{

CPS 510
Assignment 10

Medical Clinic Information DBMS

Suhaib Khan
Kevin Dao

};

Table of Contents

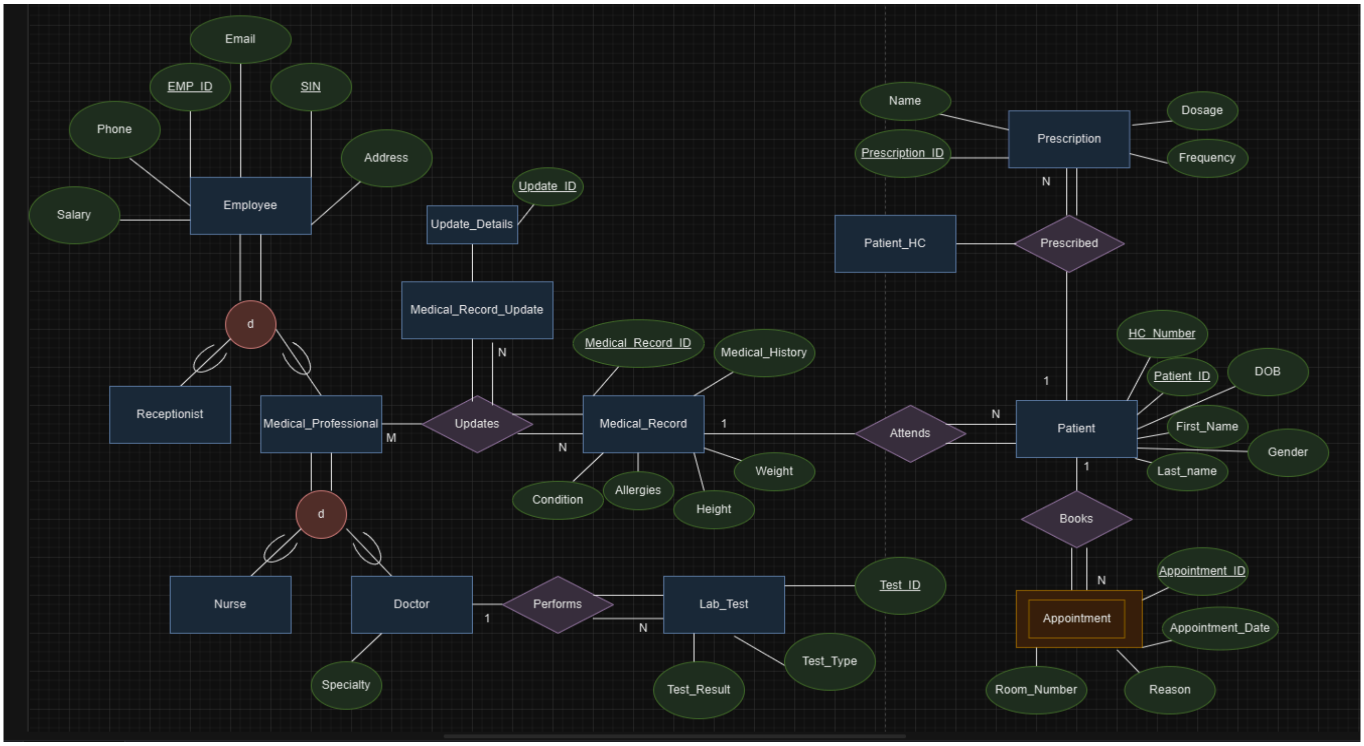
Introduction

Medical clinics play a vital role in society. It is an efficient place where a patient can go for various checkups, tests, and overall health assessments. The information that a clinic has about its patients is very detailed and sensitive which means that it must be kept in an organized manner. That being said, the information must also be readily accessible as quickly as possible. The topic we propose is a Medical clinic information database system. This will store entities such as a patient's identifiable information, medical information, prescription details, appointments, Lab Tests, Doctor details, Health Card information and more, including all their attributes, so employees can easily access all of a patient's medical information.

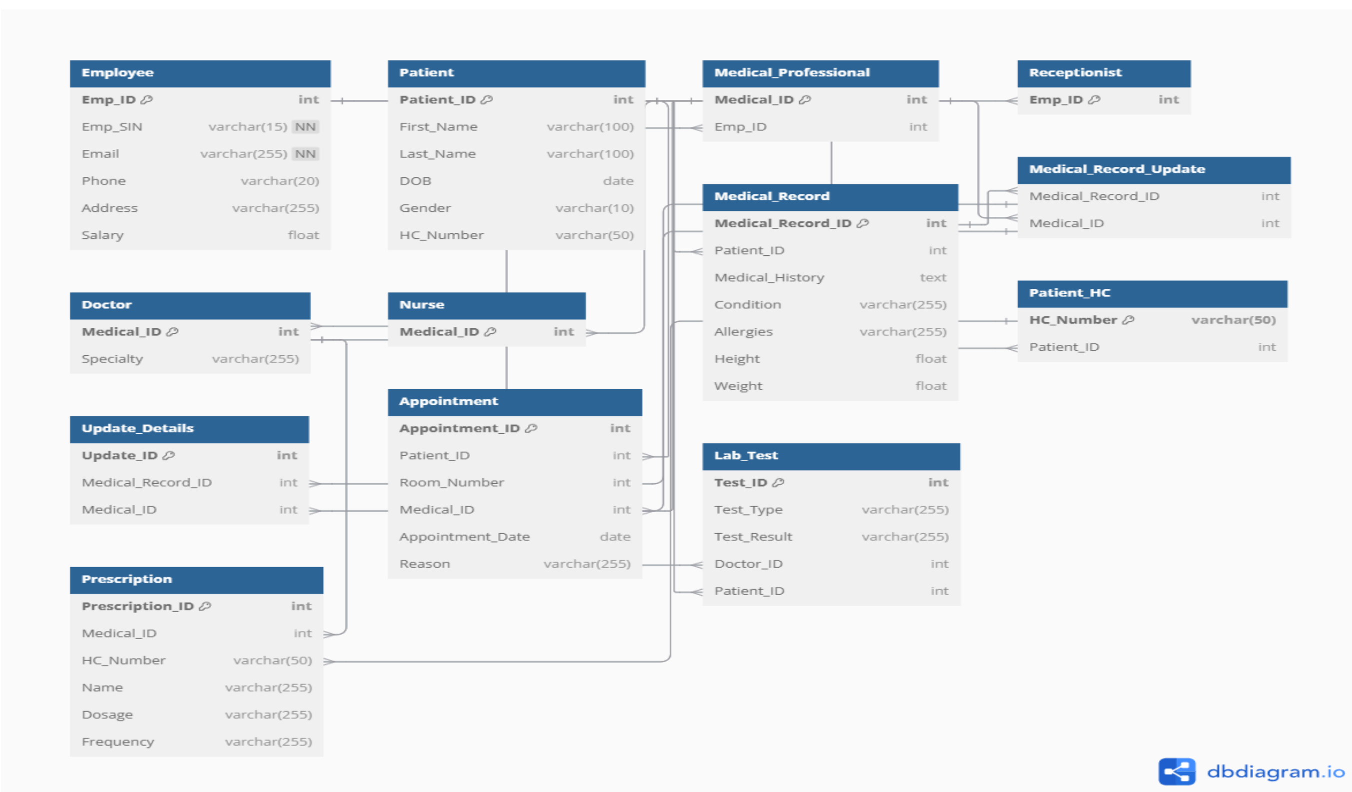
To make this database system more efficient for the particular application of a medical clinic, it is also able to store the information about the employees. Employees have essential information such as salary, SIN, email, and addresses stored just in case they might be needed for employment purposes.

