With indexing:

The screenshot shows a MySQL Workbench interface with several SQL statements in the query editor and their corresponding EXPLAIN results in the results grid.

```

42 • create index idx_Staff_gen on Staff(Gender);
43 • select * from Staff where Gender = 'Male';
44 • explain analyze select * from Staff where Gender = 'Male';
45 •
46
47
48 • create index idx_sup_Ban on Supplier(Bank_Name);
49 • select * from supplier where Bank_Name = 'XYZ Bank';
50 • explain analyze select * from supplier where Bank_Name = 'XYZ Bank';
51
52 • create index idx_Emeg_Dis on Emergency(Disease);
53 • select * from Emergency where Disease = 'Meningitis';
54 • explain analyze select * from Emergency where Disease = 'Meningitis';
55

```

**Result Grid:**

```

EXPLAIN
-> Covering index lookup on Emergency using idx_Emeg_Dis
(Disease='Meningitis') (cost=0.595 rows=3) (actual time=0.0449..0.0542)
Rows=3 loops=1

```

- 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:

The screenshot shows a MySQL Workbench interface with several SQL statements in the query editor and their corresponding EXPLAIN results in the results grid.

```

1 • use dispensary1;
2 • show tables;
3 • select * from medication;
4 • select * from medicines_equipments;
5 • select * from supply_transaction;
6 • show Indexes from OPD;
7 • select * from OPD;
8 • create index Idx_OPD_type on OPD(Case_Type);
9 • select * from OPD where Case_Type='Old';
10 • explain analyze select * from OPD where Case_Type='Old';
11 • drop index Idx_OPD_type on OPD;
12 • drop table if exists senior_specialist;
13 • create table senior_specialist as (SELECT * FROM doctor WHERE doctor.Experience > '20');
14

```

**Result Grid:**

```

EXPLAIN
-> Filter: (opd.Case_T -> Filter: (opd.Case_Type = 'Old') (cost=50.8 rows=50) (actual time=0.0841..0.352 rows=134 loops=1)
-> Table scan on OPD (cost=50.8 rows=500) (actual time=0.0703..0.287 rows=500 loops=1)

```

### With indexing:

```

1 • use dispensary1;
2 • show tables;
3 • select * from medication;
4 • select * from medicines_equipments;
5 • select * from supply_transaction;
6 • show Index from OPD;
7 • select * from OPD;
8 • create index Idx_OPD_type on OPD(Case_Type);
9 • select * from OPD where Case_Type='Old';
10 • explain analyze select * from OPD where Case_Type='Old';
11 • drop index Idx_OPD_type on OPD;
12 • drop table if exists senior_specialist;
13 • create table senior_specialist as (SELECT * FROM doctor WHERE doctor.Experience > '20');

```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | EXPLAIN  
-> Index lookup on OPD using idx\_OPD\_type (Case\_Type='Old') (cost=15.6 rows=134) (actual time=0.323..0.411 rows=134 loops=1)

- 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:

```

186 • Pincode INT NOT NULL CHECK (Pincode >= 100000 AND Pincode <= 999999),
187 • Account_Number BIGINT NOT NULL CHECK (Account_Number > 0),
188 • Bank_Name VARCHAR(100) NOT NULL,
189 • IFSC_Code VARCHAR(20) NOT NULL,
190 • Branch VARCHAR(100) NOT NULL
191 • );
192 • select * from Supplier;
193 • create index idx_sup_Ban on Supplier(Bank_Name);
194 • select * from supplier where Bank_Name = 'XYZ Bank';
195 • explain analyze select * from supplier where Bank_Name = 'XYZ Bank';
196 • drop index idx_sup_Ban on Supplier;
197 • DROP TABLE IF EXISTS Supplier_Contact;
198 • CREATE TABLE Supplier_Contact (

```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | EXPLAIN  
-> Filter: (supplier.Bank\_Name = 'XYZ Bank') (cost=4.25 rows=4) (actual time=0.135..0.162 rows=16 loops=1)  
-> Table scan on supplier (cost=4.25 rows=40) (actual time=0.129..0.149 rows=40 loops=1)

### With indexing:

```

6 • Pincode INT NOT NULL CHECK (Pincode >= 100000 AND Pincode <= 999999),
7 • Account_Number BIGINT NOT NULL CHECK (Account_Number > 0),
8 • Bank_Name VARCHAR(100) NOT NULL,
9 • IFSC_Code VARCHAR(20) NOT NULL,
10 • Branch VARCHAR(100) NOT NULL
11 • );
12 • select * from Supplier;
13 • create index idx_sup_Ban on Supplier(Bank_Name);
14 • select * from supplier where Bank_Name = 'XYZ Bank';
15 • explain analyze select * from supplier where Bank_Name = 'XYZ Bank';
16 • drop index idx_sup_Ban on Supplier;
17 • DROP TABLE IF EXISTS Supplier_Contact;
18 • CREATE TABLE Supplier_Contact (

```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | EXPLAIN  
-> Index lookup on supplier (cost=2.35 rows=16) (actual time=0.29..0.302 rows=16 loops=1)

- 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:

```
257     Issue_Date DATE NOT NULL,
258     Expiry_Date DATE NOT NULL,
259     Wallet_Balance DECIMAL(10, 2) NOT NULL CHECK (Wallet_Balance > 0),
260     Reimbursement_Status VARCHAR(20) NOT NULL DEFAULT 'Pending'
261   );
262 •   create index ins_stat on Insurance(Reimbursement_Status);
263 •   select * from Insurance where Reimbursement_Status = 'Pending';
264 •   explain analyze select * from Insurance where Reimbursement_Status = 'Pending';
265 •   drop index ins_stat on Insurance;
266 •   select * from Insurance;
267 •   DROP TABLE IF EXISTS Purchase_Order;
268 •   CREATE TABLE Purchase_Order(
269     Bill_Number Varchar(50) PRIMARY KEY,
```

Result Grid | Filter Rows: Export: Wrap Cell Content: EXPLAIN  
-> Filter: (insurance.Reimbursement\_Status = 'Pending') (cost=30.2 rows=30) (actual time=0.163..0.341 rows=50 loops=1)  
-> Table scan on Insurance (cost=30.2 rows=300) (actual time=0.157..0.299 rows=300 loops=1)

### With indexing:

```
257     Issue_Date DATE NOT NULL,
258     Expiry_Date DATE NOT NULL,
259     Wallet_Balance DECIMAL(10, 2) NOT NULL CHECK (Wallet_Balance > 0),
260     Reimbursement_Status VARCHAR(20) NOT NULL DEFAULT 'Pending'
261   );
262 •   create index ins_stat on Insurance(Reimbursement_Status);
263 •   select * from Insurance where Reimbursement_Status = 'Pending';
264 •   explain analyze select * from Insurance where Reimbursement_Status = 'Pending';
265 •   drop index ins_stat on Insurance;
266 •   select * from Insurance;
267 •   DROP TABLE IF EXISTS Purchase_Order;
268 •   CREATE TABLE Purchase_Order(
269     Bill_Number Varchar(50) PRIMARY KEY,
```

Result Grid | Filter Rows: Export: Wrap Cell Content: EXPLAIN  
-> Index lookup on Insurance using ins\_stat (R...) (cost=5.75 rows=50) (actual time=0.165..0.178 rows=50 loops=1)

- 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\*   

```

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: 15

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 Danielshire	115216	Female	qmorales@e
19011	Bryan	John	Curtis	2019-11-01	719	Maxwell Terrace	55	East Deborahbury	960742	Female	sarahrodr
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
19020	Michael	Dwight	Warren	2009-09-29	77	Teresa Vale	91	New Michville	541458	Female	rhriemann

F\_patients 4 ×

Output:

Action Output

#	Time	Action	Message	Duration / Fetch
22	01:00:02	create table Others_patients as (SELECT * FROM patient WHERE patient.gender = 'Female')	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 = 'Female')	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

Result Grid | Filter Rows: Export: Wrap Cell Content: 15

## M\_patients:

MySQL Workbench

File Edit View Query Database Server Tools Scripting Help

Navigator:    Query 1    sql\_practice\_section1    SQL File 2\*    SQL File 5\*    SQL File 6\*    SQL File 7\*   

SCHEMAS

- 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

M\_patients 5 ×

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 = 'Female')	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:

```

MySQL Workbench - SQL Practice_1
File Edit View Query Database Server Tools Scripting Help
Navigator: Schemas
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
Object Info Session

Query 1 sql_practice_section1 SQL File 2* SQL File 5* SQL File 6* SQL File 7*
4 • select * from M_patients;
5
6 • create table Others_patients as (SELECT * FROM patient WHERE patient.gender = 'Other');
7 • select * from Others_patients;
8

Result Grid | Filter Rows: Export: Wrap Cell Content: 15
Patient_ID First_Name Middle_Name Last_Name DOB Street_Number Street_Name Apartment_Number City Pincode Gender Email
19002 Leslie Lori White 1945-01-16 337 Murray Walks 31 North Eddie 420127 Other zachary95@exar
19003 Gary Robyn Jackson 1957-05-02 293 Christopher Isle 29 North Leonard 889541 Other dana83@exar
19005 Anthony Jesse Jones 1969-10-27 417 Heidi Stravenue 93 North Shelsbyside 248281 Other joseph06@ex
19009 Steven Jill Anderson 2004-04-10 878 Amy Knolls 75 Skimmershire 742623 Other perezrian@e
19010 Maureen Zoe Ward 1964-09-06 564 Mason Expressway 21 Austinview 256079 Other juan71@exar
19013 David Laura Evans 2016-10-07 258 Clarke Green 10 North Donnchester 460199 Other deanhifi@exar

