CSCI 5408

DATA MANAGEMENT AND WAREHOUSING

DDB Builder Sprint-2 Report

Group Members:

Chinmaya Garg	ch745692@dal.ca
Abhishek Kapoor	ab210637@dal.ca
Shivangkumar Kalpeshkumar Patel	sh587705@dal.ca
Nikulkumar Popatbhai Kukadiya	nk865270@dal.ca
Axata Darji	ax583820@dal.ca

GitLab Link: https://git.cs.dal.ca/abhishekk/csci5408_w24_ddb-builder-group-8

Table of Contents

Sprint-2 Task-5: Logical Diagram

Directions
In this phase, we're tasked with sketching a tabular structure based on the lecture discussion. Our aim is to ensure that these tables effectively highlight any instances of "Partial Dependency" or "Transitive Dependency". If we encounter partial dependencies, our strategy will be to address them by decomposing the tables into 2nd normal forms. This approach should enhance our understanding of the problem statement and enable us to create a more robust tabular structure.

Decomposing into Actionable Steps:

1. **Understanding the Requirements:** We thoroughly reviewed the requirements provided in the lecture, ensuring a clear understanding of what constitutes "Partial Dependency" and "Transitive Dependency".

2. **Drawing the Initial Tabular Structure:** We began by sketching out the initial tabular structure based on the lecture material, representing data entities and their relationships.

3. **Identifying Dependencies:** Each table was carefully examined to identify potential partial dependencies or transitive dependencies, which were documented for further analysis.

4. **Documenting Dependencies:** We documented any identified partial dependencies or transitive dependencies within the tabular structure, detailing the attributes causing the dependencies and their relationships.

5. **Decomposing Tables:** In instances where partial dependencies were found, we applied the process of decomposing tables into 2nd normal forms to rectify these issues and ensure data integrity.

6. **Updating Tabular Structure:** Following the decomposition process, the tabular structure was updated accordingly to reflect the removal of dependencies and adherence to 2nd normal form principles.

Partial Dependency in a Database

In a relational database, a partial dependency refers to a situation where a non-prime attribute is functionally dependent on only a part of the primary key, rather than the entire key. In other words, the value of a non-prime attribute is determined by only a subset of the primary key, leading to redundancy and potential anomalies in the database[1].

For example, consider a table with columns (A, B, C), where (A, B) is the primary key. If attribute C is functionally dependent on attribute B alone, and not on A, then there exists a partial dependency[2][3]. This means that changes in the value of B can affect the value of C, but changes in A do not have any impact on C.

Transitive Dependency in a Database

In a relational database, a transitive dependency occurs when the value of a non-prime attribute is functionally determined by another non-prime attribute, rather than by a candidate key. In simpler terms, it means that there is an indirect relationship between attributes in a table, where the value of one non-key attribute determines the value of another non-key attribute through a chain of dependencies[4].

For example, consider a table with attributes A, B, and C, where A is the primary key. If attribute B determines the value of attribute C, and attribute A determines the value of attribute B, then attribute C is transitively dependent on attribute A. This transitive dependency violates the rules of normalization, specifically the third normal form (3NF), which requires that all non-key attributes are dependent only on the primary key and not on other non-key attributes. To resolve transitive dependencies, the table needs to be decomposed into multiple tables, ensuring that each table represents a single logical entity with attributes that are functionally dependent only on the primary key.

Initial Logical Diagram

The logical diagram presented below illustrates the initial structure of the Hospital Management System's database. It provides a visual representation of the database tables and their attributes' dependency on the primary keys, serving as a blueprint for the system's data organization.

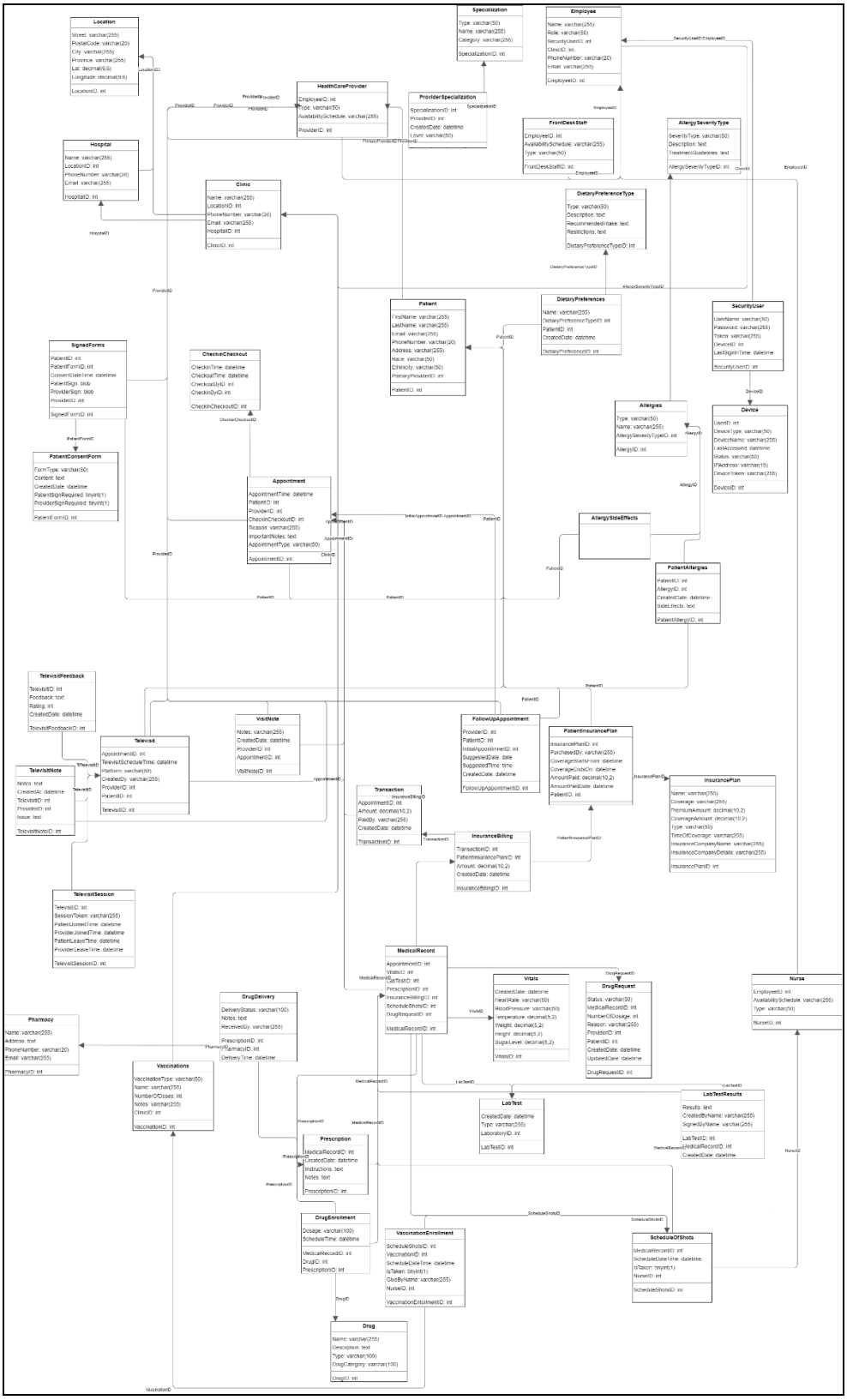


Fig 1: Initial Logical Diagram [5]

Documenting Dependencies and Decomposing Tables

1. **Hospital**
   - HospitalId is the primary key for the Hospital table.

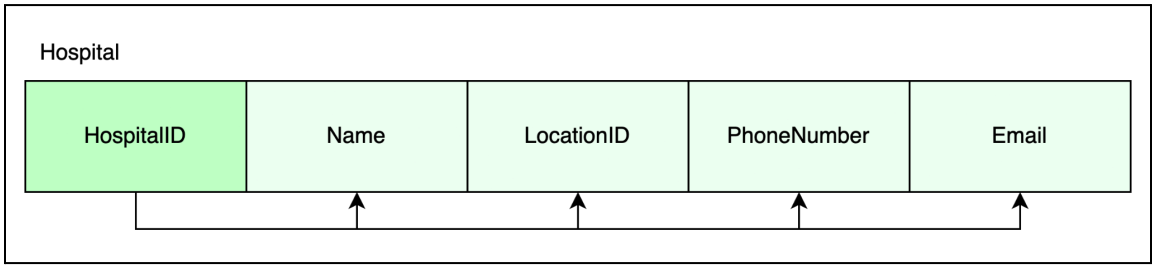


Fig 2: Logical table for Hospital

2. **Clinic**
   - ClinicID is the primary key for the Clinic table.