Practical uses of this database design, queries, and all of the architecture in this project may be used for checking active appointments in the clinic, analysis of any patients who have anomalies in their conditions, easily checking if the salaries of the employees are consistent, and many more.

ER Diagrams



Email
EMP_ID
SIN
Phone
Address
Employee
Salary
Update_ID
Update_Details
Medical_Record_Update
d
Receptionist
Medical_Professional
Updates
Medical_Record
Medical_Record_ID
Medical_History
M
N
N
1
Condition
Allergies
Weight
Height
d
Nurse
Doctor
Specialty
Performs
1
N
Lab_Test
Test_ID
Test_Type
Test_Result
Name
Prescription
Dosage
Prescription_ID
Frequency
N
Patient_HC
Prescribed
HC_Number
Patient_ID
DOB
1
Attends
N
Patient
First_Name
Gender
Last_name
1
Books
N
Appointment_ID
Appointment
Appointment_Date
Room_Number
Reason



Employee
Emp_ID int
Emp_SIN varchar(15) NN
Email varchar(255) NN
Phone varchar(20)
Address varchar(255)
Salary float
Patient
Patient_ID int
First_Name varchar(100)
Last_Name varchar(100)
DOB date
Gender varchar(10)
HC_Number varchar(50)
Medical_Professional
Medical_ID int
Emp_ID int
Receptionist
Emp_ID int
Medical_Record_Update
Medical_Record_ID int
Medical_ID int
Medical_Record
Medical_Record_ID int
Patient_ID int
Medical_History text
Condition varchar(255)
Allergies varchar(255)
Height float
Weight float
Patient_HC
HC_Number varchar(50)
Patient_ID int
Doctor
Medical_ID int
Specialty varchar(255)
Nurse
Medical_ID int
Update_Details
Update_ID int
Medical_Record_ID int
Medical_ID int
Appointment
Appointment_ID int
Patient_ID int
Room_Number int
Medical_ID int
Appointment_Date date
Reason varchar(255)
Lab_Test
Test_ID int
Test_Type varchar(255)
Test_Result varchar(255)
Doctor_ID int
Patient_ID int
Prescription
Prescription_ID int
Medical_ID int
HC_Number varchar(50)
Name varchar(255)
Dosage varchar(255)
Frequency varchar(255)
dbdiagram.io

Tables, Functional Dependencies, and Normalization

```sql
CREATE TABLE Employee (
    Emp_ID INT PRIMARY KEY,
    Emp_SIN VARCHAR(15) NOT NULL UNIQUE,
    Email VARCHAR(255) NOT NULL,
    Phone VARCHAR(20),
    Address VARCHAR(255),
    Salary DECIMAL(10, 2)
);
```

Emp_ID → {Emp_SIN, Email, Phone, Address, Salary}
Emp_SIN → {Emp_ID, Email, Phone, Address, Salary}

All values are atomic, so Employee is 1NF
There are no partial dependencies, so Employee is 2NF
There are no transitive dependencies, so Employee is 3NF
Emp_ID and EMP_SIN are both candidate keys, so Employee is BCNF

```sql
CREATE TABLE Patient (
    Patient_ID INT PRIMARY KEY,
    First_Name VARCHAR(100),
    Last_Name VARCHAR(100),
    DOB DATE,
    Gender VARCHAR(10),
    HC_Number VARCHAR(50)
);
```

Patient_ID → {First_Name, Last_Name, DOB, Gender, HC_Number}
HC_Number → {Patient_ID, First_Name, Last_Name, DOB, Gender}

All values are atomic, so Patient is 1NF
There are no partial dependencies, so Patient is 2NF
There are no transitive dependencies, so Patient is 3NF
Patient_ID and HC_Number are both candidate keys, so Patient is BCNF

```sql
CREATE TABLE Medical_Professional (
    Medical_ID INT PRIMARY KEY,
    Emp_ID INT,
    FOREIGN KEY (Emp_ID) REFERENCES Employee(Emp_ID)
);
```

Medical_ID → {Emp_ID}
Emp_ID → {Medical_ID}

All values are atomic, so Medical_Professional is 1NF
There are no partial dependencies, so Medical_Professional is 2NF

There are no transitive dependencies, so Medical_Professional is 3NF
Medical_ID and Emp_ID are both candidate keys, so Medical_Professional is BCNF

```
CREATE TABLE Receptionist (
    Emp_ID INT PRIMARY KEY,
    FOREIGN KEY (Emp_ID) REFERENCES Employee(Emp_ID)
);
```

There are no Functional Dependencies in this table, as it is only one attribute

Emp_ID is atomic, so Receptionist is 1NF
Since there is only one attribute, Receptionist is trivially 2NF, 3NF, and BCNF

```
CREATE TABLE Doctor (
    Medical_ID INT PRIMARY KEY,
    Specialty VARCHAR(255),
    FOREIGN KEY (Medical_ID) REFERENCES Medical_Professional(Medical_ID)
);
```