Others_patients 6 x
Output:
Action Output
# Time Action Message Duration / Fetch
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
27 01:03:41 select * from Others_patients LIMIT 0, 1000 295 row(s) returned 0.015 sec / 0.000 sec

```

## 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:

```

MySQL Workbench - SQL Practice_1
File Edit View Query Database Server Tools Scripting Help
Navigator: Schemas
Schemas
Filter objects
dispensary
Tables
doctor
doctor_availability
doctor_contact
hospital
hospital_contact
medical_test
medicine_equipment
opd
patient
patient_allergy
patient_contact
staff
staff_contact
Administration Schemas
Information
No object selected
Object Info Session

Query 1 sql_practice_section1 SQL File 2* SQL File 5* SQL File 6* SQL File 7*
11
12
13
14 • create table Senior_specialist as (SELECT * FROM doctor WHERE doctor.experience > '15');
15 • select * from Senior_specialist ORDER BY Experience;

Result Grid | Filter Rows: Export: Wrap Cell Content: 15
Doctor_ID First_Name Middle_Name Last_Name DOB Specialization Experience Street_Number Street_Name Apartment_Number City Pincode
1401 William Alexander Powell 1983-05-21 Orthopedics 17 94143 Berger Radial 678 West Sandy 54931C
1601 Brandon Jeffery Miller 1978-12-21 Neurology 18 9458 Bryan Ports 320 East Ashleymouth 567488
1503 Courtney Stephen Dean 1963-03-19 Oncology 29 957 Christopher Summit 9475 Joseburgh 580395
1402 Kathryn Anthony Manning 1965-04-01 Pediatrics 31 97637 David Row 954 Hopkinstad 434765
1502 Janet Jennifer Simmons 1990-05-22 Cardiology 36 961 Brown Underpass 5 Brownmouth 517981
1401 Indhuu Ieruv Clark 1977-10-11 Orthopedics 38 9883 Arthur Walk R4R9R Prnt Ravnndl 521561

Senior_specialist 9 x
Output:
Action Output
# Time Action Message Duration / Fetch
28 01:06:46 create table Senior_specialist as (SELECT * FROM doctor WHERE doctor.experienc... 7 row(s) affected Records: 7 Duplicates: 0 Warnings: 0 0.032 sec
29 01:06:51 select * from Senior_specialist LIMIT 0, 1000 7 row(s) returned 0.000 sec / 0.000 sec
30 01:07:50 select * from Senior_specialist ORDER BY Experience DESC LIMIT 0, 1000 7 row(s) returned 0.000 sec / 0.000 sec
31 01:07:56 select * from Senior_specialist ORDER BY Experience LIMIT 0, 1000 7 row(s) returned 0.000 sec / 0.000 sec

```

## Junior\_doctors:

The screenshot shows the MySQL Workbench interface with the following details:

- File Bar:** File, Edit, View, Query, Database, Server, Tools, Scripting, Help.
- Schemas Navigator:** Dispensary schema selected, showing tables: doctor, doctor\_availability, doctor\_contact, hospital, hospital\_contact, medical\_test, medicine\_equipment, opd, patient, patient\_allergy, patient\_contact, staff, staff\_contact.
- Query Editor:** SQL File 7 tab selected, containing the following SQL code:

```
14 •  create table Senior_specialist as (SELECT * FROM doctor WHERE doctor.experience > '15');
15 •  select * from Senior_specialist ORDER BY Experience;
16
17 •  create table Junior_Doctor as (SELECT * FROM doctor WHERE doctor.experience < '15');
18 •  select * from Junior_Doctor ORDER BY Experience;
```
- Result Grid:** Shows the results of the SELECT query on the Junior\_Doctor table.

Doctor_ID	First_Name	Middle_Name	Last_Name	DOB	Specialization	Experience	Street_Number	Street_Name	Apartment_Number	City	Pincode	Gender
1602	Margaret	Nicholas	Maddox	1973-07-12	Neurology	1	73401	Hurley Flat	10523	Mannfort	990149	Female
1702	Kathleen	Zachary	Graham	1989-07-08	Neurology	1	37921	Davis Mission	14256	Port Patrick	327640	Female
1501	Brinna	John	McKenzie	1965-08-14	Oncology	14	431	Mary Islands	4835	Port Spencer	307542	Male
- Action Output:** Shows the execution log for the queries.

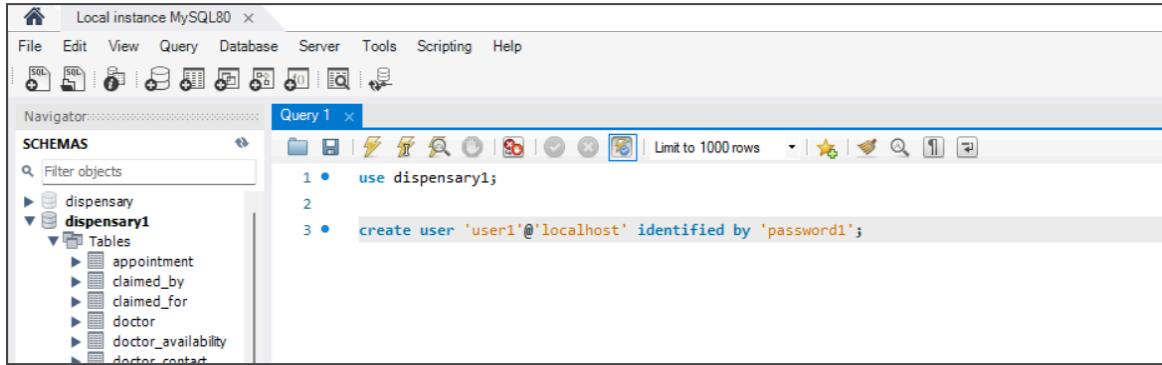
#	Time	Action	Message	Duration / Fetch
30	01:07:50	select * from Senior_specialist ORDER BY Experience DESC LIMIT 0, 1000	7row(s) returned	0.000 sec / 0.000 sec
31	01:07:56	select * from Senior_specialist ORDER BY Experience LIMIT 0, 1000	7row(s) returned	0.000 sec / 0.000 sec
32	01:09:50	create table Junior_Doctor as (SELECT * FROM doctor WHERE doctor.experienc...	3row(s) affected Records: 3 Duplicates: 0 Warnings: 0	0.047 sec
33	01:09:52	select * from Junior_Doctor ORDER BY Experience LIMIT 0, 1000	3row(s) returned	0.000 sec / 0.000 sec

## 3.2 Responsibility of G2:

### 3.2.1] SQL Queries

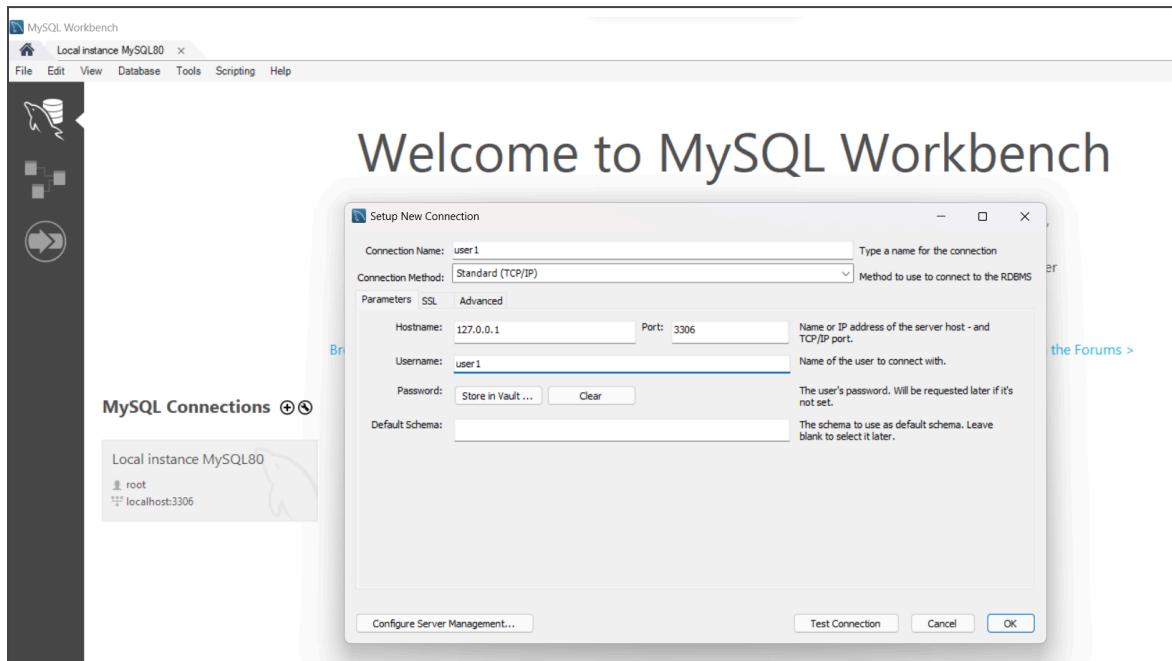
(i)

Creating user- user1 with password- password1.

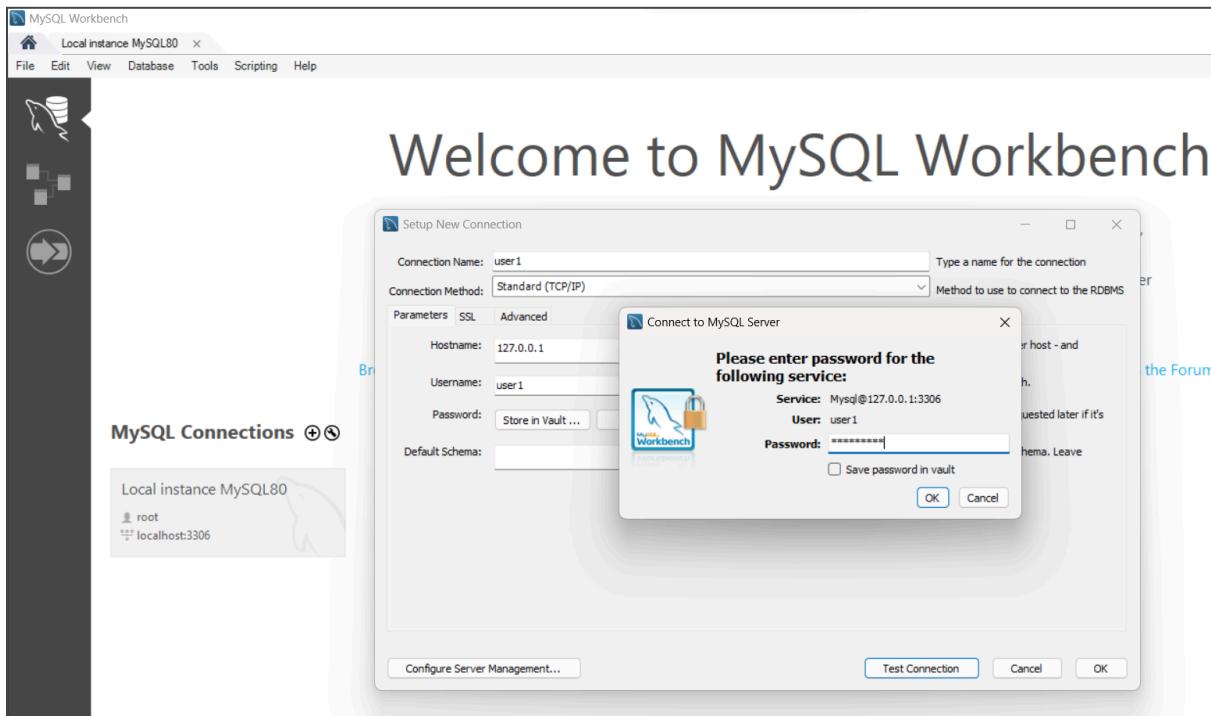