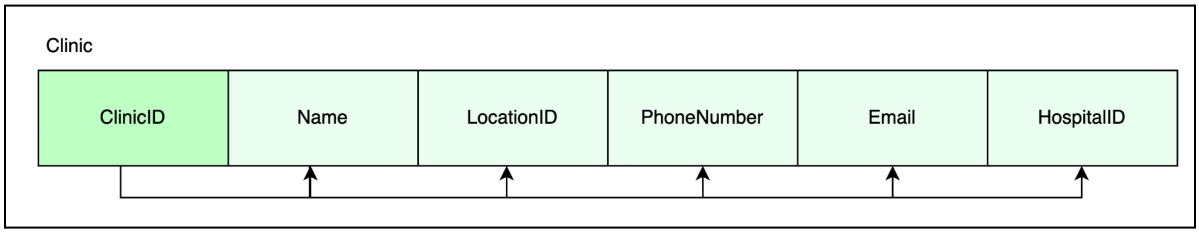


Fig 3: Logical table for Clinic

3. **Employee**
   - EmployeeID is the primary key for the Employee table.

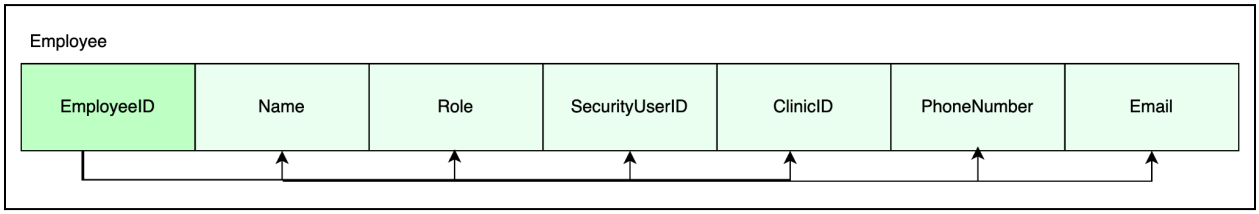


Fig 4: Logical table for Employee

4. **SecurityUser**
   - SecurityUserID is the primary key for the SecurityUser table.

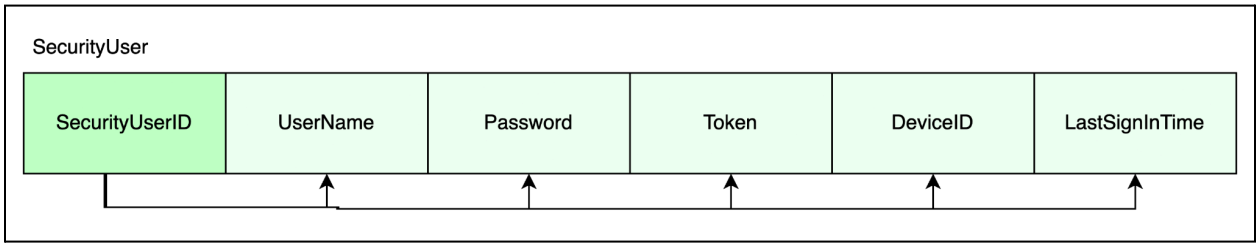


Fig 5: Logical table for SecurityUser

5. **HealthCareProvider**
    - ProviderID is the primary key for the HealthCareProvider table.

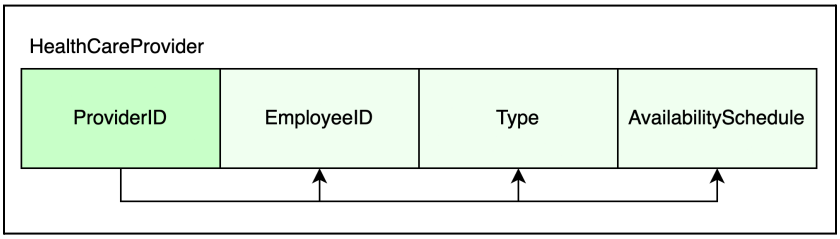


Fig 6: Logical table for HealthCareProvider

6. **Nurse**
    - NurseID is the primary key for the Nurse table.

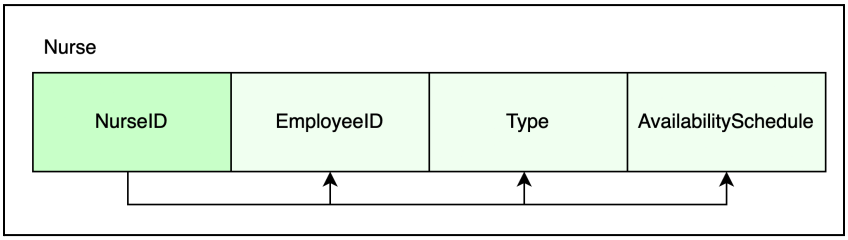


Fig 7: Logical table for Nurse

7. **FrontDeskStaff**
    - FrontDeskStaffID is the primary key for the FrontDeskStaff table.

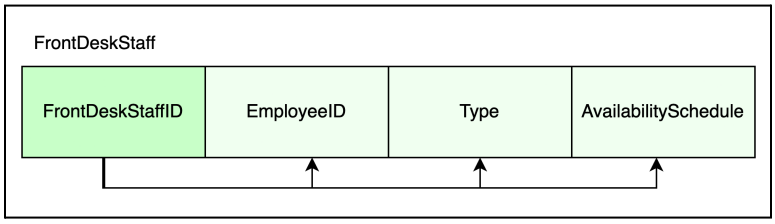


Fig 8: Logical table for FrontDeskStaff

8. **Patient**
    - PatientID is the primary key for the Patient table

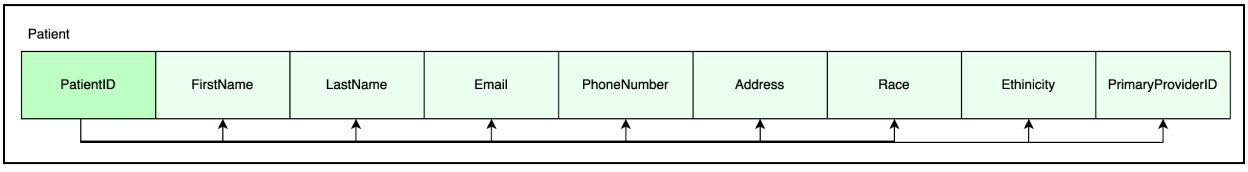


Fig 9: Logical table for Patient

9. Appointment

- AppointmentId is the primary key for the Appointment table.

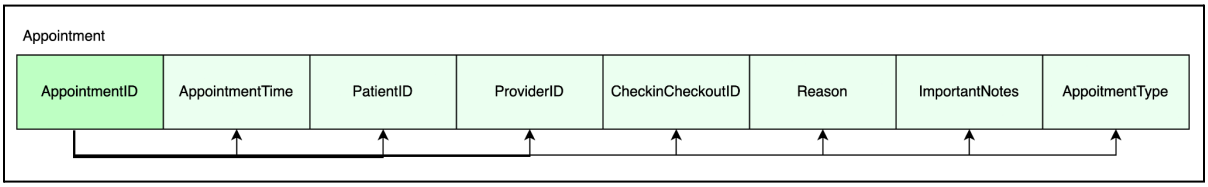


Fig 10: Logical table for Appointment

10. CheckinCheckout

- CheckinCheckoutId is the primary key for the CheckinCheckout table.

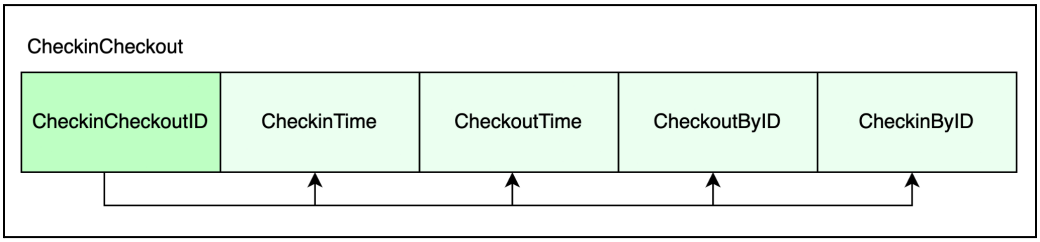


Fig 11: Logical table for CheckinCheckout

11. PatientInsurancePlan

- PatientInsurancePlanId is the primary key for the PatientInsurancePlan table.

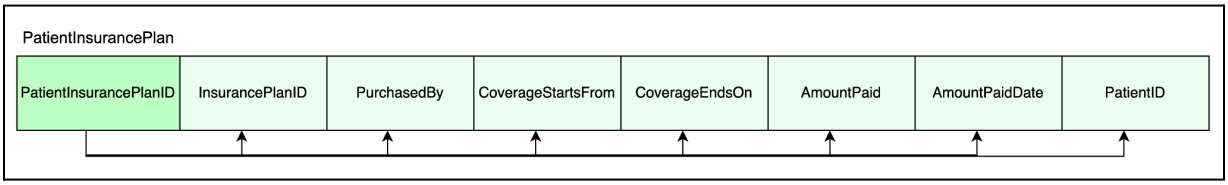


Fig 12: Logical table for PatientInsurancePlan

12. InsurancePlan

- InsurancePlanId is the primary key for the InsurancePlan table.