Medical_ID → {Specialty}

All values are atomic, so Doctor is 1NF
There are no partial dependencies, so Doctor is 2NF
There are no transitive dependencies, so Doctor is 3NF
Medical_ID is the candidate key, so Doctor is BCNF

```
CREATE TABLE Nurse (
    Medical_ID INT PRIMARY KEY,
    FOREIGN KEY (Medical_ID) REFERENCES Medical_Professional(Medical_ID)
);
```

There are no Functional Dependencies in this table, as it is only one attribute

Medical_ID is atomic, so Nurse is 1NF
Since there is only one attribute, Nurse is trivially 2NF, 3NF, and BCNF

```
CREATE TABLE Medical_Record (
    Medical_Record_ID INT PRIMARY KEY,
    Patient_ID INT,
    Medical_History TEXT,
    `Condition` VARCHAR(255),
    Allergies VARCHAR(255),
    Height DECIMAL(5, 2),
    Weight DECIMAL(5, 2),
    FOREIGN KEY (Patient_ID) REFERENCES Patient(Patient_ID)
);
```

Medical_Record_ID → {Patient_ID, Medical_History, Condition, Allergies, Height, Weight}

All values are atomic, so Medical_Record is 1NF

There are no partial dependencies, so Medical_Record is 2NF
There are no transitive dependencies, so Medical_Record is 3NF
Medical_Record_ID is the candidate key, so Medical_Record is BCNF

```
CREATE TABLE Medical_Record_Update (
    Medical_Record_ID INT,
    Medical_ID INT,
    PRIMARY KEY (Medical_Record_ID, Medical_ID),
    FOREIGN KEY (Medical_Record_ID) REFERENCES Medical_Record(Medical_Record_ID),
    FOREIGN KEY (Medical_ID) REFERENCES Medical_Professional(Medical_ID)
);
```

There are no Functional Dependencies in this table, as both attributes are primary keys

Both attributes are atomic, so Medical_Record_Update is 1NF
Since both attributes are PK's, Medical_Record_Update is trivially 2NF, 3NF, and BCNF

```
CREATE TABLE Update_Details (
    Update_ID INT PRIMARY KEY,
    Medical_Record_ID INT,
    Medical_ID INT,
    FOREIGN KEY (Medical_Record_ID, Medical_ID) REFERENCES
Medical_Record_Update(Medical_Record_ID, Medical_ID)
);
```

Update_ID → {Medical_Record_ID, Medical_ID}

All values are atomic, so Update_Details is 1NF
There are no partial dependencies, so Update_Details is 2NF
There are no transitive dependencies, so Update_Details is 3NF
Update_ID is the candidate key, so Update_Details is BCNF

```
CREATE TABLE Appointment (
    Appointment_ID INT PRIMARY KEY,
    Patient_ID INT,
    Room_Number INT,
    Medical_ID INT,
    Appointment_Date TIMESTAMP,
    Reason VARCHAR(255),
    FOREIGN KEY (Patient_ID) REFERENCES Patient(Patient_ID),
    FOREIGN KEY (Medical_ID) REFERENCES Medical_Professional(Medical_ID)
);
```

Appointment_ID → {Patient_ID, Room_Number, Medical_ID, Appointment_Date, Reason}

All values are atomic, so Appointment is 1NF
There are no partial dependencies, so Appointment is 2NF
There are no transitive dependencies, so Appointment is 3NF
Appointment_ID is the candidate key, so Appointment is BCNF

```sql
CREATE TABLE Lab_Test (
    Test_ID INT PRIMARY KEY,
    Test_Type VARCHAR(255),
    Test_Result VARCHAR(255),
    Doctor_ID INT,
    Patient_ID INT,
    FOREIGN KEY (Doctor_ID) REFERENCES Doctor(Medical_ID),
    FOREIGN KEY (Patient_ID) REFERENCES Patient(Patient_ID)
);
```

Test_ID → {Test_Type, Test_Result, Doctor_ID, Patient_ID}

All values are atomic, so Lab_Test is 1NF
There are no partial dependencies, so Lab_Test is 2NF
There are no transitive dependencies, so Lab_Test is 3NF
Test_ID is the candidate key, so Lab_Test is BCNF

```sql
CREATE TABLE Patient_HC (
    HC_Number VARCHAR(50) PRIMARY KEY,
    Patient_ID INT,
    FOREIGN KEY (Patient_ID) REFERENCES Patient(Patient_ID)
);
```

HC_Number → {Patient_ID}

All values are atomic, so Patient_HC is 1NF
There are no partial dependencies, so Patient_HC is 2NF
There are no transitive dependencies, so Patient_HC is 3NF
HC_Number is the candidate key, so Patient_HC is BCNF

```sql
CREATE TABLE Prescription (
    Prescription_ID INT PRIMARY KEY,
    Medical_ID INT,
    HC_Number VARCHAR(50),
    Name VARCHAR(255),
    Dosage VARCHAR(255),
    Frequency VARCHAR(255),
    FOREIGN KEY (HC_Number) REFERENCES Patient_HC(HC_Number),
    FOREIGN KEY (Medical_ID) REFERENCES Doctor(Medical_ID)
);
```

Prescription_ID → {Medical_ID, HC_Number, Name, Dosage, Frequency}

All values are atomic, so Prescription is 1NF
There are no partial dependencies, so Prescription is 2NF
There are no transitive dependencies, so Prescription is 3NF
Prescription_ID is the candidate key, so Prescription is BCNF

Dummy Inserts

Here, we show some dummy insert values we added in the Database to show how the database storage works. 1 Insert for each of the 13 tables is shown.

```
INSERT INTO Employee VALUES (9, 'SIN135462756', 'lisa.su@gmail.com', '135-046-2756',
'3211 Birch St', 40000.00);

INSERT INTO Patient VALUES (1, 'John', 'Doe', '1985-05-15', 'Male', 'HC12345');

INSERT INTO Medical_Professional VALUES (1, 1);

INSERT INTO Receptionist VALUES (2);

INSERT INTO Doctor VALUES (1, 'Cardiology');

INSERT INTO Nurse VALUES (5);

INSERT INTO Medical_Record VALUES (2, 2, 'Asthma', 'Obesity', 'Pollen', 186, 120);

INSERT INTO Medical_Record_Update VALUES (1, 1);

INSERT INTO Update_Details VALUES (1, 1, 1);

INSERT INTO Appointment VALUES (1, 1, 101, 1, '2024-12-01 10:00:00', 'Routine Checkup');

INSERT INTO Lab_Test VALUES (1, 'Blood Test', 'Normal', 1, 1);

INSERT INTO Patient_HC VALUES ('HC12345', 1);

INSERT INTO Prescription VALUES (2, 2, 'HC54321', 'Ventolin', '2 Puffs', 'As Needed');
```