```
use dispensary1;
create user 'user1'@'localhost' identified by 'password1';
```

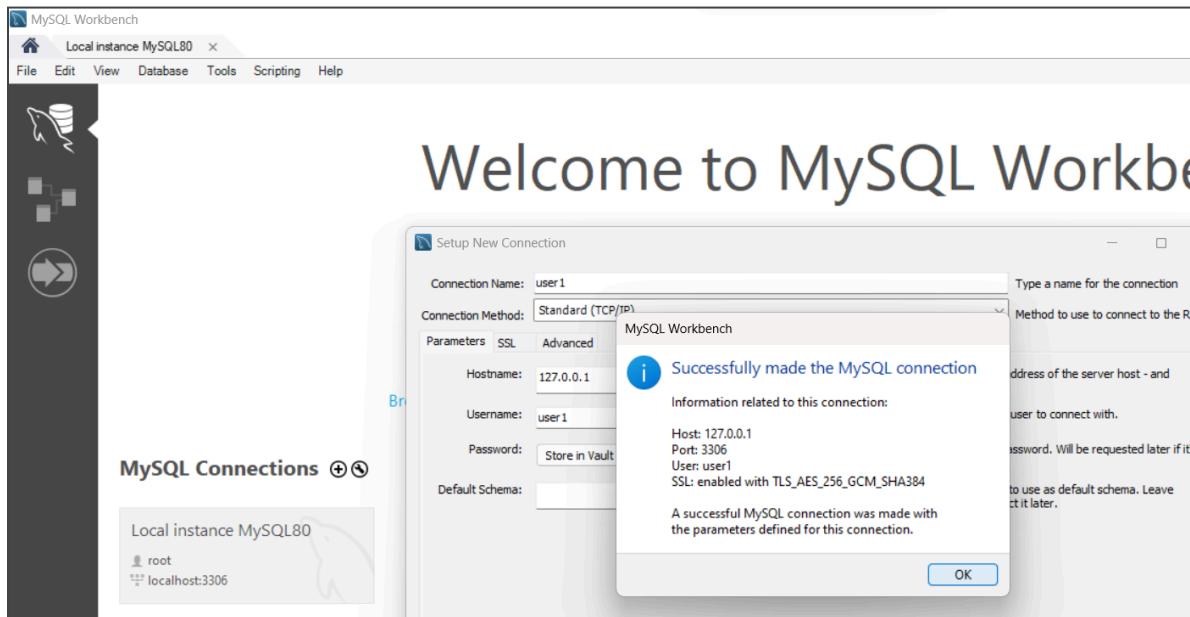
Setting up a new connection with user1.



Entering the password.

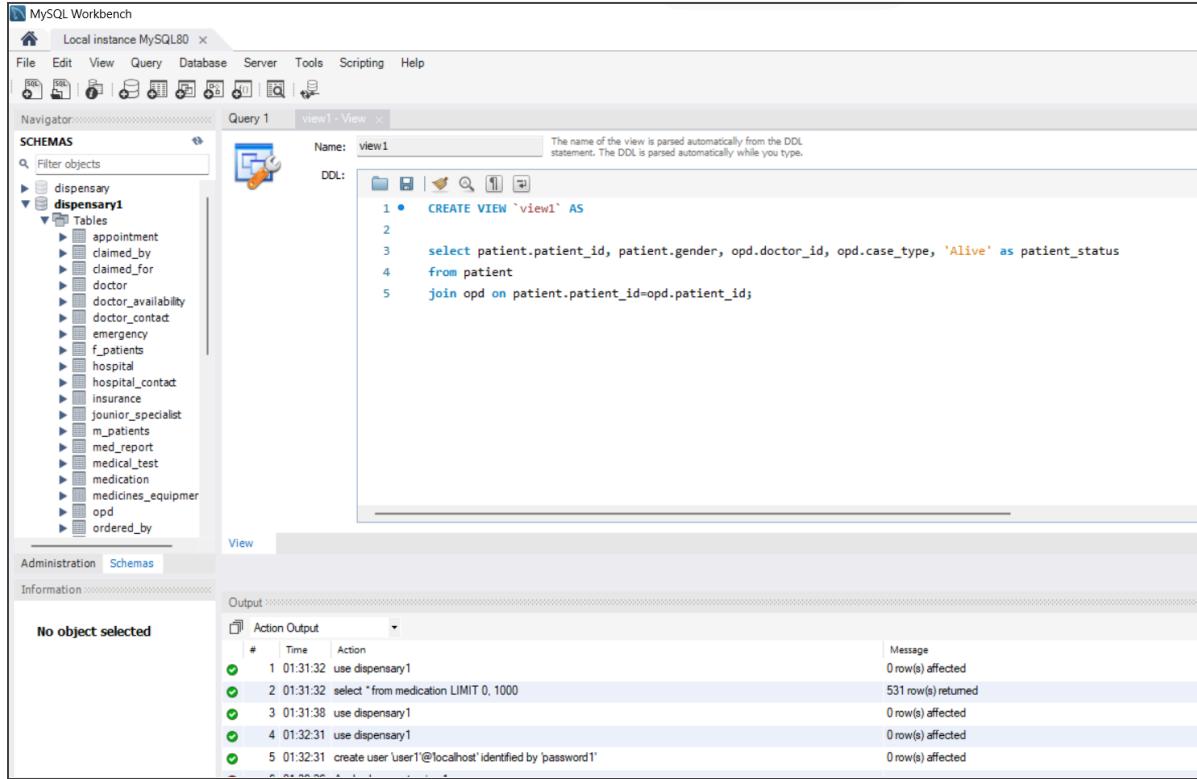


Connection of user1 successfully established.



(ii)

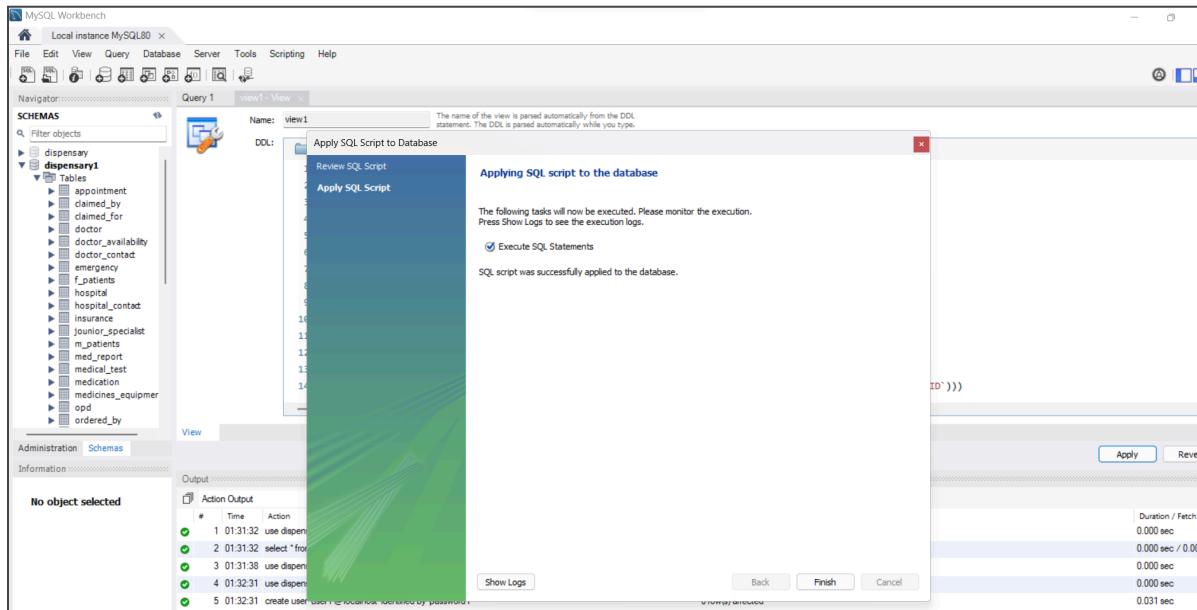
Creating a view- view1 with the Patient Table and OPD Table with patient\_id, gender column from the Patient Table and case\_type from the OPD Table. Patient\_id is the foreign key in the OPD Table. We have also created a user-defined column, patient\_status with user-defined data type.