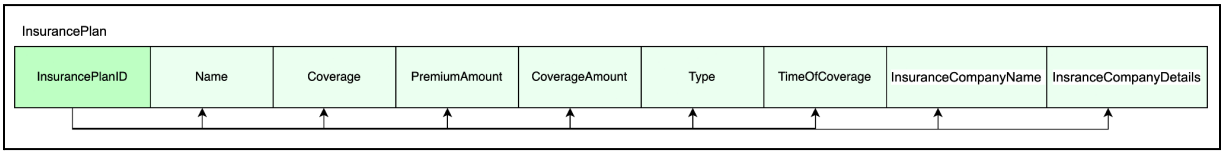


Fig 13: Logical table for InsurancePlan

13. InsuranceBilling

- InsuranceBillingId is the primary key for the InsuranceBilling table.

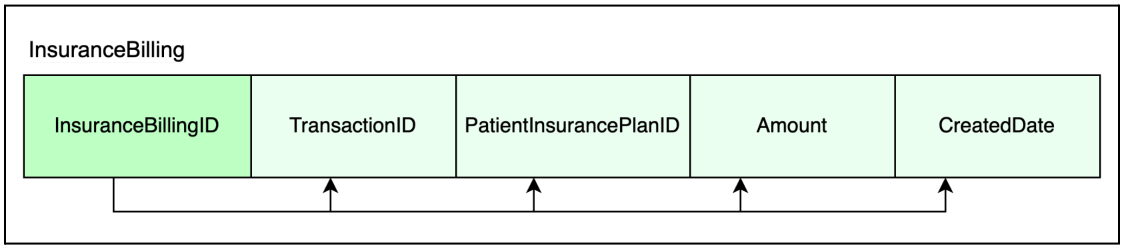


Fig 14: Logical table for InsuranceBilling

14. Transaction

- TransactionId is the primary key for the Transaction table.

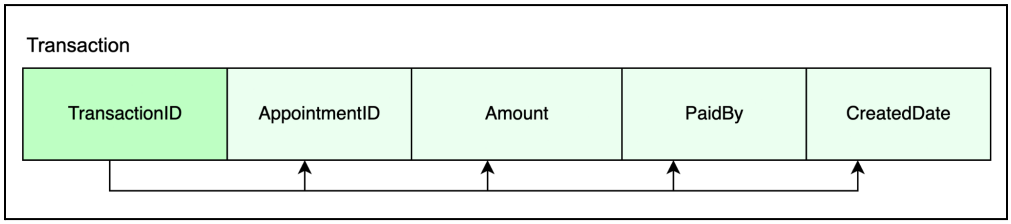


Fig 15: Logical table for Transaction

15. PatientConsentForm

- PatientConsentFormId is the primary key for the PatientConsentForm table

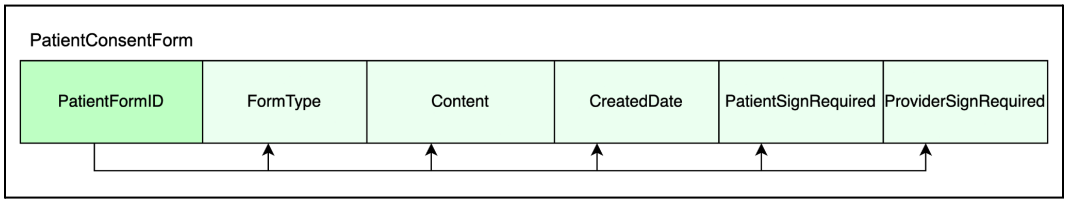


Fig 16: Logical table for PatientConsentForm

16. SignedForm

- SignedFormId is the primary key for the SignedForm table

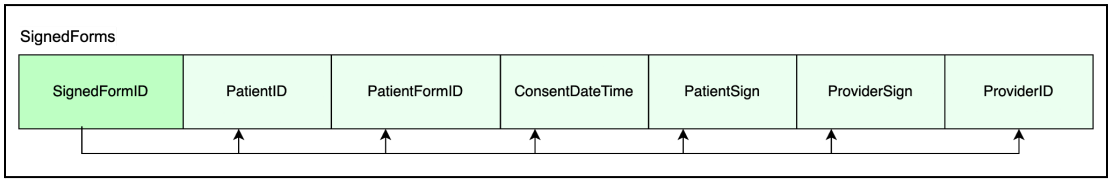


Fig 17: Logical table for SignedForm

17. Vaccinations

- VaccinationsId is the primary key for the Vaccinations table.

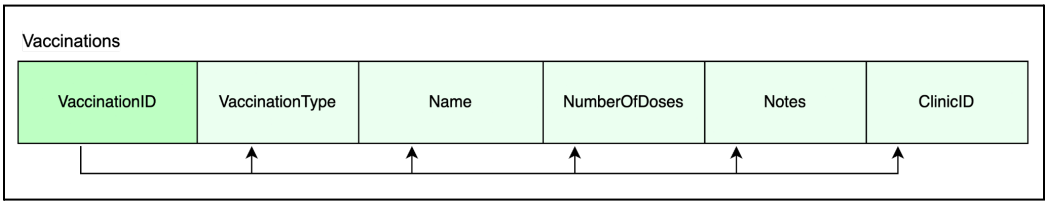


Fig 18: Logical table for Vaccinations

18. FollowUpAppointment

- FollowUpAppointmentId is the primary key for the FollowUpAppointment table.

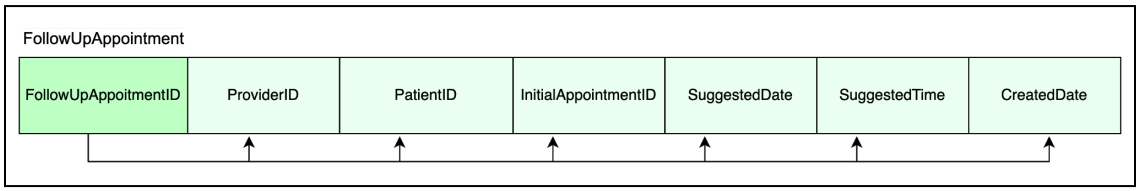


Fig 19: Logical table for FollowUpAppointment

19. VaccinationEnrollment

- VaccinationEnrollmentId is the primary key for the VaccinationEnrollment table.

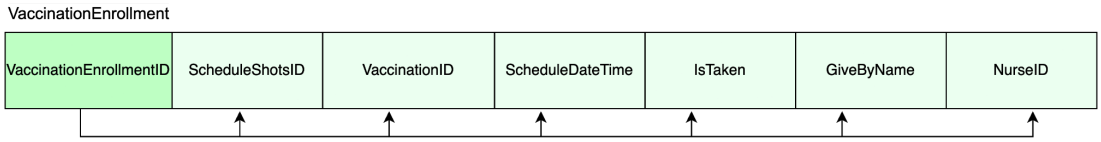


Fig 20: Logical table for VaccinationEnrollment

20. Allergies

- AllergiesId is the primary key for the Allergies table.

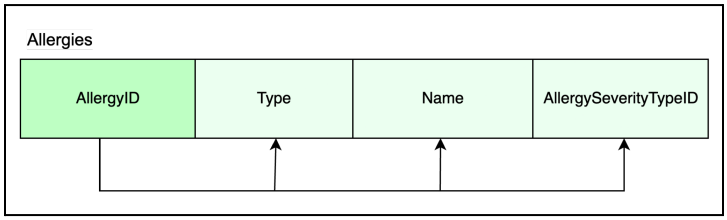


Fig 21: Logical table for Allergies

21. PatientAllergies

- PatientAllergiesId is the primary key for the PatientAllergies table.

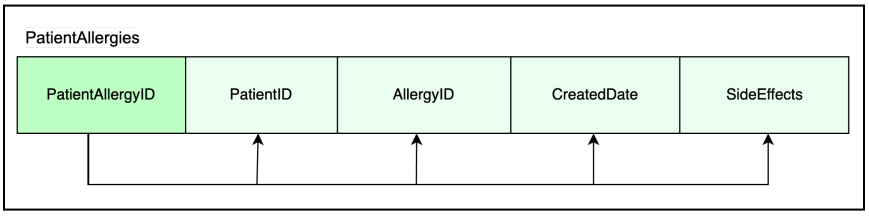


Fig 22: Logical table for PatientAllergies

22. Televisit

- TelevisitId is the primary key for the Televisit table.

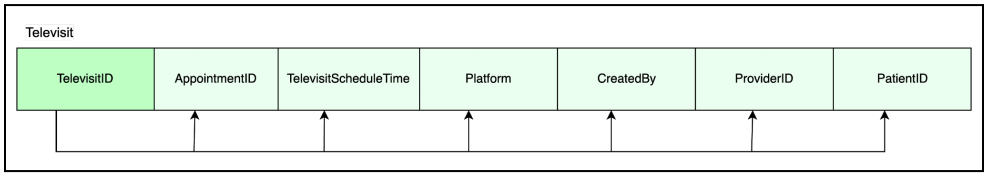


Fig 23: Logical table for Televisit

23. TelevisitSession

- TelevisitSessionId is the primary key for the TelevisitSession table.