2NF and 3NF Table Decomposition Example

Original Updates Table:

```sql
CREATE TABLE Updates (
    Medical_Record_ID INT,
    Medical_ID INT,
    Update_ID INT PRIMARY KEY,
    FOREIGN KEY (Medical_Record_ID) REFERENCES
Medical_Record(Medical_Record_ID),
    FOREIGN KEY (Medical_ID) REFERENCES Medical_Professional(Medical_ID)
);
```

Decomposition to 2NF for Updates table:

- Split Updates to remove partial dependency, creating two tables:

```sql
CREATE TABLE Medical_Record_Update (
    Medical_Record_ID INT,
    Medical_ID INT,
    PRIMARY KEY (Medical_Record_ID, Medical_ID),
    FOREIGN KEY (Medical_Record_ID) REFERENCES
Medical_Record(Medical_Record_ID),
    FOREIGN KEY (Medical_ID) REFERENCES Medical_Professional(Medical_ID)
 );
```

- Update_Details table:

```sql
CREATE TABLE Update_Details (
    Update_ID INT PRIMARY KEY,
    Medical_Record_ID INT,
    Medical_ID INT,
    FOREIGN KEY (Medical_Record_ID, Medical_ID) REFERENCES
Medical_Record_Update(Medical_Record_ID, Medical_ID)
);
```

Original Prescription Table:

```sql
CREATE TABLE Prescription (
    Prescription_ID INT PRIMARY KEY,
    Medical_ID INT,
    HC_Number VARCHAR(50),
    Name VARCHAR(255),
    Dosage VARCHAR(255),
    Frequency VARCHAR(255),
    Patient_ID INT,
    FOREIGN KEY (Patient_ID) REFERENCES Patient(Patient_ID),
    FOREIGN KEY (Medical_ID) REFERENCES Doctor(Medical_ID)
);
```

Decomposition to 3NF for Prescription Table:

- Remove transitive dependency by splitting Prescription into two tables:

```sql
CREATE TABLE Patient_HC (
    HC_Number VARCHAR(50) PRIMARY KEY,
    Patient_ID INT,
    FOREIGN KEY (Patient_ID) REFERENCES Patient(Patient_ID)
);
```

- Modified Prescription Table:

```sql
CREATE TABLE Prescription (
    Prescription_ID INT PRIMARY KEY,
    Medical_ID INT,
    HC_Number VARCHAR(50),
    Name VARCHAR(255), Dosage VARCHAR(255),
    Frequency VARCHAR(255),
    FOREIGN KEY (HC_Number) REFERENCES Patient_HC(HC_Number),
    FOREIGN KEY (Medical_ID) REFERENCES Doctor(Medical_ID)
);
```

Berenstein Algorithm Example

This Berenstein Algorithm Example is used on the Prescription Table:

Step 1:
R(Prescription_ID, Medical_ID, HC_Number, Name, Dosage, Frequency, Patient_ID}

FDs: {Prescription_ID, Medical_ID → HC_Number,
Prescription_ID, Medical_ID → Name,
Prescription_ID, Medical_ID → Dosage,
Prescription_ID, Medical_ID → Frequency,
Prescription_ID, Medical_ID → Patient_ID,
HC_Number → Patient_ID,
HC_Number → Name,
Patient_ID → HC_Number,
Patient_ID → Name}

Step 2:
From the information above, the FDs Patient_ID → Name, Prescription_ID, Medical_ID → Name are redundant.
Left hand side of the remaining FDs are minimal

Step 3:

- Three candidate keys:
  {Prescription_ID, Medical_ID}
  {HC_Number}
  {Patient_ID}

Step 4:
We get n relations:

R1(Prescription_ID, Medical_ID, HC_Number, Name, Dosage, Frequency, Patient_ID)

R2(HC_Number, Patient_ID, Name)

R3(Patient_ID, HC_Number, Name)

Since Attributes R2 and R3 are a subset of R1, we can eliminate R2 and R3

Final Schema:
R1(Prescription_ID, Medical_ID, HC_Number, Name, Dosage, Frequency, Patient_ID)

Simple Queries, RA's, and Views

Here are the simple queries with the results using additional dummy inserts not shown, as well as views, and relational algebras.

```
-- This query calculates the total salary for each unique job title, helping the clinic manage payroll expenses across different roles.
SELECT DISTINCT Emp_ID, SUM(Salary) AS Total_Salary
FROM Employee
GROUP BY Emp_ID
ORDER BY Total_Salary DESC;

EMP_ID TOTAL_SALARY
---------- ------------
         8        90000
         7        85000
         5        75000
         3        70000
         6        65000
         4        60000
         1        55000
         2        45000
         9        40000
        10        35000
```

$_{Emp\_ID}F_{SUM(Salary)}(Employee)$

```
-- Shows the distribution of patients by gender, helping the clinic analyze its patient demographic breakdown.
SELECT DISTINCT Gender, COUNT(Patient_ID) AS Patient_Count
FROM Patient
GROUP BY Gender
ORDER BY Patient_Count DESC;

GENDER     PATIENT_COUNT
---------- -------------
Male                   3
Female                 2
```

$_{Gender}F_{COUNT(Patient\_ID)}(Patient)$

```sql
-- Displays Employee ID with their associated medical ID
SELECT Emp_ID, Medical_ID
FROM Medical_Professional;

 EMP_ID MEDICAL_ID
---------- ----------
         1          1
         4          2
         5          3
         6          4
         7          5
         8          6
         9          7
        10          8
```

$\Pi_{Emp\_ID,\ Medical\_ID}(Medical\_Professional)$

```sql
-- Shows a list of all unique receptionists in the clinic
SELECT Emp_ID
FROM Receptionist
ORDER BY Emp_ID;

 EMP_ID
----------
         2
         3
```

$\Pi_{Emp\_ID}(Receptionist)$

```sql
-- Shows how many doctors specialize in different fields, allowing the
clinic to balance its specialization focus.
SELECT DISTINCT Specialty, COUNT(Medical_ID) AS Doctor_Count
FROM Doctor
GROUP BY Specialty
ORDER BY Doctor_Count DESC;

SPECIALTY       DOCTOR_COUNT
--------------- ------------
Neurology                  1
Dermatology                1
```