```
CREATE VIEW `view1` AS
select patient.patient_id, patient.gender, opd.doctor_id, opd.case_type, 'Alive' as patient_status
from patient
join opd on patient.patient_id=opd.patient_id;
```

The screenshot shows the MySQL Workbench interface. In the left sidebar, under the 'Schemas' section, there is a tree view of tables in the 'dispensary1' schema. The 'Tables' node is expanded, showing various tables like appointment, claimed\_by, claimed\_for, doctor, doctor\_availability, doctor\_contact, emergency, f\_patients, hospital, hospital\_contact, insurance, jounior\_specialist, m\_patients, med\_report, medical\_test, medication, medicines\_equipmer, opd, and ordered\_by. The main query editor window is titled 'view1 - View' and contains the SQL DDL code for creating the view. Below the code, the 'Output' pane displays the execution log with several entries, indicating successful creation of the user and the view itself.

View1 created.



## Displaying view1.

The screenshot shows the MySQL Workbench interface with the following details:

- File Bar:** Local instance MySQL80, File, Edit, View, Query, Database, Server, Tools, Scripting, Help.
- Toolbar:** Includes icons for Home, Database, Navigator, Query, Schema, Tools, Help, and a search bar.
- Navigator:** Shows the schema 'dispensary1' with its tables: appointment, claimed\_by, claimed\_for, doctor, doctor\_availability, doctor\_contact, emergency, f\_patients, hospital, hospital\_contact, insurance, jounior\_specialist, m\_patients, med\_report, medical\_test, medication, medicines\_equipmer, opd, ordered\_by.
- Query Editor:** Title 'Query 1 - view1 - View'. The query is:

```

1 • use dispensary1;
2
3 • create user 'user1'@'localhost' identified by 'password1';
4
5 • select * from view1;
    
```
- Result Grid:** Title 'view1\_2'. The grid displays the following data:

patient_id	gender	doctor_id	case_type	patient_status
23024	Other	1403	New	Alive
19055	Female	1403	New	Alive
19011	Male	1503	New	Alive
21060	Male	1602	New	Alive
19027	Female	1702	New	Alive
19034	Female	1601	New	Alive
22013	Male	1502	New	Alive
19013	Other	1601	New	Alive
19006	Female	1702	New	Alive
21026	Male	1602	New	Alive
21030	Male	1402	New	Alive
23012	Female	1601	New	Alive
22022	Other	1702	New	Alive
22016	Female	1401	New	Alive
22018	Other	1402	New	Alive
22051	Other	1701	New	Alive
19039	Female	1601	New	Alive
21038	Male	1702	New	Alive
19023	Female	1602	New	Alive
23007	Female	1601	New	Alive
20025	Male	1402	New	Alive
23010	Female	1702	New	Alive
20032	Other	1602	New	Alive
23020	Female	1503	New	Alive
22018	Other	1403	New	Alive
21038	Male	1701	Old	Alive
22057	Other	1402	New	Alive

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\_availability, with a user-defined data type having two values, “Night” and “Day”.

The screenshot shows the MySQL Workbench interface with the following details:

- File Bar:** Local instance MySQL80, File, Edit, View, Query, Database, Server, Tools, Scripting, Help.
- Toolbar:** Includes icons for Home, Database, Navigator, Query, Schema, Tools, Help, and a search bar.
- Navigator:** Shows the schema 'dispensary1' with its tables: appointment, claimed\_by, claimed\_for, doctor, doctor\_availability, doctor\_contact, emergency, f\_patients, hospital, hospital\_contact, insurance, jounior\_specialist, m\_patients, med\_report.
- Query Editor:** Title 'new\_view - View'. The query is:

```

Name: new_view
The name of the view is parsed automatically from the DDL statement. The DDL is parsed automatically while you type.

CREATE VIEW `view2` AS
SELECT emergency.license_number, emergency.disease, hospital.name,
CASE
WHEN ROW_NUMBER() OVER (ORDER BY emergency.license_number) <= (SELECT COUNT(*) FROM emergency) / 2 THEN 'Night'
ELSE 'Day'
END AS hospital_availability
FROM emergency
JOIN hospital ON emergency.license_number=hospital.license_number;
    
```

## Displaying view2.

The screenshot shows the MySQL Workbench interface with a query editor and a results grid. The query editor contains the following SQL code:

```
1 • use dispensary1;
2
3 • create user 'user1'@'localhost' identified by 'password1';
4
5 • select * from view1;
6 • select * from view2;
```

The results grid displays data from the view2 table, which is a union of view1 and view2. The columns are license\_number, disease, name, and hospital\_availability. The data includes various medical conditions and their associated hospitals and availability times.

license_number	disease	name	hospital_availability
196018	Atrial Septal Defect	UCLA Medical Center	Night
196018	Asthma	UCLA Medical Center	Night
196018	Sepsis	UCLA Medical Center	Night
274690	Pneumonia	Stanford Health Care	Night
274690	Fractures	Stanford Health Care	Night
274690	Anaphylaxis	Stanford Health Care	Night
274690	Meningitis	Stanford Health Care	Night
366789	Meningitis	Cleveland Clinic	Night
366789	Anaphylaxis	Cleveland Clinic	Night
366789	Seizures (Epilepsy)	Cleveland Clinic	Night
366789	Dehydration	Cleveland Clinic	Night
366789	Severe Allergic Reaction	Cleveland Clinic	Night
366789	Kawasaki Disease	Cleveland Clinic	Night
511309	Heatstroke	Memorial Hospital	Night
511309	Concussion	Memorial Hospital	Night
511309	Fractures	Memorial Hospital	Night
511309	Sepsis	Memorial Hospital	Night
511309	Heatstroke	Memorial Hospital	Night
581630	Dehydration	Johns Hopkins Hospital	Night
581630	Kawasaki Disease	Johns Hopkins Hospital	Day
581630	Diabetic Ketoacidosis	Johns Hopkins Hospital	Day
585857	Concussion	Massachusetts Gene...	Day
657454	Diabetic Ketoacidosis	Brigham and Women...	Day
657454	Appendicitis	Brigham and Women...	Day
657454	Acute Bronchitis	Brigham and Women...	Day
657454	Anaphylaxis	Brigham and Women...	Day
657454	Appendicitis	Brigham and Women...	Day

(iii)

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

The screenshot shows the MySQL Workbench interface. In the left sidebar, under the 'SCHEMAS' section, the 'dispensary1' schema is selected. The 'Tables' section lists various tables such as appointment, claimed\_by, claimed\_for, doctor, etc. In the main query editor window, the following SQL commands are run:

```
1 • use dispensary1;
2
3 • create user 'user1'@'localhost' identified by 'password1';
4
5 • grant select,update,delete on dispensary1.patient to 'user1'@'localhost';
6
7 • show grants for 'user1'@'localhost';
```

The results grid shows the grants for user1@localhost, including:

- GRANT USAGE ON \*.\* TO 'user1'@'localhost'
- GRANT SELECT, UPDATE, DELETE ON 'dispens...'

(iv)

Granting SELECT permission on view1 to user1.

The screenshot shows the MySQL Workbench interface. In the left sidebar, under the 'SCHEMAS' section, the 'dispensary1' schema is selected. The 'Tables' section lists various tables. In the main query editor window, the following SQL commands are run:

```
1 • use dispensary1;
2
3 • create user 'user1'@'localhost' identified by 'password1';
4
5 • grant select on view1 to 'user1'@'localhost';
6
7 • show grants for 'user1'@'localhost';
```

The results grid shows the grants for user1@localhost, including:

- GRANT USAGE ON \*.\* TO 'user1'@'localhost'
- GRANT SELECT, UPDATE, DELETE ON 'dispens...'
- GRANT SELECT ON 'dispensary1'.`view1` TO ...

A tooltip highlights the last line of the results grid: 'GRANT SELECT ON `dispensary1`.`view1` TO `user1`@`localhost`'.

(v) Logging in as user1.

### Table1 operations-

SELECT operation on the Patient Table.

The screenshot shows the MySQL Workbench interface with the following details:

- File Bar:** Local Instance MySQL80, user1
- Navigator:** Schemas (dispensary1 selected), Tables, Views, Stored Procedures, Functions
- Query Editor:**

```
1 • use dispensary1;
2
3 • select * from patient;
```
- Result Grid:** Displays 14 rows of patient data with columns: Patient\_ID, First\_Name, Middle\_Name, Last\_Name, DOB, Street\_Number, Street\_Name, Apartment\_Number, City, Pincode, Gender, Email.

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	oai@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 Heathfurt	536915	Female	huffman@example.net
19006	Leslie	Timothy	Parker	1968-01-10	315	Obrien Squares	5	South Mitchell	647290	Female	pday@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	gonzalezhon@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	Stevensburg	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.

The screenshot shows the MySQL Workbench interface with the following details:

- File Bar:** Local Instance MySQL80, user1
- Navigator:** Schemas (dispensary1 selected), Tables, Views, Stored Procedures, Functions
- Query Editor:**