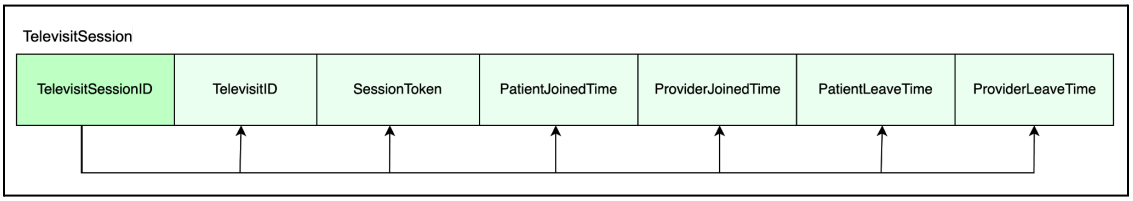


Fig 24: Logical table for TelevisitSession

24. TelevisitFeedback

- TelevisitFeedbackId is the primary key for the TelevisitFeedback table.

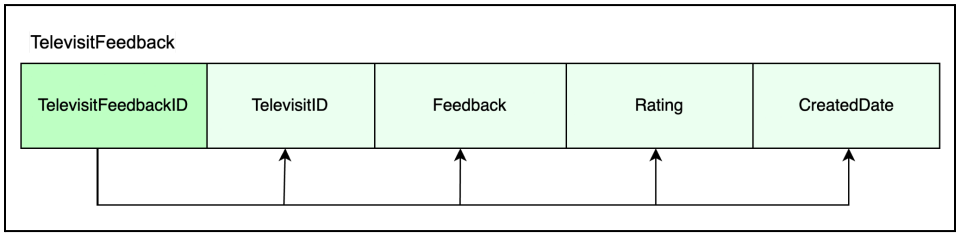


Fig 25: Logical table for TelevisitFeedback

25. TelevisitNotes

- TelevisitNotesId is the primary key for the TelevisitNotes table

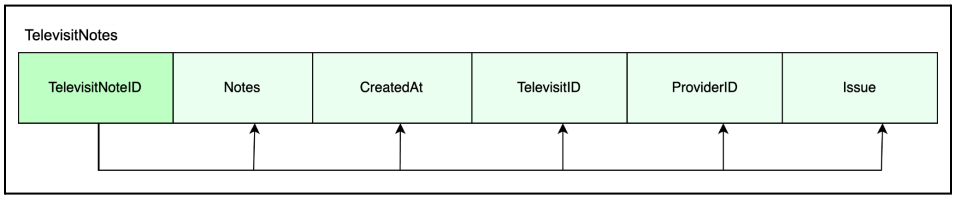


Fig 26: Logical table for TelevisitNotes

26. VisitNotes

- VisitNotesId is the primary key for the VisitNotes table.

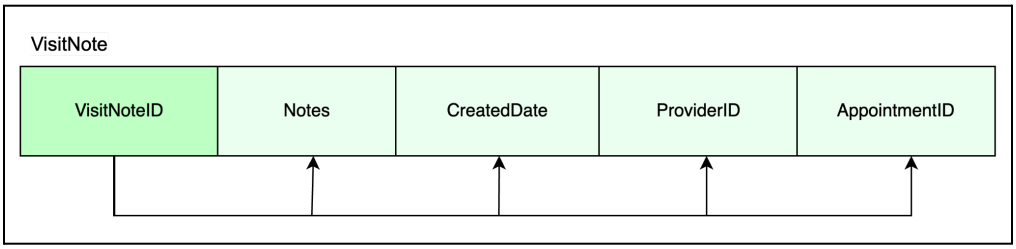


Fig 27: Logical table for VisitNotes

27. DietaryPreferences

- DietaryPreferencesId is the primary key for the DietaryPreferences table.

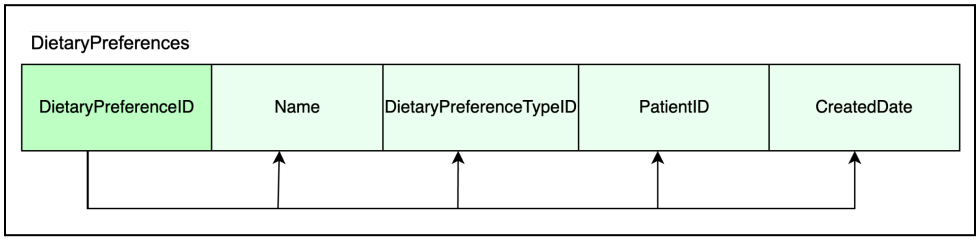


Fig 28: Logical table for DietaryPreferences

28. ProviderSpecialization

- ProviderSpecializationId is the primary key for the ProviderSpecialization table.

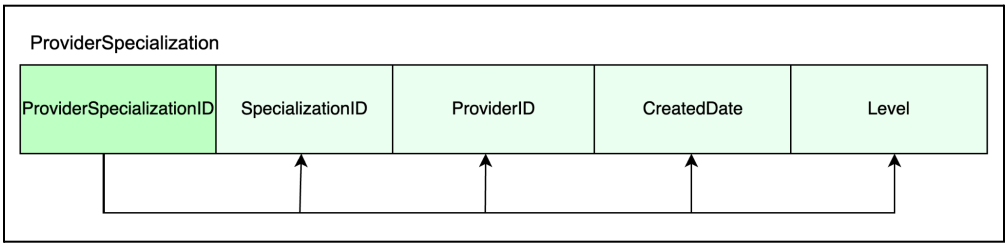


Fig 29: Logical table for ProviderSpecialization

29. Specialization

- SpecializationId is the primary key for the Specialization table.

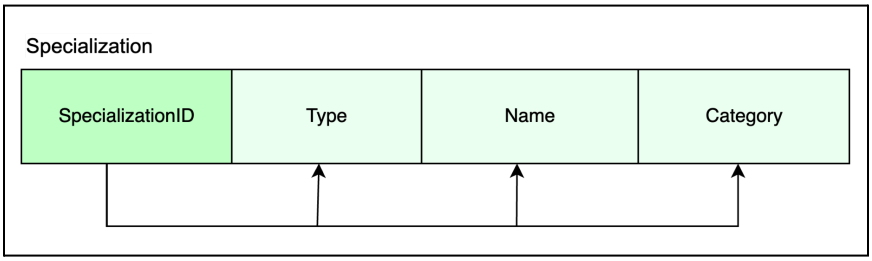


Fig 30: Logical table for Specialization

30. DrugRequest

- DrugRequestId is the primary key for the DrugRequest table.

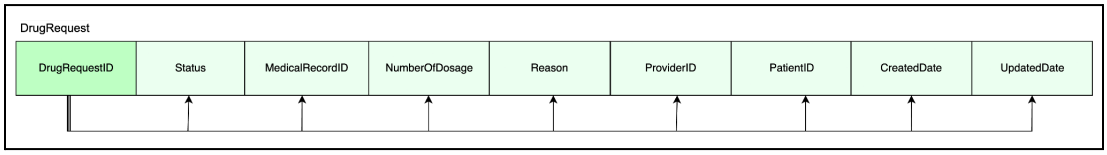


Fig 31: Logical table for DrugRequest

31. Location

- LocationId is the primary key for the Location table.

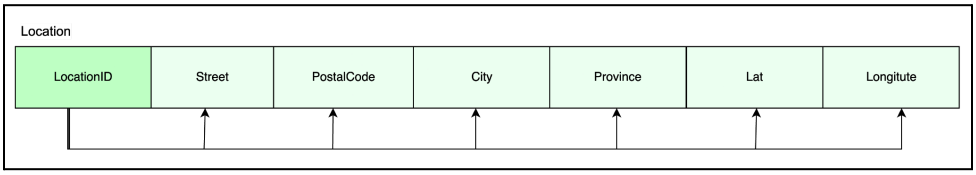


Fig 32: Logical table for Location

32. Device

- DeviceId is the primary key for the Device table.

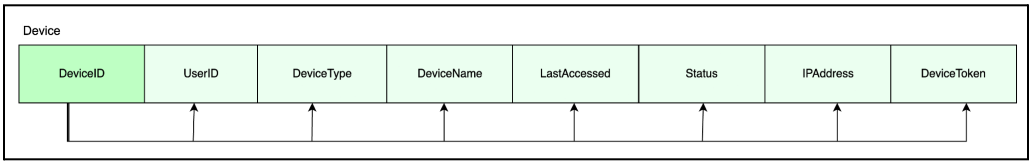


Fig 33: Logical table for Device

33. AllergySeverityType

- AllergySeverityTypeId is the primary key for the AllergySeverityType table.

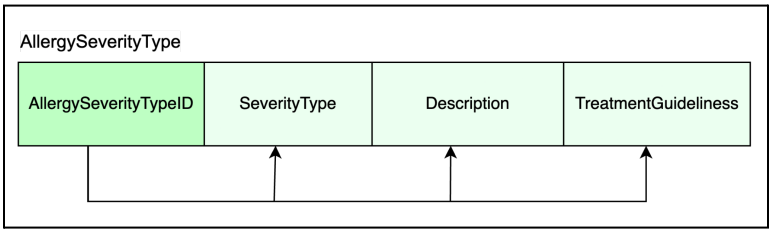


Fig 34: Logical table for AllergySeverityType

34. DietaryPreferenceType

- DietaryPreferenceTypeId is the primary key for the DietaryPreferenceType table.