${}_{Specialty}F_{COUNT(Medical\_ID)}(Doctor)$

```
-- Displays how many nurses are available in the clinic, helping with
staffing and resource allocation.
SELECT DISTINCT Medical_ID, COUNT(Medical_ID) AS Nurse_Count
FROM Nurse
GROUP BY Medical_ID
ORDER BY Nurse_Count DESC;

MEDICAL_ID NURSE_COUNT
---------- -----------
         5           1
         8           1
         7           1
         6           1
```

$_{Medical\_ID}F_{COUNT(Medical\_ID)}(Nurse)$

```
-- Displays the Patient ID with their Name and Dosage for easy viewing
SELECT HC_Number, Frequency, Dosage
FROM Prescription;

HC_NUMBER  FREQUENCY                      DOSAGE
---------- ------------------------------ ------------------------------
HC12345    Twice a Day                    500mg
HC54321    As Needed                      2 Puffs
HC98765    Three Times a Day              200mg
HC56789    As Needed                      50mg
```

$\Pi_{HC\_Number,\ Frequency,\ Dosage}(Prescription)$

```
-- Shows which patients have the most appointments
SELECT DISTINCT Patient_ID, COUNT(Appointment_ID) AS Appointment_Count
FROM Appointment
GROUP BY Patient_ID
ORDER BY Appointment_Count DESC;

PATIENT_ID APPOINTMENT_COUNT
---------- -----------------
         1                 1
         2                 1
         3                 1
```

$_{Patient\_ID}F_{COUNT(Appointment\_ID)}(Appointment)$

```sql
-- This Medical Record Query checks for patients with Obesity, and it displays their height and weight for verification
SELECT Patient_ID, Height AS Height_CM, Weight AS Weight_KG, Condition
FROM Medical_Record
WHERE Condition LIKE '%Obesity%'
ORDER BY Weight;
```

PATIENT_ID	HEIGHT_CM	WEIGHT_KG	CONDITION
5	159	80	Obesity
3	200	90	Obesity
2	186	120	Obesity

$\Pi_{\text{Patient\_ID, Height, Weight, Condition}}\ (\sigma_{\text{Condition LIKE 'Obesity'}}(\text{Medical\_Record}))$

```sql
-- Check Which Medical Proffesional has Been updating the records, to keep in track of how consistent each medical proffesional is
SELECT Medical_id, COUNT(Update_id) AS Update_Count
FROM Update_Details
GROUP BY Medical_id
ORDER BY Update_count;
```

MEDICAL_ID	UPDATE_COUNT
1	1
2	1
3	1
5	1
4	1

${}_{\text{Medical\_ID}}F_{\text{COUNT(Update\_ID)}}(\text{Update\_Details})$

```sql
-- Check the Number of Results each lab test result has, this will allow the clinic to correctly stock up on treatments
SELECT Test_result AS Status, COUNT(*) AS Result_Count
FROM Lab_test
GROUP BY Test_result
ORDER BY Result_Count DESC;
```

```
STATUS                         RESULT_COUNT
------------------------------ ------------
No Abnormalities                          1
Normal                                    1
Inflammation Detected                     1
```

$_{Test\_Result}F_{COUNT(*)}(Lab\_Test)$

```sql
-- Count the amount of perscriptions each doctor has given out
SELECT D.Medical_ID, COUNT(Pr.Prescription_ID) AS Prescription_Count
FROM Doctor D
JOIN Prescription Pr ON D.Medical_ID = Pr.Medical_ID
GROUP BY D.Medical_ID
ORDER BY Prescription_Count DESC;
```

```
MEDICAL_ID PRESCRIPTION_COUNT
---------- ------------------
         1                  1
         3                  1
         4                  1
         2                  1
```

$_{Medical\_ID}F_{COUNT(Prescription\_ID)}(\Pi_{D.Medical\_ID,\ Pr.Prescription\_ID}\ (Doctor \bowtie Prescription))$

```sql
-- Retrieve details of appointments
SELECT A.Appointment_ID, P.First_Name, P.Last_Name, MP.Medical_ID,
A.Reason, A.Appointment_Date
FROM Appointment A
JOIN Patient P ON A.Patient_ID = P.Patient_ID
JOIN Medical_Professional MP ON A.Medical_ID = MP.Medical_ID;
```

```
APPOINTMENT_ID FIRST_NAME      LAST_NAME       MEDICAL_ID REASON
APPOINTMENT_DATE
-------------- --------------- --------------- ----------
------------------------------ ---------------------------
             1 John            Doe                      1 Routine Checkup
24-12-01 10:00:00.000000000
             2 Jane            Smith                    2 Asthma Follow-Up
24-12-02 11:00:00.000000000
             3 Michael         Brown                    3 Pain Management
24-12-03 09:30:00.000000000
             4 Lisa            White                    4 Migraine
```

$\Pi_{\text{A.Appointment\_ID, P.First\_Name, P.Last\_Name, MP.Medical\_ID, A.Reason, A.Appointment\_Date}}((\text{Appointment} \bowtie \text{Patient}) \bowtie \text{Medical\_Professional})$

```sql
-- List patients along with their associated medical records
SELECT P.Patient_ID, MR.Medical_Record_ID, MR.Condition, MR.Allergies
FROM Patient P
JOIN Medical_Record MR ON P.Patient_ID = MR.Patient_ID;
```

```
PATIENT_ID MEDICAL_RECORD_ID CONDITION                      ALLERGIES
---------- ----------------- ------------------------------
------------------------------
         1                 1 None                           None
         2                 2 Obesity                        Pollen
         3                 3 Obesity                        None
         4                 4 Cancer                         Gluten
         5                 5 Obesity                        Penicillin
```

$\Pi_{\text{P.Patient\_ID, MR.Medical\_Record\_ID, MR.Condition, MR.Allergies}}(\text{Patient} \bowtie \text{Medical\_Record})$

```sql
-- VIEWS
-- Create a view that lists doctors and their specialties.
CREATE OR REPLACE VIEW Doctor_Specialty_View AS
SELECT E.Emp_ID, MP.Medical_ID, D.Specialty, E.Salary
FROM Employee E
JOIN Medical_Professional MP ON E.Emp_ID = MP.Emp_ID
JOIN Doctor D ON MP.Medical_ID = D.Medical_ID;

-- Create a view to simplify access to patient prescription details.
CREATE OR REPLACE VIEW Prescription_Details_View AS
SELECT PH.Patient_ID, Pr.Name, Pr.Dosage, Pr.Frequency
FROM Prescription Pr
JOIN Patient_HC PH ON Pr.HC_Number = PH.HC_Number
JOIN Patient P ON PH.Patient_ID = P.Patient_ID;

-- Create a view summarizing appointments by patient.
CREATE OR REPLACE VIEW Appointment_Summary_View AS
SELECT A.Patient_ID, COUNT(A.Appointment_ID) AS Total_Appointments
FROM Appointment A
GROUP BY A.Patient_ID;
```