```
1 • use dispensary1;
2
3 • update patient set apartment_number=100*apartment_number;
```
- Result Grid:** Displays 14 rows of patient data with columns: Patient\_ID, First\_Name, Middle\_Name, Last\_Name, DOB, Street\_Number, Street\_Name, Apartment\_Number, City, Pincode, Gender, Email.

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	oai@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 Heathfurt	536915	Female	huffman@example.net
19006	Leslie	Timothy	Parker	1968-01-10	315	Obrien Squares	5	South Mitchell	647290	Female	pday@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	gonzalezhon@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	Stevensburg	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.

The screenshot shows the MySQL Workbench interface. In the Navigator pane, the schema 'dispensary1' is selected. In the Query Editor (Query 1), the following SQL code is run:

```
1 • use dispensary1;
2
3 • update patient set apartment_number=100*apartment_number;
4
5 • select * from patient;
```

The Result Grid displays the updated data for the 'patient' table. The 'Apartment\_Number' column has been multiplied by 100. The first few rows of the result are:

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	2100	Zacharyberg	366855	Other	xcarr@example.org
19002	Brandon	Maria	Lester	2000-11-13	555	Fuller Parkway	1500	Michelmouth	681901	Male	oai@example.com
19003	Howard	Maria	Bernard	1955-10-14	741	Hardin Track	1400	Claymouth	607898	Other	michele76@example.org
19004	Gregory	Bianca	Cannon	1962-11-01	881	Jones Loaf	500	East CassandraTown	110947	Male	brandonhester@example.org

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.

The screenshot shows the MySQL Workbench interface. In the Navigator pane, the schema 'dispensary1' is selected. In the Query Editor (Query 1), the following SQL code is run:

```
1 • use dispensary1;
2
3 • delete from doctor_contact
4   where doctor_id=1503;
5
6 • select * from doctor_contact;
7
```

The Result Grid displays the data from the 'doctor\_contact' table. The row with doctor\_id 1503 is highlighted. The data is:

Doctor_ID	Phone_Number
1401	9690601948
1403	9496646039
1501	9117878756
1501	9398199945
1502	9318509333
1503	9087382848
1503	9891191283
1601	9658537411
1601	9719632722
1602	9541482061
1602	9887838121
1701	9379165317
1701	9607407513

After deleting records with 1503 doctor\_id.

The screenshot shows the MySQL Workbench interface. The query editor contains the following SQL code:

```
1 • use dispensary1;
2
3 • delete from doctor_contact
4   where doctor_id=1503;
5
6 • select * from doctor_contact;
7
```

The result grid displays the following data:

Doctor_ID	Phone_Number
1401	9690601948
1403	9496646039
1501	9117878756
1501	9398199945
1502	9318509333
1601	9658537411
1601	9719632722
1602	9541482061
1602	9887838121
1701	9379165317
1701	9607407513
1702	9167351405
1702	9619366826

## View1 operations-

SELECT operation on the view1.

The screenshot shows the MySQL Workbench interface. The query editor contains the following SQL code:

```
1 • use dispensary1;
2
3 • select * from view1;
4
```

The result grid displays the following data:

patient_id	gender	doctor_id	case_type	patient_status
23024	Other	1403	New	Alive
19055	Female	1403	New	Alive
19011	Male	1503	New	Alive
21060	Male	1602	New	Alive
19027	Female	1702	New	Alive
19034	Female	1601	New	Alive
22013	Male	1502	New	Alive
19013	Other	1601	New	Alive
19006	Female	1702	New	Alive
21026	Male	1602	New	Alive
21030	Male	1402	New	Alive
23012	Female	1601	New	Alive
22022	Other	1702	New	Alive

## UPDATE operation on view1.

The screenshot shows the MySQL Workbench interface. In the left sidebar, under the 'SCHEMAS' section, 'dispensary1' is selected. In the main query editor window, the following SQL code is entered:

```
1 • use dispensary1;
2
3 • update view1 set case_type='old';
4
```

In the 'Output' pane at the bottom, there is a table titled 'Action Output' showing the history of actions:

#	Time	Action	Message	Duration / Fetch
1	19 02:20:46	use dispensary1;	500 row(s) returned	0.000 sec / 0.000 sec
2	19 02:21:54	update view1 set case_type='old';	Error Code: 1142. UPDATE command denied to user 'user1'@'localhost' for table 'view1'	0.000 sec
3	21 02:22:43	update view1 set case_type='old';	Error Code: 1142. UPDATE command denied to user 'user1'@'localhost' for table 'view1'	0.000 sec

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.

The screenshot shows the MySQL Workbench interface. In the left sidebar, under the 'SCHEMAS' section, 'dispensary1' is selected. In the main query editor window, the following SQL code is entered:

```
1 • use dispensary1;
2
3 • delete from view1;
4
```

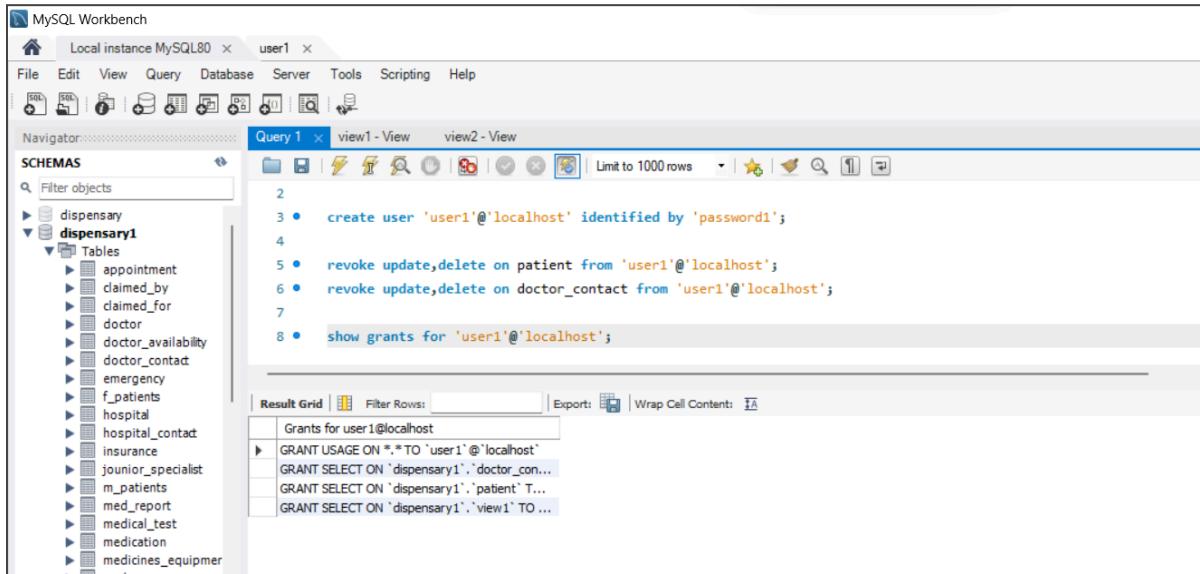
In the 'Output' pane at the bottom, there is a table titled 'Action Output' showing the history of actions:

#	Time	Action	Message	Duration / Fetch
1	21 02:22:43	use dispensary1;	Error Code: 1142. UPDATE command denied to user 'user1'@'localhost' for table 'view1'	0.000 sec
2	22 02:23:15	delete from view1	Error Code: 1142. DELETE command denied to user 'user1'@'localhost' for table 'view1'	0.016 sec
3	23 02:23:17	delete from view1	Error Code: 1142. DELETE command denied to user 'user1'@'localhost' for table 'view1'	0.000 sec

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.



The screenshot shows the MySQL Workbench interface with the following details:

- File**: Local instance MySQL80, user1
- Schemas**: dispensary, dispensary1 (selected), hospital, hospital\_contact, insurance, jounior\_specialist, m\_patients, med\_report, medical\_test, medication, medicines\_equipmer
- Tables**: appointment, claimed\_by, claimed\_for, doctor, doctor\_availability, doctor\_contact, emergency, f\_patients, hospital, hospital\_contact, insurance, jounior\_specialist, m\_patients, med\_report, medical\_test, medication, medicines\_equipmer
- Query 1**:

```
2
3 •  create user 'user1'@'localhost' identified by 'password1';
4
5 •  revoke update,delete on patient from 'user1'@'localhost';
6 •  revoke update,delete on doctor_contact from 'user1'@'localhost';
7
8 •  show grants for 'user1'@'localhost';
```
- Result Grid**: Grants for user1@localhost

GRANT USAGE ON *.* TO 'user1'@'localhost'
GRANT SELECT ON `dispensary1`.`doctor_con...
GRANT SELECT ON `dispensary1`.`patient` T...
GRANT SELECT ON `dispensary1`.`view1` TO ...

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 screenshot shows the MySQL Workbench interface. The left sidebar displays the Navigator with the schema 'dispensary1' selected, showing tables, views, stored procedures, and functions. The main area is titled 'Query 1' and contains the following SQL code:

```
1 • use dispensary1;
2
3 • select * from patient;
```

The results are displayed in a grid titled 'Result Grid'. The columns are: Patient\_ID, First\_Name, Middle\_Name, Last\_Name, DOB, Street\_Number, Street\_Name, Apartment\_Number, City, Pincode, Gender, and Email. The data consists of 19 rows of patient information.

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	oai@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	O'Brien Squares	5	South Mitchell	647290	Female	pday@example.com
19007	Jason	Kathryn	Anderson	1975-05-18	421	Gina Mills	42	North Nathanielshire	621234	Male	romeroleith@example.org
19008	Carlos	Lori	Herman	1980-08-02	553	Tyler Forges	92	Ramoberg	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 Junburgh	951215	Other	perezanthony@example.net
19014	Holly	Brittany	Burnett	1956-10-08	151	Peterson Ports	33	North Devin	823519	Other	ecarpenter@example.com
19015	Christopher	Wendy	Gibbs	1968-03-24	204	Moses Trail	99	South Johnny	602634	Other	carolmorgan@example.org
19016	Amy	Sarah	Harris	1991-12-17	374	Baker Rue	88	Smithtown	505964	Male	christopherjohnson@example.c...
19017	Sharon	Aaron	Oliver	1978-06-17	514	Earl Highway	77	Sarisbury	238377	Female	randallonell@example.com

The SELECT operation works.

UPDATE operation on the Patient Table.

The screenshot shows the MySQL Workbench interface. The left sidebar displays the Navigator with the schema 'dispensary1' selected. The main area is titled 'Query 1' and contains the following SQL code:

```
1 • use dispensary1;
2
3 • update patient set apartment_number=200*apartment_number;
```

The results are displayed in a grid titled 'Output' under the 'Action Output' tab. The log shows three actions:

#	Time	Action	Message	Duration / Fetch
26	02:27:24	select * from patient LIMIT 0, 1000	300 row(s) returned	0.000 sec / 0.000 sec
27	02:28:21	update patient set apartment_number=200*apartment_number	Error Code: 1142: UPDATE command denied to user 'user1'@'localhost' for table 'patient'	0.000 sec
28	02:28:23	update patient set apartment_number=200*apartment_number	Error Code: 1142: UPDATE command denied to user 'user1'@'localhost' for table 'patient'	0.000 sec

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.

The screenshot shows the MySQL Workbench interface. In the top-left corner, it says "Local instance MySQL80" and "user1". The "Query" tab is active, displaying the following SQL code:

```
1 • use dispensary1;
2
3 • delete from patient;
4 • delete from doctor_contact;
```

In the bottom-right pane, there is an "Output" section titled "Action Output" which lists three log entries:

#	Time	Action	Message	Duration / Fetch
28	02:28:23	update patient set apartment_number=200'apartment_number	Error Code: 1142. UPDATE command denied to user 'user1'@'localhost' for table 'patient'	0.000 sec
29	02:29:17	delete from patient	Error Code: 1142. DELETE command denied to user 'user1'@'localhost' for table 'patient'	0.000 sec
30	02:29:24	delete from doctor_contact	Error Code: 1142. DELETE command denied to user 'user1'@'localhost' for table 'doctor_contact'	0.000 sec

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 screenshot shows the MySQL Workbench interface. In the top-left corner, it says "Local instance MySQL80" and "user1". The "Query" tab is active, displaying the following SQL code:

```
1 • use dispensary1;
2
3 • select * from view1;
```

In the bottom-right pane, there is a "Result Grid" showing the data from the view:

patient_id	gender	doctor_id	case_type	patient_status
23024	Other	1403	New	Alive
19055	Female	1403	New	Alive
19011	Male	1503	New	Alive
21060	Male	1602	New	Alive
19027	Female	1702	New	Alive
19034	Female	1601	New	Alive
22013	Male	1502	New	Alive
19013	Other	1601	New	Alive
19006	Female	1702	New	Alive
21026	Male	1602	New	Alive
21030	Male	1402	New	Alive
23012	Female	1601	New	Alive
22022	Other	1702	New	Alive
22016	Female	1401	New	Alive
22018	Other	1402	New	Alive
22051	Other	1701	New	Alive
19039	Female	1601	New	Alive

The SELECT operation works.

## UPDATE operation on view1.

The screenshot shows the MySQL Workbench interface. In the top-left corner, it says "Local instance MySQL80" and "user1". The main area has a "Query 1" tab open with the following SQL code:

```
1 • use dispensary1;
2
3 • update view1 set case_type='old';
```

In the bottom right corner of the query window, there is an error message: "Error Code: 1142. UPDATE command denied to user 'user1'@'localhost' for table 'view1'".

The "Output" section at the bottom shows the following log entries:

#	Time	Action	Message	Duration / Fetch
31	02:29:50	select * from view1 LIMIT 0, 1000	500 row(s) returned	0.000 sec / 0.000 sec
32	02:30:27	update view1 set case_type='old'	Error Code: 1142. UPDATE command denied to user 'user1'@'localhost' for table 'view1'	0.000 sec
33	02:30:28	update view1 set case_type='old'	Error Code: 1142. UPDATE command denied to user 'user1'@'localhost' for table 'view1'	0.000 sec

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.

The screenshot shows the MySQL Workbench interface. In the top-left corner, it says "Local instance MySQL80" and "user1". The main area has a "Query 1" tab open with the following SQL code:

```
1 • use dispensary1;
2
3 • delete from view1;
```

In the bottom right corner of the query window, there is an error message: "Error Code: 1142. UPDATE command denied to user 'user1'@'localhost' for table 'view1'".

The "Output" section at the bottom shows the following log entries:

#	Time	Action	Message	Duration / Fetch
33	02:30:28	update view1 set case_type='old'	Error Code: 1142. UPDATE command denied to user 'user1'@'localhost' for table 'view1'	0.000 sec
34	02:30:50	delete from view1	Error Code: 1142. DELETE command denied to user 'user1'@'localhost' for table 'view1'	0.000 sec
35	02:30:51	delete from view1	Error Code: 1142. DELETE command denied to user 'user1'@'localhost' for table 'view1'	0.000 sec

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:

The screenshot shows the MySQL Workbench interface. The left sidebar displays the schema tree for 'dispensary1' with tables like college, dispensary, and popop. The central pane shows a SQL editor with the following code:

```
1 • use dispensary;
2 • show tables;
3 • select * from doctor;
4 • select * from prescribes;
5
6 • UPDATE prescribes
7   SET doctor_id = 1000
8   WHERE doctor_id = 1403;
9
```

The results grid shows the 'prescribes' table with the following data:

Prescription_ID	Doctor_ID
P0258	1403
P0263	1403
P0264	1403
P0270	1403
P0272	1403
P0313	1403

The output pane shows the execution log and an error message at the bottom:

```
Action Output
# Time Action
5 23:38:17 select * from f_patients LIMIT 0, 1000
Message: 104 row(s) returned
Duration / Fetch: 0.000 sec / 0.000 sec
6 23:45:49 use dispensary1
Message: 0 row(s) affected
Duration / Fetch: 0.000 sec
7 23:45:49 show tables
Message: 35 row(s) returned
Duration / Fetch: 0.000 sec / 0.000 sec
8 23:45:49 select * from doctor LIMIT 0, 1000
Message: 10 row(s) returned
Duration / Fetch: 0.000 sec / 0.000 sec
9 23:45:49 select * from prescribes LIMIT 0, 1000
Message: 500 row(s) returned
Duration / Fetch: 0.000 sec / 0.000 sec
10 23:45:49 UPDATE prescribes SET doctor_id = 1000 WHERE doctor_id = 1403
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`))
```

##### 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:

The screenshot shows the MySQL Workbench interface. In the SQL Editor pane, a script is run against the 'dispensary1' database:

```
1 • use dispensary1;
2 • show tables;
3 • select * from doctor;
4 • select * from prescribes;
5
6 • DELETE FROM Doctor
7 WHERE doctor_id = 1403;
8
9
```

The Result Grid pane displays the contents of the 'prescribes' table:

Prescription_ID	Doctor_ID
P0017	1401
P0028	1401
P0044	1401
P0045	1401
P0049	1401
P0051	1401
.....	1401
Result 11	doctor 12 prescribes 13

The Output pane shows the execution log:

#	Time	Action	Message	Duration / Fetch
1	23:56:12	use dispensary1	0 row(s) affected	0.000 sec
2	23:56:12	show tables	35 row(s) returned	0.016 sec / 0.000 sec
3	23:56:12	select * from doctor LIMIT 0, 1000	10 row(s) returned	0.000 sec / 0.000 sec
4	23:56:12	select * from prescribes LIMIT 0, 1000	500 row(s) returned	0.000 sec / 0.000 sec
5	23:56:12	DELETE FROM Doctor WHERE doctor_id = 1403	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`))	0.015 sec

### 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.

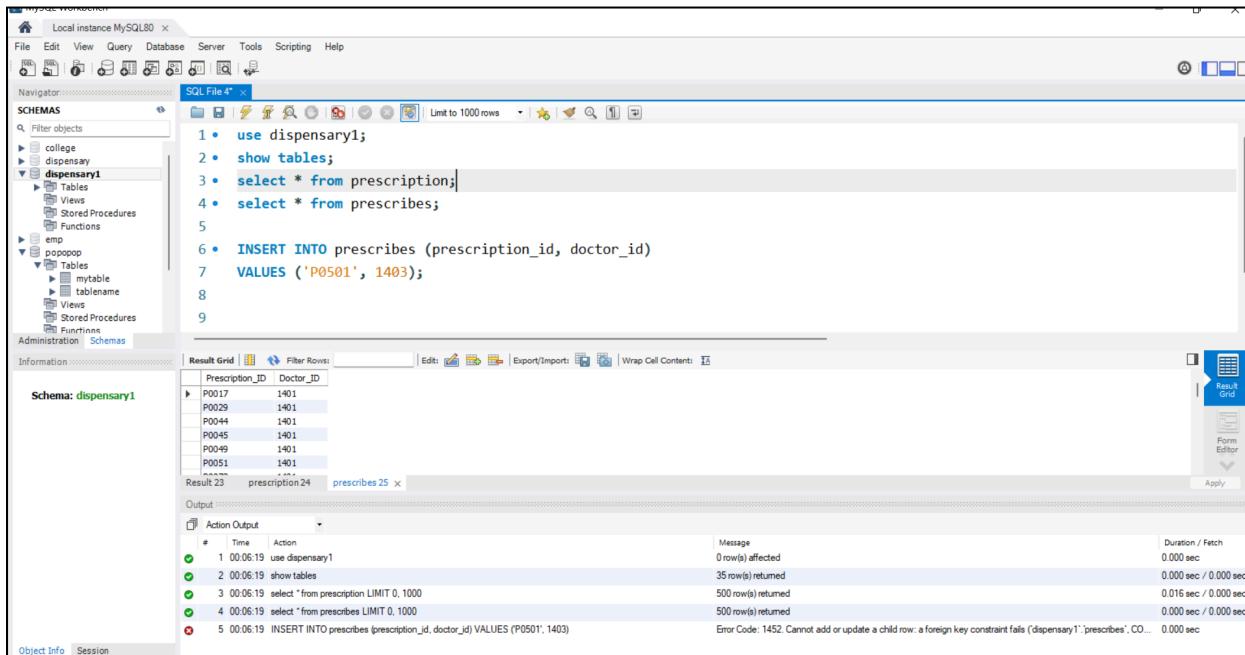
When we attempt to delete a row from the doctor table where doctor\_id is 1403, the foreign key constraint prevents it because there are corresponding records in the prescribed table that reference this doctor\_id. Deleting the row in the doctor table would leave orphaned references in the prescribed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



The screenshot shows the MySQL Workbench interface. In the top-left pane, the 'Schemas' tree shows the 'dispensary1' schema selected. The 'Tables' node under 'dispensary1' has several tables listed: college, dispensary, prescriptions, prescribes, mytable, and tablename. In the main query editor window, the following SQL code is entered:

```
1 • use dispensary1;
2 • show tables;
3 • select * from prescription;
4 • select * from prescribes;
5
6 • INSERT INTO prescribes (prescription_id, doctor_id)
7   VALUES ('P0501', 1403);
8
9
```

The 'Result Grid' below the editor shows the contents of the 'prescription' table:

Prescription_ID	Doctor_ID
P0017	1401
P0029	1401
P0044	1401
P0045	1401
P0049	1401
P0051	1401
...	

The 'Output' pane at the bottom displays the execution log:

#	Time	Action	Message	Duration / Fetch
1	00:06:19	use dispensary1	0 row(s) affected	0.000 sec
2	00:06:19	show tables	35 row(s) returned	0.000 sec / 0.000 sec
3	00:06:19	select * from prescription LIMIT 0, 1000	500 row(s) returned	0.016 sec / 0.000 sec
4	00:06:19	select * from prescribes LIMIT 0, 1000	500 row(s) returned	0.000 sec / 0.000 sec
5	00:06:19	INSERT INTO prescribes (prescription_id, doctor_id) VALUES ('P0501', 1403)	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`))	0.000 sec

#### 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 'prescribes' 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 'prescribes' 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:

The screenshot shows the MySQL Workbench interface. In the SQL Editor pane, the following SQL code is run:

```
1 • use dispensary1;
2 • show tables;
3 • select * from doctor;
4 • select * from prescribes;
5
6 • UPDATE Doctor
7   SET doctor_id = 1000
8   where doctor_id = 1403
9
```

In the Result Grid pane, the output of the SELECT statements is shown:

Prescription_ID	Doctor_ID
P0017	1401
P0029	1401
P0044	1401
P0045	1401
P0049	1401
P0051	1401

In the Action Output pane, the log of actions is displayed:

#	Time	Action	Message	Duration / Fetch
1	00:11:48	use dispensary1	0 row(s) affected	0.000 sec
2	00:11:48	show tables	35 row(s) returned	0.000 sec / 0.000 sec
3	00:11:48	select * from doctor LIMIT 0, 1000	10 row(s) returned	0.000 sec / 0.000 sec
4	00:11:48	select * from prescribes LIMIT 0, 1000	500 row(s) returned	0.000 sec / 0.000 sec
5	00:11:48	UPDATE Doctor SET doctor_id = 1000 where doctor_id = 1403	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`))	0.000 sec

### 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:**

```
1 •  use dispensary1;
2 •  show tables;
3 •  set foreign_key_checks = 0;
4 •  select * from doctor;
5 •  select * from prescribes;
6 •  UPDATE Doctor
7    SET doctor_id = 1000
8  where doctor_id = 1403;
9 •  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	X									

**No error message!**

Action	Output
#	Time Action
2	14:25:59 show tables
3	14:25:59 set foreign_key_checks = 0
4	14:25:59 select * from doctor LIMIT 0, 1000
5	14:25:59 select * from prescribes LIMIT 0, 1000
6	14:25:59 UPDATE Doctor SET doctor_id = 1000 where doctor_id = 1403
7	14:25:59 select * from doctor LIMIT 0, 1000

### **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:

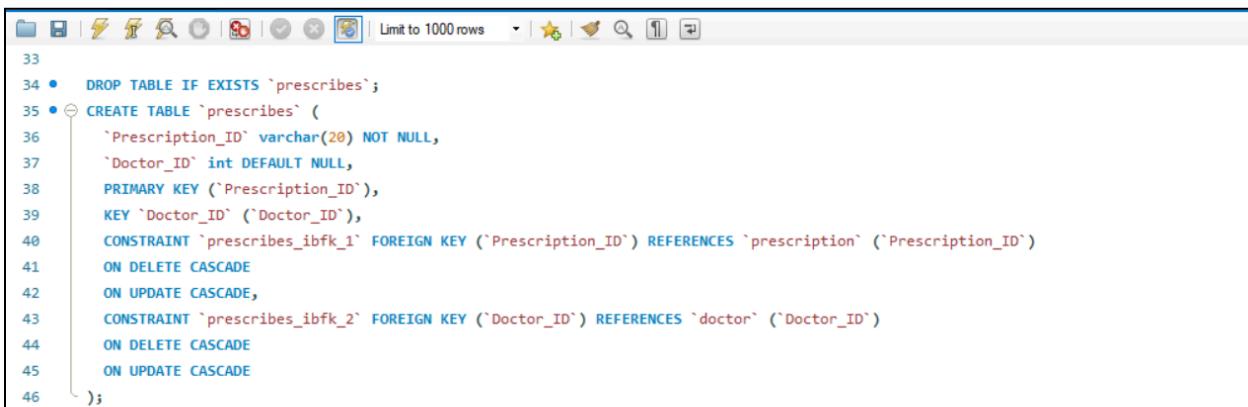
- ON DELETE CASCADE
- ON UPDATE CASCADE

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" 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.

The screenshot shows the SQL Server Management Studio interface. In the top pane, two queries are run:

```
SQL File 4" SQL File 3"
49
50
51 • select * from prescribes;
52 • SELECT COUNT(*) AS total_entries FROM prescribes;
53
54
55
```

The bottom pane displays the results of the second query in a grid:

Prescription_ID	Doctor_ID
P0442	1402
P0443	1402
P0444	1402
P0445	1402
P0446	1402
P0447	1402
P0448	1402
P0449	1402
P0450	1402
P0002	1403
P0007	1403
P0027	1403
P0033	1403
P0040	1403
P0052	1403
P0062	1403
P0076	1403
P0090	1403

Output window:

Action	Time	Action	Message	Duration / Fetch
202	18:12:45	select * from prescribes LIMIT 0, 1000	500 row(s) returned	0.000 sec / 0.000 sec
203	18:12:45	SELECT COUNT() AS total_entries FROM prescribes LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec

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.

The screenshot shows the SQL Server Management Studio interface. In the top pane, a DELETE statement is run on the doctor table:

```
SQL File 4" SQL File 3"
58
59
60
61 • select * from doctor;
62 • DELETE FROM doctor
63 WHERE doctor_id = 1403;
64 • select * from doctor;
```

The bottom pane displays the results of the final query in a grid:

Doctor_ID	First_Name	Middle_Name	Last_Name	DOB	Gender	Email	Specialization	Experience	Street_Number	Street_Name	Apartment_Number	City	Pincode
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	Brooke	John	McKenzie	1965-08-14	Male	brianna.mckenzie@gmail.com	Oncology	14	431	Mary Islands	4835	Port Spencer	307542
1502	Janet	Jenifer	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	Jeffery	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	Harley 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
*	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL

Output window:

Action	Time	Action	Message	Duration / Fetch
9	18:21:18	DELETE FROM doctor WHERE doctor_id = 1403	1 row(s) affected	0.000 sec
10	18:21:18	select * from doctor LIMIT 0, 1000	9 row(s) returned	0.000 sec / 0.000 sec

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’.

SQL File 4\* SQL File 3\* | Limit to 1000 rows | Filter Cell Content:  | Wrap Cell Content:

```
68
69
70 •   select * from prescribes;
71 •   SELECT COUNT(*) AS total_entries FROM prescribes;
72
73
```

Result Grid |  Filter Rows:  | Edit:    | Export/Import:   | Wrap Cell Content:

Prescription_ID	Doctor_ID
P0429	1402
P0431	1402
P0441	1402
P0442	1402
P0443	1402
P0461	1402
P0467	1402
P0471	1402
P0479	1402
P0491	1402
P0048	1501
P0063	1501
P0065	1501
P0074	1501
P0091	1501
P0093	1501
P0096	1501

prescribes 15 x Result 16

Output ::

#	Time	Action	Message	Duration / Fetch
11	18:31:18	select * from prescribes LIMIT 0, 1000	444 row(s) returned	0.000 sec / 0.000 sec
12	18:31:18	SELECT COUNT(*) AS total_entries FROM prescribes LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec

Total entries also get reduced to 444.