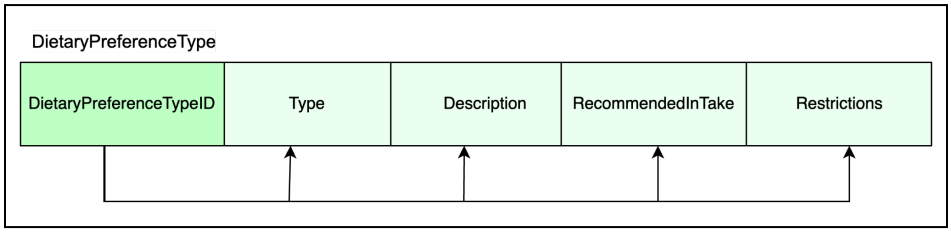


Fig 35: Logical table for DietaryPreferenceType

35. AllergySideEffects

- PatientID is the primary key for the AllergySideEffects table.

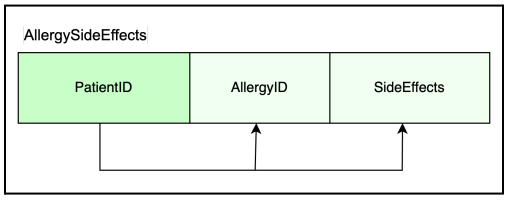


Fig 36: Logical table for AllergySideEffects

36. MedicalRecord

- MedicalRecordId is the primary key for the MedicalRecord table.

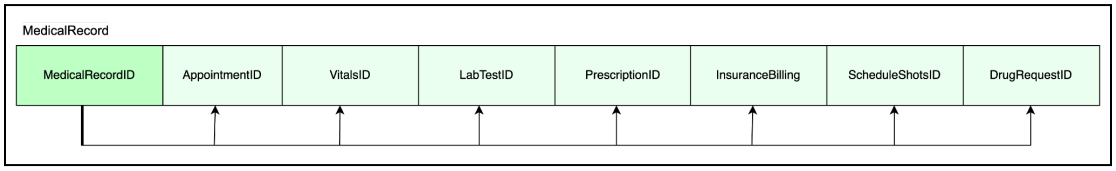


Fig 37: Logical table for MedicalRecord

37. Vitals

- VitalsId is the primary key for the Vitals table.

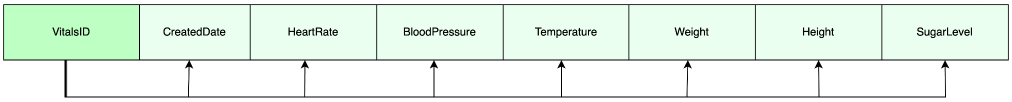


Fig 38: Logical table for Vitals

38. ScheduleOfShots

- ScheduleOfShotsId is the primary key for the ScheduleOfShots table.

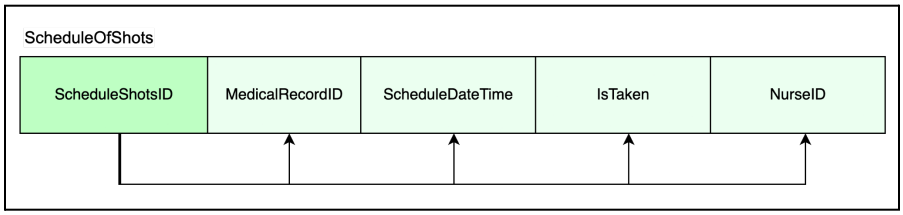


Fig 39: Logical table for ScheduleOfShots

39. Prescription

- PrescriptionId is the primary key for the Prescription table.

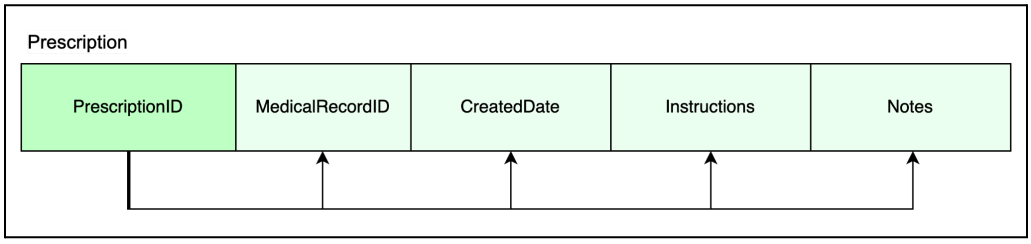


Fig 40: Logical table for Prescription

40. Drug

- DrugId is the primary key for the Drug table.

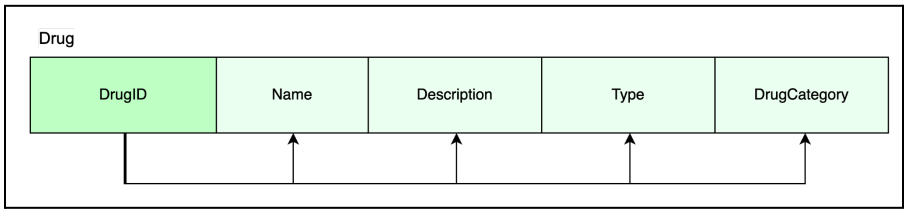


Fig 41: Logical table for Drug

41. DrugEnrollment

- DrugEnrollmentId is the primary key for the DrugEnrollment table.

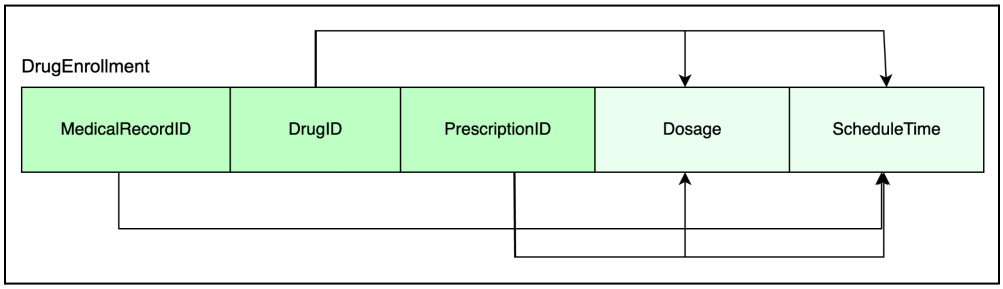


Fig 42: Logical table for DrugEnrollent

42. LabTest

- LabTestId is the primary key for the LabTest tab

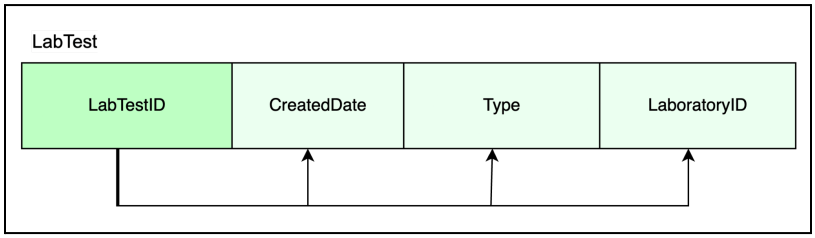


Fig 43: Logical table for LabTest

43. LabTestResults

- LabTestID+MedicalRecordID+CreatedDate is the primary key for the LabTestResults table.

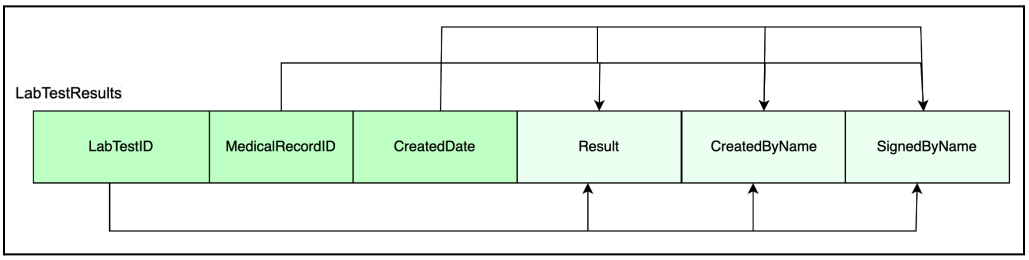


Fig 44: Logical table for LabTestResults

44. Pharmacy

- PharmacyID is the primary key for the Pharmacy tab

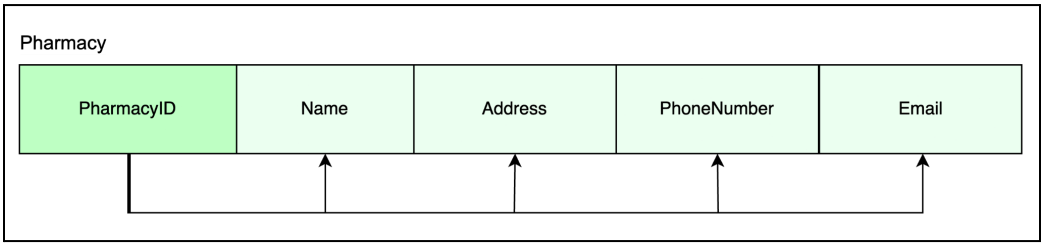


Fig 45: Logical table for Pharmacy

45. DrugDelivery

- PrescriptionID+PharmacyID+DeliveryTime is the primary key for the DrugDelivery table

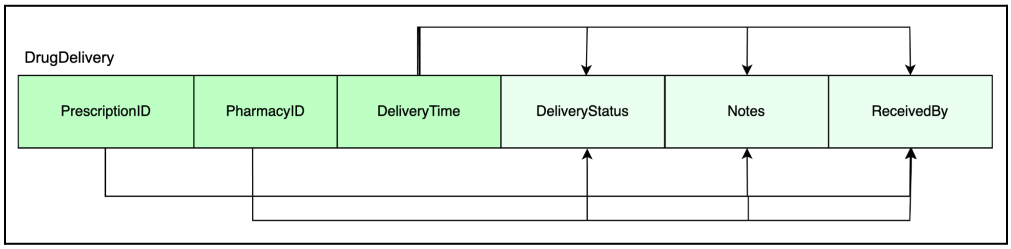


Fig 46: Logical table for DrugDelivery

Dependencies of the attributes on the above logical tables on the primary key:

As shown in all of the above tables we introduced our attributes in such a way that it will be directly dependent on all the primary key attributes and to do so we took below steps which reduced the complexity of the attribute dependencies and reduced the normalization issues in our database structure.