Advanced Queries and RA's

```
-- Query 1: Query which shows a list of doctors who have prescribed less than 5 prescriptions,
-- Useful for documentation and to see which doctors are not prescribing as much as others
SELECT D.Medical_ID, D.Specialty, COUNT(P.Prescription_ID)
FROM Doctor D
JOIN Prescription P ON D.Medical_ID = P.Medical_ID
GROUP BY D.Medical_ID, D.Specialty
HAVING COUNT(P.Prescription_ID) < 5;

MEDICAL_ID SPECIALTY       COUNT(P.PRESCRIPTION_ID)
---------- -------------- ------------------------
         3 Pediatrics                             1
         1 Cardiology                             1
         2 Neurology                              1
```

$\sigma_{Prescription\_Count<5}({}_{Medical\_ID,\ Specialty}F_{COUNT(Prescription\_ID)} \rightarrow Prescription\_Count(\Pi_{D.Medical\_ID,\ D.Specialty,\ P.Prescription\_ID}(Doctor \bowtie Prescription)))$

```
-- Get avg height and weight for both M and F patients seperately.
SELECT P.Gender, AVG(MR.Height) AS Avg_Height, AVG(MR.Weight) AS Avg_Weight
FROM Patient P
JOIN Medical_Record MR ON P.Patient_ID = MR.Patient_ID
GROUP BY P.Gender
HAVING COUNT(P.PATIENT_ID) > 1;

GENDER     AVG_HEIGHT AVG_WEIGHT
---------- ---------- ----------
Male       176.333333         79
Female          176.5       94.5
```

$\sigma_{Patient\_Count\ >1}({}_{Gender}F_{AVG(MR.Height)} \rightarrow Avg\_Height,\ AVG(MR.Weight) \rightarrow Avg\_Weight,\ COUNT(P.Patient\_ID) \rightarrow Patient\_Count(\Pi_{P.Gender,\ MR.Height,\ MR.Weight,\ P.Patient\_ID}(Patient \bowtie Medical\_Record)))$

```sql
-- Query 3 (Rewritten): Check which patients do not have an appointment booked in the database
SELECT Patient.Patient_ID, Patient.First_Name, Patient.Last_Name
FROM Patient
WHERE NOT EXISTS (
    SELECT 1
    FROM Appointment
    WHERE Appointment.Patient_ID = Patient.Patient_ID
);
no rows selected
```

$\Pi_{Patient\_ID,\ First\_Name,\ Last\_Name}(Patient - \Pi_{Patient\_ID,\ First\_Name,\ Last\_Name}(Patient \bowtie (\Pi_{Patient\_ID}(Appointment))))$

```sql
-- Query 4: Selecting patients who have above average weight in the database
-- This will help in seeing which patients physical health may need a closer look
SELECT Medical_Record.Patient_ID, Patient.First_Name, Patient.Last_Name, Medical_Record.Weight
FROM Medical_Record
JOIN Patient ON Medical_Record.Patient_ID = Patient.Patient_ID
WHERE Medical_Record.Weight > (SELECT AVG(Weight) FROM Medical_Record);

PATIENT_ID FIRST_NAME      LAST_NAME           WEIGHT
---------- --------------- --------------- ----------
         2 Jane            Smith                  120
         3 Michael         Brown                   90
```

$\Pi_{Medical\_Record.Patient\_ID,\ Patient.First\_Name,\ Patient.Last\_Name,\ Medical\_Record.Weight}(\sigma_{Medical\_Record.Weight>AVG\_Weight}((Medical\_Record \bowtie Patient) \bowtie F_{AVG(Weight)(Medical\_Record)}))$

-- Query 5: Query which shows a list of doctors who have done less than 5 lab tests,
-- Useful for documentation and to see which doctors are not testing as much as others

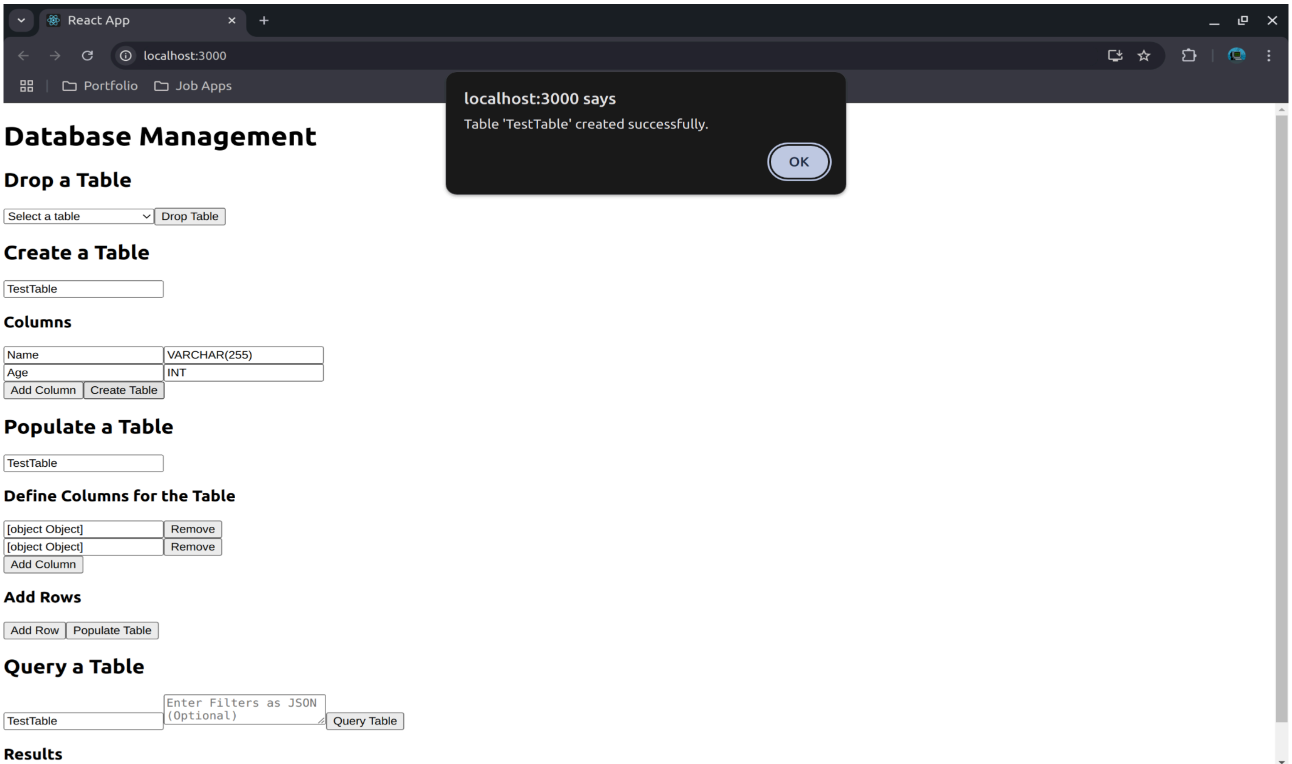
```
SELECT D.Medical_ID, D.Specialty, COUNT(LT.Test_ID) AS Lab_Test_Count
FROM Doctor D
JOIN Lab_Test LT ON D.Medical_ID = LT.Doctor_ID
GROUP BY D.Medical_ID, D.Specialty
HAVING COUNT(LT.Test_ID) < 5;
```

```
MEDICAL_ID SPECIALTY       LAB_TEST_COUNT
---------- --------------- --------------
         3 Pediatrics                   1
         1 Cardiology                   1
         2 Neurology                    1
         4 Dermatology                  1
```