SQL File 4\* SQL File 3\* | Limit to 1000 rows | Filter Cell Content:  | Wrap Cell Content:

```
68
69
70 •   select * from prescribes;
71 •   SELECT COUNT(*) AS total_entries FROM prescribes;
72
73
```

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

total_entries
444

## Updating Referenced Primary Key Value

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

The screenshot shows a SQL interface with two tabs at the top: "SQL File 4\*" and "SQL File 3\*". The "SQL File 4\*" tab is active, displaying the following SQL code:

```
59  
60  
61  
62  
63  
64 • select * from prescribes;  
65 • SELECT COUNT(*) AS total_entries FROM prescribes;  
66  
67
```

Below the code is a "Result Grid" table with columns "Prescription\_ID" and "Doctor\_ID". The data is as follows:

Prescription_ID	Doctor_ID
P0442	1402
P0443	1402
P0461	1402
P0467	1402
P0471	1402
P0479	1402
P0491	1402
P0002	1403
P0007	1403
P0027	1403
P0033	1403
P0040	1403
P0052	1403
P0060	1402

At the bottom left, there are tabs for "prescribes 28" and "Result 29". The "Output" section shows the execution history:

Action Output	
# Time Action	Message
46 19:24:22 select * from prescribes LIMIT 0, 1000	500 row(s) returned
47 19:24:22 SELECT COUNT() AS total_entries FROM prescribes LIMIT 0, 1000	1 row(s) returned

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 screenshot shows the SQL Server Management Studio interface. In the top pane, a script window titled 'SQL File 4' contains the following SQL code:

```

53
54
55 • UPDATE doctor
56     SET doctor_id = 1000
57     WHERE doctor_id = 1403;
58 • select * from doctor;
59
60
61

```

The bottom pane displays the results of the query in a 'Result Grid'. The grid has columns: Doctor\_ID, First\_Name, Middle\_Name, Last\_Name, DOB, Gender, Email, Specialization, Experience, Street\_Number, Street\_Name, Apartment\_Number, City, Pincode. The data shows 19 rows of doctor information, with the 14th row being updated from Doctor\_ID 1403 to 1000. The 'Output' tab at the bottom shows the execution details for the update statement.

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	84998	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	Broiana	John	Mckenzie	1965-08-14	Male	broiana.mckenzie@gmail.com	Oncology	14	431	Mary Islands	4835	Port Spencer	307542
1502	Janet	Jenifer	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	Jeffery	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	Manifort	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
*	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL

doctor 26 x

Output ::

#	Time	Action	Message	Duration / Fetch
42	19:22:31	UPDATE doctor SET doctor_id = 1000 WHERE doctor_id = 1403	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0	0.016 sec
43	19:22:31	select * from doctor LIMIT 0, 1000	10 row(s) returned	0.000 sec / 0.000 sec

The new Doctor\_ID gets updated in the ‘Prescribes’ table too.

The screenshot shows the SQL Server Management Studio interface. In the top pane, a script window titled 'SQL File 4' contains the following SQL code:

```

62
63
64 • select * from prescribes;
65 • SELECT COUNT(*) AS total_entries FROM prescribes;
66
67
68
69
70

```

The bottom pane displays the results of the query in a 'Result Grid'. The grid has columns: Prescription\_ID, Doctor\_ID. The data shows 500 rows of prescription entries, with the 14th row being updated from Doctor\_ID 1403 to 1000. The 'Output' tab at the bottom shows the execution details for the update statement.

Prescription_ID	Doctor_ID
P0462	1000
P0474	1000
P0481	1000
P0485	1000
P0498	1000
P0017	1401
P0029	1401
P0044	1401
P0045	1401
P0049	1401
P0051	1401
P0072	1401
P0078	1401
...	...
prescribes 19	Result 20
doctor 21	doctor 22
prescribes 23 x	Result 24

Output ::

#	Time	Action	Message	Duration / Fetch
38	18:35:15	select * from prescribes LIMIT 0, 1000	500 row(s) returned	0.000 sec / 0.000 sec
39	18:35:15	SELECT COUNT(*) AS total_entries FROM prescribes LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec

### 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:

$$\begin{aligned} t &\leftarrow \delta_{\text{reimbursement\_status} = \text{"Approved"} }(\text{insurance}) \\ p &\leftarrow t \bowtie \text{claimed\_by} \\ &\Upsilon_{\text{sum(due\_amount)}}(\pi_{\text{due\_amount}}(p)) \end{aligned}$$

#### Screenshot:

The screenshot shows a SQL database interface with the following details:

- Query Editor:** Displays the SQL query:

```
1 • use dispensary1;
2 • SELECT SUM(due_amount)
3 •   FROM (SELECT due_amount
4 •     FROM claimed_by
5 •     NATURAL JOIN (SELECT * FROM insurance WHERE reimbursement_status = "Approved") AS t
6 •   ) AS p;
7
8
9
10
11
12
```
- Result Grid:** Shows the result of the query:

Sum(due_amount)
1170421.00
- Action Output:** Shows the execution log:

#	Time	Action	Message	Duration / Fetch
1	16:56:35	SELECT SUM(due_amount) FROM (SELECT due_amount FROM claimed_by NATURAL JOIN (SELECT... 1 row(s) returned		0.000 sec / 0.000 sec

## 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:

$$\text{prescribes} \leftarrow \text{prescribes} \cup \{ ('P0501', 1050) \}$$

### Screenshot:

The screenshot shows a MySQL command-line interface. The command history pane contains the following SQL code:

```
1 • use dispensary1;
2 • insert into prescribes(prescription_ID, Doctor_ID) values ('P0501',1050);
3
4
```

The output pane shows the following results:

#	Time	Action
1	16:50:40	insert into prescribes(prescription_ID, Doctor_ID) values ('P0501',1050)

A message box displays the error:

Message  
Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails ('dispensary1','prescribes',CON...  
Duration / Fetch 0.000 sec

## 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:

$$\begin{aligned} \text{left\_join} &\leftarrow \rho_d(\text{doctor}) \bowtie \rho_o(\text{opd}) \\ \text{filter} &\leftarrow \delta_{o.date > '2023-11-01' \wedge o.date < '2023-12-01'} (\text{left\_join}) \\ \text{grouped} &\leftarrow \gamma_{o.doctor_id, \text{count}(o.patient_id)} \text{ AS } \text{number\_of\_patients\_checked} (\text{filter}) \\ \pi_{o.doctor_id, \text{number\_of\_patients\_checked}} &(\text{grouped}) \end{aligned}$$

### Screenshot:

The screenshot shows a SQL query being run in a database environment. The query is as follows:

```
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) ;
```

The result grid displays the following data:

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=Hl0D2eo-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=qEwRxoAiej4QzcT9c
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:

```

74 • drop table if exists st_image;
75 • create table st_image(
76     img_id INT unsigned not null auto_increment primary key,
77     caption varchar(55) not null,
78     img longblob default null
79 );
80 • insert into st_image (caption, img) values
81 ("William", "https://as2.ftcdn.net/v2/jpg/06/01/95/47/1000_F_601954739_dJ0VcsE17js0vq8Ag2hx8giMpo71km3o.jpg"),
82 ("Janet", "https://media.istockphoto.com/id/1270790502/photo/medical-concept-of-indian-beautiful-female-doctor-with-note-book.jpg?s=612x612&w=
83 ("Courtney", "https://media.istockphoto.com/id/1250738979/photo/3d-cartoon-character-medical-doctor.webp?b=1&s=170667a&w=0&k=20&c=4pj4U90eh2XA
84 ("Brandon", "https://media.istockphoto.com/id/859284102/photo/3d-doctor-thinking.webp?b=1&s=170667a&w=0&k=20&c=VQWRiLvafak86eEUTR1vHdy8Wh6R2Xf
85

```

img_id	caption	img
1	peter parker	BLOB
2	William	BLOB
3	Janet	BLOB
4	Courtney	BLOB
5	Brandon	BLOB
HULL	HULL	HULL

## 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 function 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:

$$\text{original\_staff} \leftarrow \Pi_{\text{staff\_id}, \text{salary}, \text{join\_date}}$$
  

$$\text{min\_join\_date} \leftarrow \Pi_{\text{join\_date}}(\delta_{\text{min}(\text{join\_date})}(\text{original\_staff}))$$
  

$$\text{updated\_salary} \leftarrow \Pi_{\text{staff\_id}, (\text{CASE})}$$
  

$$(\text{WHEN } \text{join\_date} = \text{min\_join\_date} \text{ THEN } \text{salary} * 1.5)$$
  

$$(\text{WHEN } \text{join\_date} < '2018-12-31' \text{ THEN } \text{salary} * 1.1)$$
  

$$(\text{WHEN } \text{join\_date} < '2020-12-31' \text{ THEN } \text{salary} * 1.05)$$
  

$$(\text{ELSE } \text{salary})$$
  

$$(\text{END}) \text{ AS salary}$$
  

$$(\text{original\_staff})$$
  

$$\text{final\_staff} \leftarrow (\text{staff\_salary}) \cup \text{updated\_salary}$$

## Screenshot:

The screenshot shows a MySQL Workbench interface. In the top-left, there is a code editor window containing the following SQL code:

```
8 • UPDATE staff
1   SET salary =
2   CASE
3       when join_date = (select min(join_date) from staff) then salary*1.5
4       when join_date < '2018-12-31' then salary * 1.1
5       when join_date < '2020-12-31' then salary * 1.05
6   ELSE salary
7 END;
8
```

In the bottom-right corner of the code editor, there are "Context Help" and "Snippets" buttons.

Below the code editor is a "put" button.

At the bottom of the interface is an "Action Output" table:

#	Time	Action	Message	Duration / Fetch
125	17:13:02	UPDATE staff SET salary = CASE when join_date = (select min(join_date) from staff) then salary*1.5 ...	Error Code: 1111. Invalid use of group function	0.000 sec
126	17:13:54	UPDATE staff SET salary = CASE when join_date < '2018-12-31' then salary * 1.1 ...	Error Code: 1093. You can't specify target table 'staff' for update in FROM clause	0.000 sec
127	17:14:28	UPDATE staff SET salary = CASE when join_date < '2020-12-31' then salary * 1.05 ...	Error Code: 1093. You can't specify target table 'staff' for update in FROM clause	0.000 sec

## 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.