Steps taken to avoid dependencies the database model

To reduce dependencies and enhance the distributed database's architecture, the following steps were taken:

- **Identifying Relationships and Key Data Entities**: Initially, the top goals were identifying the key data entities, their connections, and the access patterns that influence database architecture. This solution allowed for greater flexibility and abstraction because it was implemented independent of any specific database technology.

- **Dissection into Modular Components**: To prevent relying too much on specific dependencies, the system was divided into more manageable, independent modules. For example, we separated entities into components linked to pharmacies, hospitals, vaccinations, and other topics.

- **Simplifying Relationships**: Relationship complexity was decreased to avoid the creation of transitive dependencies. Using ternary links or associative entities was favored.

After taking care of all of the above things our tables are normalized and below is the improved physical diagram which contains all the normalized tables. Also, for a clear vision of Fig 47 please refer to the reference link[6].

Updated physical diagram:

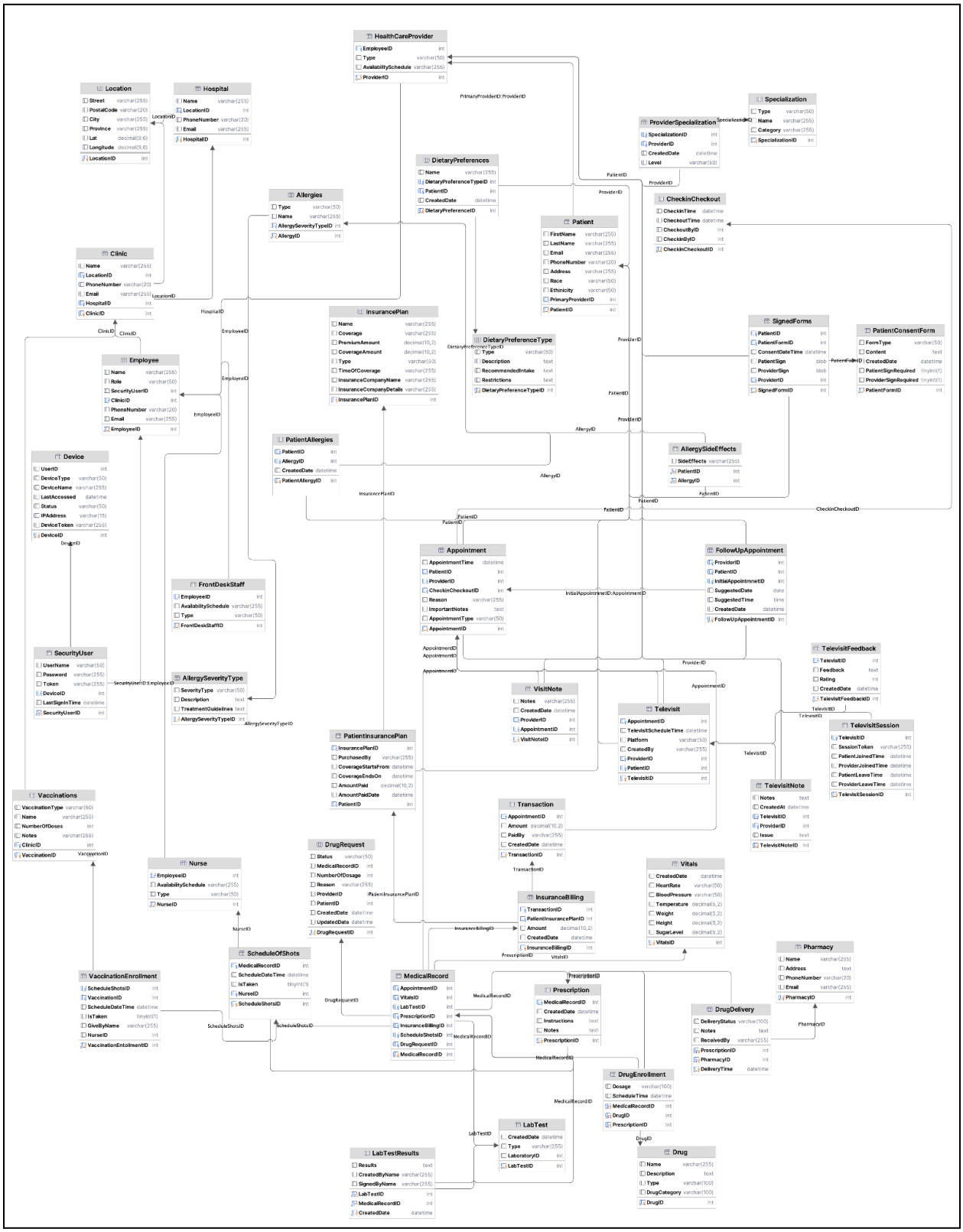


Fig 47: Updated physical diagram [6]

Sprint-2 Task-6: DDL

Directions

As we move into the physical phase of our project, we're tasked with translating the table structures into concrete Data Definition Language (DDL) statements using MySQL. This step involves comprehending the intricacies of the problem statement and crafting DDL statements that accurately reflect the logical structure of our data entities.