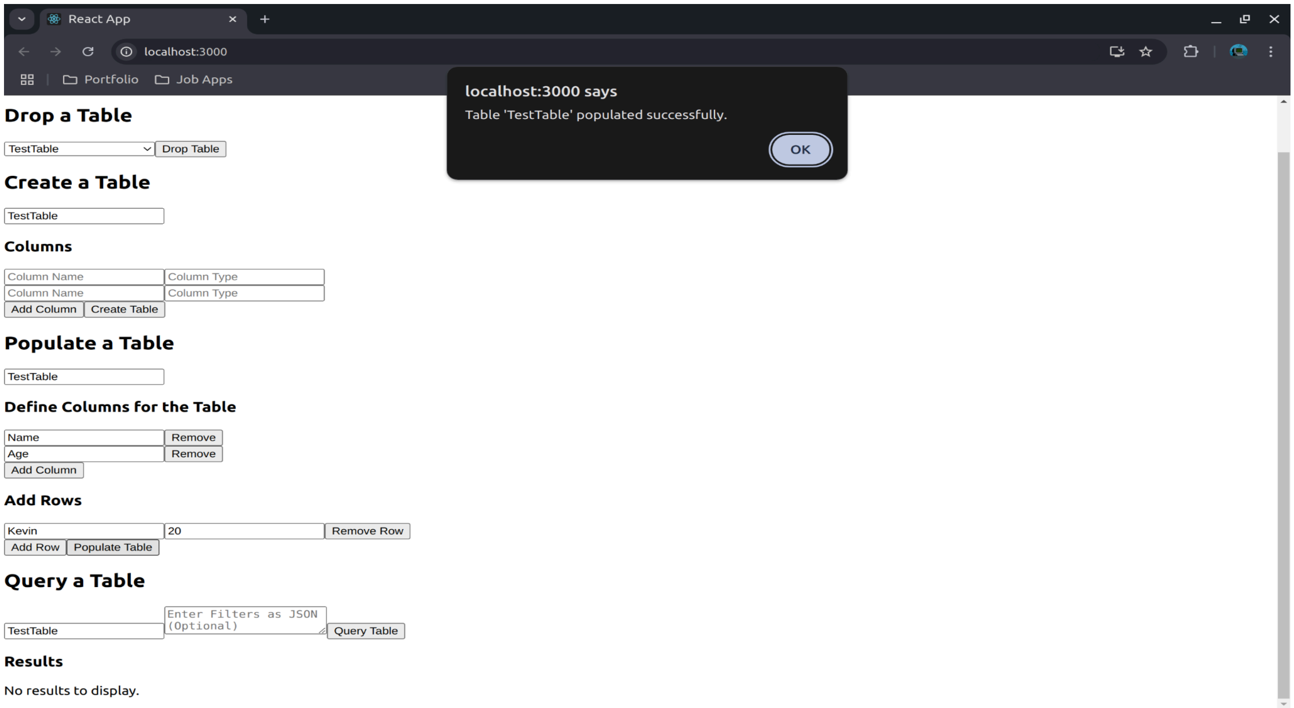
$\sigma_{\text{Lab\_Test\_Count}<5}({}_{\text{Medical\_ID, Specialty}}F_{\text{COUNT(LT.Test\_ID)}} \rightarrow \text{Lab\_Test\_Count } (\Pi_{\text{D.Medical\_ID, D.Specialty, LT.Test\_ID}}(\text{Doctor} \bowtie \text{Lab\_Test})))$

UI / GUI Implementation

Creating a Table using the "Create a Table" function in the GUI:

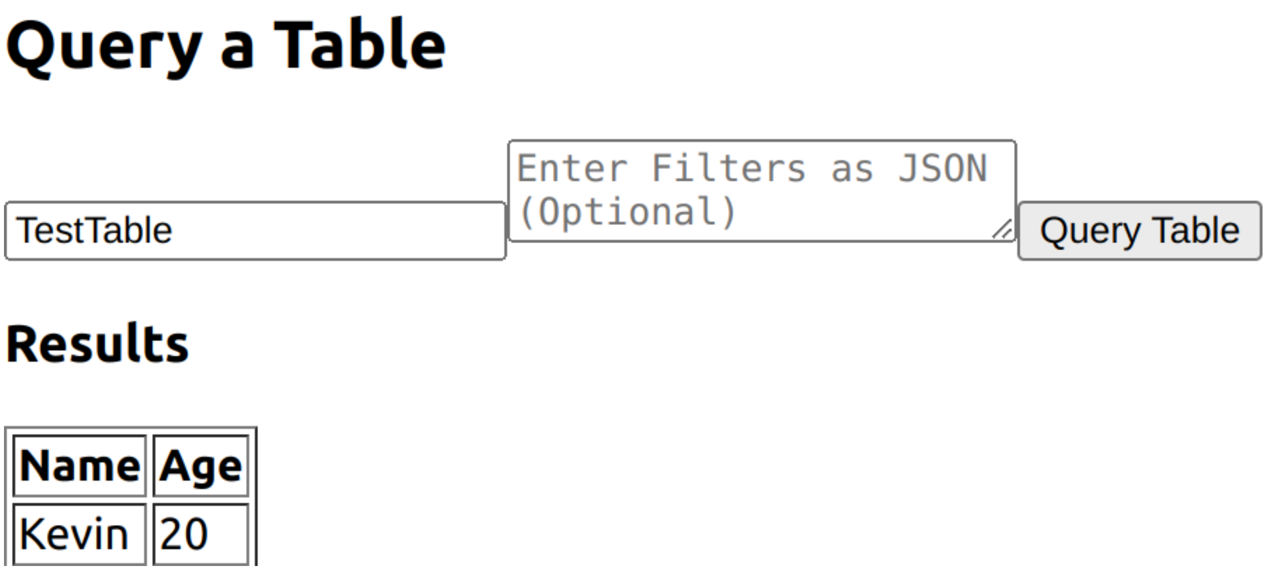


Populating table using "Define Columns for the Table" function to add columns, and adding data with "Add Rows" function:

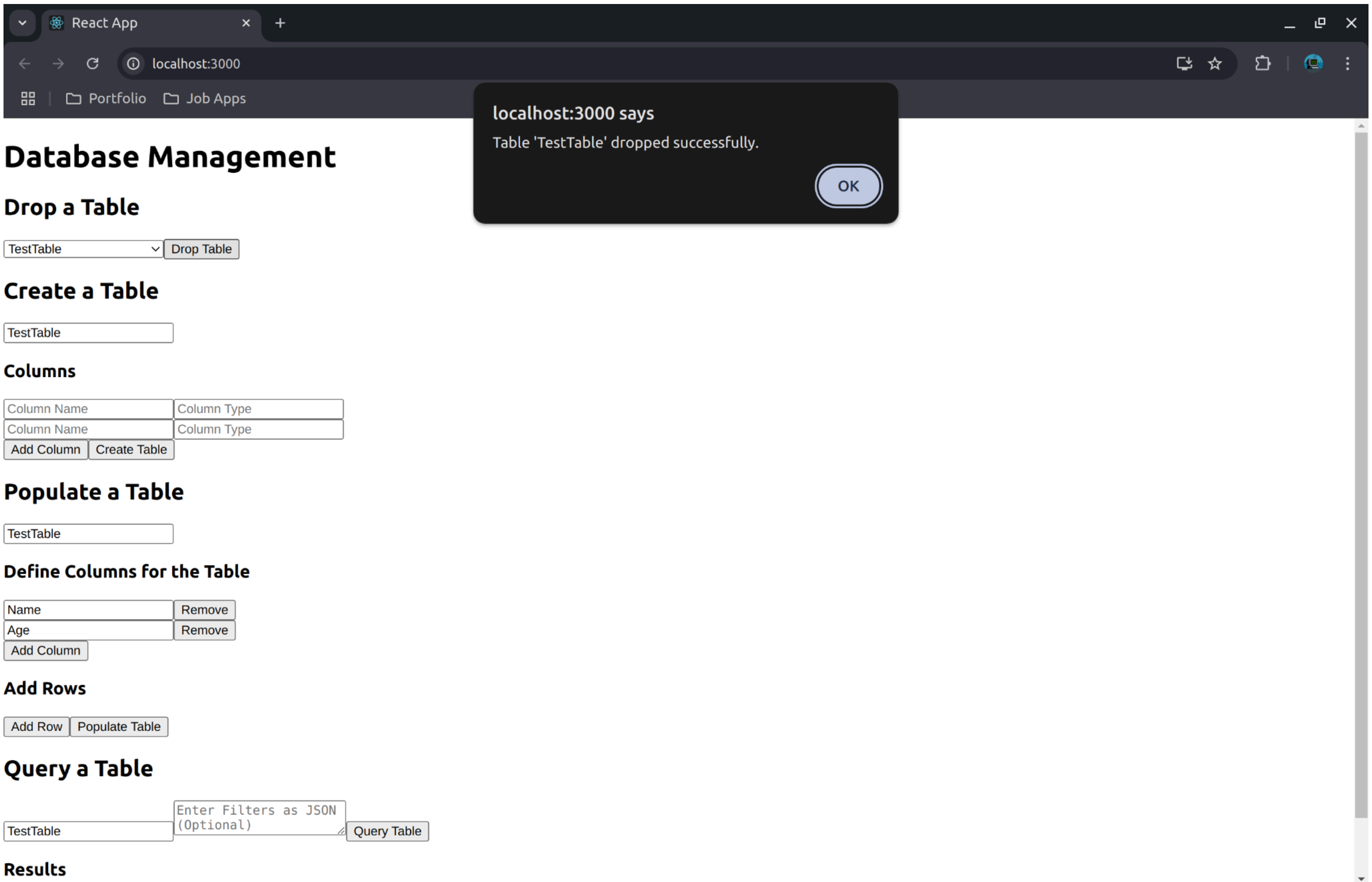


21

We can see the table and our results from the data, we can also query using json format such as { "name": "Kevin”, “age” : “20” }:



We can drop a table by scrolling down and finding our desired table using the “Drop a Table” function:



22

Installing GUI Instructions

---

1. Ensure that Node.js is installed (If not, instructions to install Node.js can be found on their official website)
2. Clone the GUI repository found on the Github.
3. Navigate to the frontend directory:
   cd /path/to/frontend
4. Install all required packages using npm:
   npm install
5. Navigate to the backend directory:
   cd /path/to/backend
6. Install all required packages:
   npm install
7. Configure Environment Variables (Create a .env file in the backend directory and fill in the connection details):
   DB_HOST=localhost
   DB_USER=root
   DB_PASS=your_password

DB_NAME=your_database_name

8. Start the backend server:

    node server.js

9. Navigate to the frontend directory:

    cd /path/to/frontend

10. Start the frontend server (The server will start on http://localhost:3000):

    npm start

11. Use the UI to:
    - Create tables.
    - Drop tables.
    - Populate tables.
    - Query and update records.