DDL Statements

```
CREATE TABLE Hospital (
    HospitalID INT PRIMARY KEY,
    Name VARCHAR(255),
    LocationID INT,
    PhoneNumber VARCHAR(20),
    Email VARCHAR(255)
);

CREATE TABLE Clinic (
    ClinicID INT PRIMARY KEY,
    Name VARCHAR(255),
    LocationID INT,
    PhoneNumber VARCHAR(20),
    Email VARCHAR(255),
    HospitalID INT
);

CREATE TABLE Employee (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(255),
    Role VARCHAR(50),
    SecurityUserID INT,
    ClinicID INT,
    PhoneNumber VARCHAR(20),
    Email VARCHAR(255)
);

CREATE TABLE SecurityUser (
    SecurityUserID INT PRIMARY KEY,
    UserName VARCHAR(50),
    Password VARCHAR(255),
    Token VARCHAR(255),
    DeviceID INT,
    LastSignInTime DATETIME
);

CREATE TABLE HealthCareProvider (
    ProviderID INT PRIMARY KEY,
    EmployeeID INT,
    Type VARCHAR(50),
    AvailabilitySchedule VARCHAR(255)
);
```

```sql
CREATE TABLE Nurse (
    NurseID INT PRIMARY KEY,
    EmployeeID INT,
    AvailabilitySchedule VARCHAR(255),
    Type VARCHAR(50)
);

CREATE TABLE FrontDeskStaff (
    FrontDeskStaffID INT PRIMARY KEY,
    EmployeeID INT,
    AvailabilitySchedule VARCHAR(255),
    Type VARCHAR(50)
);

CREATE TABLE Patient (
    PatientID INT PRIMARY KEY,
    FirstName VARCHAR(255),
    LastName VARCHAR(255),
    Email VARCHAR(255),
    PhoneNumber VARCHAR(20),
    Address VARCHAR(255),
    Race VARCHAR(50),
    Ethinicity VARCHAR(50),
    PrimaryProviderID INT
);

CREATE TABLE Appointment (
    AppointmentID INT PRIMARY KEY,
    AppointmentTime DATETIME,
    PatientID INT,
    ProviderID INT,
    CheckinCheckoutID INT,
    Reason VARCHAR(255),
    ImportantNotes TEXT,
    AppointmentType VARCHAR(50)
);

CREATE TABLE CheckinCheckout (
    CheckinCheckoutID INT PRIMARY KEY,
    CheckinTime DATETIME,
    CheckoutTime DATETIME,
    CheckoutByID INT,
    CheckinByID INT
);

CREATE TABLE PatientInsurancePlan (
    PatientInsurancePlanID INT PRIMARY KEY,
    InsurancePlanID INT,
    PurchasedBy VARCHAR(255),
    CoverageStartsFrom DATETIME,
    CoverageEndsOn DATETIME,
    AmountPaid DECIMAL(10, 2),
    AmountPaidDate DATETIME,
    PatientID INT
);
```

```sql
CREATE TABLE InsurancePlan (
    InsurancePlanID INT PRIMARY KEY,
    Name VARCHAR(255),
    Coverage VARCHAR(255),
    PremiumAmount DECIMAL(10, 2),
    CoverageAmount DECIMAL(10, 2),
    Type VARCHAR(50),
    TimeOfCoverage VARCHAR(255),
    InsuranceCompanyName VARCHAR(255),
    InsuranceCompanyDetails VARCHAR(255)
);

CREATE TABLE InsuranceBilling (
    InsuranceBillingID INT PRIMARY KEY,
    TransactionID INT,
    PatientInsurancePlanID INT,
    Amount DECIMAL(10, 2),
    CreatedDate DATETIME
);

CREATE TABLE Transaction (
    TransactionID INT PRIMARY KEY,
    AppointmentID INT,
    Amount DECIMAL(10, 2),
    PaidBy VARCHAR(255),
    CreatedDate DATETIME
);

CREATE TABLE PatientConsentForm (
    PatientFormID INT PRIMARY KEY,
    FormType VARCHAR(50),
    Content TEXT,
    CreatedDate DATETIME,
    PatientSignRequired BOOLEAN,
    ProviderSignRequired BOOLEAN
);

CREATE TABLE SignedForms (
    SignedFormID INT PRIMARY KEY,
    PatientID INT,
    PatientFormID INT,
    ConsentDateTime DATETIME,
    PatientSign BLOB, -- Assuming binary data for signature
    ProviderSign BLOB,
    ProviderID INT
);

CREATE TABLE Vaccinations (
    VaccinationID INT PRIMARY KEY,
    VaccinationType VARCHAR(50),
    Name VARCHAR(255),
    NumberOfDoses INT,
    Notes VARCHAR(255),
    ClinicID INT
);
```

```
CREATE TABLE FollowUpAppointment (
    FollowUpAppointmentID INT PRIMARY KEY,
    ProviderID INT,
    PatientID INT,
    InitialAppointmnetID INT,
    SuggestedDate DATE,
    SuggestedTime TIME,
    CreatedDate DATETIME
);

CREATE TABLE VaccinationEnrollment (
    VaccinationEntollmentID INT PRIMARY KEY,
    ScheduleShotsID INT,
    VaccinationID INT,
    ScheduleDateTime DATETIME,
    IsTaken BOOLEAN,
    GiveByName VARCHAR(255),
    NurseID INT
);

CREATE TABLE Allergies (
    AllergyID INT PRIMARY KEY,
    Type VARCHAR(50),
    Name VARCHAR(255),
    AllergySeverityTypeID INT
);

CREATE TABLE PatientAllergies (
    PatientAllergyID INT PRIMARY KEY,
    PatientID INT,
    AllergyID INT,
    CreatedDate DATETIME
);

CREATE TABLE Televisit (
    TelevisitID INT PRIMARY KEY,
    AppointmentID INT,
    TelevisitScheduleTime DATETIME,
    Platform VARCHAR(50),
    CreatedBy VARCHAR(255),
    ProviderID INT,
    PatientID INT
);

CREATE TABLE TelevisitSession (
    TelevisitSessionID INT PRIMARY KEY,
    TelevisitID INT,
    SessionToken VARCHAR(255),
    PatientJoinedTime DATETIME,
    ProviderJoinedTime DATETIME,
    PatientLeaveTime DATETIME,
    ProviderLeaveTime DATETIME
);
```

```
CREATE TABLE TelevisitFeedback (
    TelevisitFeedbackID INT PRIMARY KEY,
    TelevisitID INT,
    Feedback TEXT,
    Rating INT,
    CreatedDate DATETIME
);

CREATE TABLE TelevisitNote (
    TelevisitNoteID INT PRIMARY KEY,
    Notes TEXT,
    CreatedAt DATETIME,
    TelevisitID INT,
    ProviderID INT,
    Issue TEXT
);

CREATE TABLE VisitNote (
    VisitNoteID INT PRIMARY KEY,
    Notes VARCHAR(255),
    CreatedDate DATETIME,
    ProviderID INT,
    AppointmentID INT
);

CREATE TABLE DietaryPreferences (
    DietaryPreferenceID INT PRIMARY KEY,
    Name VARCHAR(255),
    DietaryPreferenceTypeID INT,
    PatientID INT,
    CreatedDate DATETIME
);

CREATE TABLE ProviderSpecialization (
    ProviderSpecializationID INT PRIMARY KEY,
    SpecializationID INT,

    ProviderID INT,
    CreatedDate DATETIME,
    Level VARCHAR(50)
);

CREATE TABLE Specialization (
    SpecializationID INT PRIMARY KEY,
    Type VARCHAR(50),
    Name VARCHAR(255),
    Category VARCHAR(255)
);

CREATE TABLE DrugRequest (
    DrugRequestID INT PRIMARY KEY,
    Status VARCHAR(50),
    MedicalRecordID INT,
    NumberOfDosage INT,
    Reason VARCHAR(255),
    ProviderID INT,
    PatientID INT,
```

```
    CreatedDate DATETIME,
    UpdatedDate DATETIME
);

CREATE TABLE Location (
    LocationID INT PRIMARY KEY,
    Street VARCHAR(255),
    PostalCode VARCHAR(20),
    City VARCHAR(255),
    Province VARCHAR(255),
    Lat DECIMAL(9, 6),
    Longitude DECIMAL(9, 6)
);

CREATE TABLE Device (
    DeviceID INT PRIMARY KEY,
    UserID INT,
    DeviceType VARCHAR(50),
    DeviceName VARCHAR(255),
    LastAccessed DATETIME,
    Status VARCHAR(50),
    IPAddress VARCHAR(15),
    DeviceToken VARCHAR(255)
);

CREATE TABLE AllergySeverityType (
    AllergySeverityTypeID INT PRIMARY KEY,
    SeverityType VARCHAR(50),
    Description TEXT,
    TreatmentGuidelines TEXT
);

CREATE TABLE DietaryPreferenceType (
    DietaryPreferenceTypeID INT PRIMARY KEY,
    Type VARCHAR(50),
    Description TEXT,
    RecommendedIntake TEXT,
    Restrictions TEXT
);

CREATE TABLE AllergySideEffects (
    PatientID INT,
    AllergyID INT,
    SideEffects VARCHAR(255),
    PRIMARY KEY (PatientID, AllergyID)
);

CREATE TABLE MedicalRecord (
    MedicalRecordID INT PRIMARY KEY,
    AppointmentID INT,
    VitalsID INT,
    LabTestID INT,
    PrescriptionID INT,
    InsuranceBillingID INT,
    ScheduleShotsID INT,
    DrugRequestID INT
);
```

```
CREATE TABLE Vitals (
    VitalsID INT PRIMARY KEY,
    CreatedDate DATETIME,
    HeartRate VARCHAR(50),
    BloodPressure VARCHAR(50),
    Temperature DECIMAL(5, 2),
    Weight DECIMAL(5, 2),
    Height DECIMAL(5, 2),
    SugarLevel DECIMAL(5, 2)
);

CREATE TABLE ScheduleOfShots (
    ScheduleShotsID INT PRIMARY KEY,
    MedicalRecordID INT,
    ScheduleDateTime DATETIME,
    IsTaken BOOLEAN,
    NurseID INT
);

CREATE TABLE Prescription (
PrescriptionID INT PRIMARY KEY,
    MedicalRecordID INT,
    CreatedDate DATETIME,
    Instructions TEXT,
    Notes TEXT
);

CREATE TABLE Drug (
    DrugID INT PRIMARY KEY,
    Name VARCHAR(255),
    Description TEXT,
    Type VARCHAR(100),
    DrugCategory VARCHAR(100)
);

CREATE TABLE DrugEnrollment (
    MedicalRecordID INT,
    DrugID INT,
    PrescriptionID INT,
    Dosage VARCHAR(100),
    ScheduleTime DATETIME,
    PRIMARY KEY (MedicalRecordID, DrugID, PrescriptionID)
);

CREATE TABLE LabTest (
    LabTestID INT PRIMARY KEY,
    CreatedDate DATETIME,
    Type VARCHAR(255));

CREATE TABLE LabTestResults (
    LabTestID INT,
    MedicalRecordID INT,
    Results TEXT,
    CreatedDate DATETIME,
    CreatedByName VARCHAR(255),
    SignedByName VARCHAR(255),
    PRIMARY KEY (LabTestID, MedicalRecordID, CreatedDate)
```

```
);

CREATE TABLE Pharmacy (
    PharmacyID INT PRIMARY KEY,
    Name VARCHAR(255),
    Address TEXT,
    PhoneNumber VARCHAR(20),
    Email VARCHAR(255)
);

CREATE TABLE DrugDelivery (
    PrescriptionID INT,
    PharmacyID INT,
    DeliveryTime DATETIME,
    DeliveryStatus VARCHAR(100),
    Notes TEXT,
    ReceivedBy VARCHAR(255),
    PRIMARY KEY (PrescriptionID, PharmacyID, DeliveryTime)
);
```

Below are the Alter table queries as all the tables are not available during giving the references to other tables while executing the DDL commands we introduced our foreign key references later on.

```
ALTER TABLE Clinic
ADD FOREIGN KEY (HospitalID) REFERENCES Hospital(HospitalID);

ALTER TABLE Employee
ADD FOREIGN KEY (ClinicID) REFERENCES Clinic(ClinicID);

ALTER TABLE SecurityUser
ADD FOREIGN KEY (SecurityUserID) REFERENCES Employee(EmployeeID);

ALTER TABLE HealthCareProvider
ADD FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID);

ALTER TABLE Patient
ADD FOREIGN KEY (PrimaryProviderID) REFERENCES HealthCareProvider(ProviderID);

ALTER TABLE Employee
ADD FOREIGN KEY (ClinicID) REFERENCES Clinic(ClinicID);

ALTER TABLE HealthCareProvider
ADD FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID);

ALTER TABLE Nurse
ADD FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID);

ALTER TABLE FrontDeskStaff
ADD FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID);

ALTER TABLE PatientInsurancePlan
ADD FOREIGN KEY (PatientID) REFERENCES Patient(PatientID),
ADD FOREIGN KEY (InsurancePlanID) REFERENCES InsurancePlan(InsurancePlanID);
```

```
ALTER TABLE InsuranceBilling
ADD FOREIGN KEY (TransactionID) REFERENCES Transaction(TransactionID),
ADD FOREIGN KEY (PatientInsurancePlanID) REFERENCES
PatientInsurancePlan(PatientInsurancePlanID);

ALTER TABLE Transaction
ADD FOREIGN KEY (AppointmentID) REFERENCES Appointment(AppointmentID);

ALTER TABLE VaccinationEnrollment
ADD FOREIGN KEY (ScheduleShotsID) REFERENCES ScheduleOfShots(ScheduleShotsID),
ADD FOREIGN KEY (VaccinationID) REFERENCES Vaccinations(VaccinationID);

ALTER TABLE FollowUpAppointment
ADD FOREIGN KEY (ProviderID) REFERENCES HealthCareProvider(ProviderID),
ADD FOREIGN KEY (PatientID) REFERENCES Patient(PatientID),
ADD FOREIGN KEY (InitialAppointmnetID) REFERENCES Appointment(AppointmentID);

ALTER TABLE VisitNote
ADD FOREIGN KEY (ProviderID) REFERENCES HealthCareProvider(ProviderID),
ADD FOREIGN KEY (AppointmentID) REFERENCES Appointment(AppointmentID);

ALTER TABLE ProviderSpecialization
ADD FOREIGN KEY (ProviderID) REFERENCES HealthCareProvider(ProviderID),
ADD FOREIGN KEY (SpecializationID) REFERENCES Specialization(SpecializationID);

ALTER TABLE AllergySideEffects
ADD FOREIGN KEY (PatientID) REFERENCES Patient(PatientID),
ADD FOREIGN KEY (AllergyID) REFERENCES Allergies(AllergyID);

ALTER TABLE Clinic
ADD FOREIGN KEY (LocationID) REFERENCES Location(LocationID);

ALTER TABLE Hospital
ADD FOREIGN KEY (LocationID) REFERENCES Location(LocationID);

ALTER TABLE Allergies
ADD FOREIGN KEY (AllergySeverityTypeID) REFERENCES
AllergySeverityType(AllergySeverityTypeID);

ALTER TABLE DietaryPreferences
ADD FOREIGN KEY (PatientID) REFERENCES Patient(PatientID),
ADD FOREIGN KEY (DietaryPreferenceTypeID) REFERENCES
DietaryPreferenceType(DietaryPreferenceTypeID);

ALTER TABLE TelevisitNote
ADD FOREIGN KEY (TelevisitID) REFERENCES Televisit(TelevisitID),
ADD FOREIGN KEY (ProviderID) REFERENCES HealthCareProvider(ProviderID);

ALTER TABLE TelevisitFeedback
ADD FOREIGN KEY (TelevisitID) REFERENCES Televisit(TelevisitID);

ALTER TABLE TelevisitSession
ADD FOREIGN KEY (TelevisitID) REFERENCES Televisit(TelevisitID);
```

```
ALTER TABLE Televisit
ADD FOREIGN KEY (AppointmentID) REFERENCES Appointment(AppointmentID),
ADD FOREIGN KEY (ProviderID) REFERENCES HealthCareProvider(ProviderID),
ADD FOREIGN KEY (PatientID) REFERENCES Patient(PatientID);

ALTER TABLE PatientAllergies
ADD FOREIGN KEY (PatientID) REFERENCES Patient(PatientID),
ADD FOREIGN KEY (AllergyID) REFERENCES Allergies(AllergyID);

ALTER TABLE SignedForms
ADD FOREIGN KEY (PatientID) REFERENCES Patient(PatientID),
ADD FOREIGN KEY (PatientFormID) REFERENCES PatientConsentForm(PatientFormID),
ADD FOREIGN KEY (ProviderID) REFERENCES HealthCareProvider(ProviderID);

ALTER TABLE Appointment
ADD FOREIGN KEY (PatientID) REFERENCES Patient(PatientID),
ADD FOREIGN KEY (ProviderID) REFERENCES HealthCareProvider(ProviderID),
ADD FOREIGN KEY (CheckinCheckoutID) REFERENCES
CheckinCheckout(CheckinCheckoutID);

ALTER TABLE MedicalRecord
ADD FOREIGN KEY (AppointmentID) REFERENCES Appointment(AppointmentID),
ADD FOREIGN KEY (VitalsID) REFERENCES Vitals(VitalsID),
ADD FOREIGN KEY (LabTestID) REFERENCES LabTest(LabTestID),
ADD FOREIGN KEY (PrescriptionID) REFERENCES Prescription(PrescriptionID)
ADD FOREIGN KEY (InsuranceBillingID) REFERENCES
InsuranceBilling(InsuranceBillingID),
ADD FOREIGN KEY (ScheduleShotsID) REFERENCES ScheduleOfShots(ScheduleShotsID),
ADD FOREIGN KEY (DrugRequestID) REFERENCES DrugRequest(DrugRequestID);

ALTER TABLE Vaccinations
ADD FOREIGN KEY (ClinicID) REFERENCES Clinic(ClinicID);

ALTER TABLE ScheduleOfShots
ADD FOREIGN KEY (MedicalRecordID) REFERENCES MedicalRecord(MedicalRecordID),
ADD FOREIGN KEY (NurseID) REFERENCES Nurse(NurseID);

ALTER TABLE Prescription
ADD FOREIGN KEY (MedicalRecordID) REFERENCES MedicalRecord(MedicalRecordID);

ALTER TABLE DrugEnrollment
ADD FOREIGN KEY (MedicalRecordID) REFERENCES MedicalRecord(MedicalRecordID),
ADD FOREIGN KEY (DrugID) REFERENCES Drug(DrugID),
ADD FOREIGN KEY (PrescriptionID) REFERENCES Prescription(PrescriptionID);

ALTER TABLE LabTestResults
ADD FOREIGN KEY (LabTestID) REFERENCES LabTest(LabTestID),
ADD FOREIGN KEY (MedicalRecordID) REFERENCES MedicalRecord(MedicalRecordID);

ALTER TABLE DrugDelivery
ADD FOREIGN KEY (PrescriptionID) REFERENCES Prescription(PrescriptionID),
ADD FOREIGN KEY (PharmacyID) REFERENCES Pharmacy(PharmacyID);

ALTER TABLE SecurityUser
ADD FOREIGN KEY (DeviceID) REFERENCES Device(DeviceID);
```

Sprint-2 Task-7: Distributed Database

Fragmentation

- Fragmentation refers to the process of splitting data and database objects into smaller units. These smaller units are called fragments which are distributed across multiple nodes or servers in a network.
- Fragmentation is done to improve performance, increase availability, fault tolerance, and enhance scalability in distributed environments.

Fragmentation Types

- Database-level fragmentation and table-level fragmentation refer to different granularities at which data can be fragmented within a database system.
- Database-level fragmentation operates at the level of entire databases, dividing them into smaller units, while table-level fragmentation operates at the level of individual tables within a database, dividing them into smaller fragments based on specific criteria.
- Furthermore, both database-level and table-level fragmentation can occur in two main ways: horizontal and vertical partitioning. In horizontal fragmentation, rows of a table are divided based on a condition or a criterion. In vertical fragmentation, attributes or columns of a table are divided among multiple nodes or servers.

Choosing Fragmentation Type

- In our scenario, we don't have any data available in any of the tables so it is quite difficult to make a decision on which fragmentation to choose. Even if there are no tables or data present, a database still has a schema that defines its structure.
- Horizontal fragmentation at the database level allows us to divide the schema into smaller, manageable fragments, each representing functional modules or subsystems pertinent to specific operations.
- For instance, in our database, where we store information about various clinics, we opt for fragmentation based on clinic locations. This approach ensures that data related to clinics in distinct geographical areas are organized into separate fragments, facilitating efficient data management and retrieval processes.
- This approach ensures that even without actual data in tables, the database architecture is optimized for scalability, performance, and modularity.

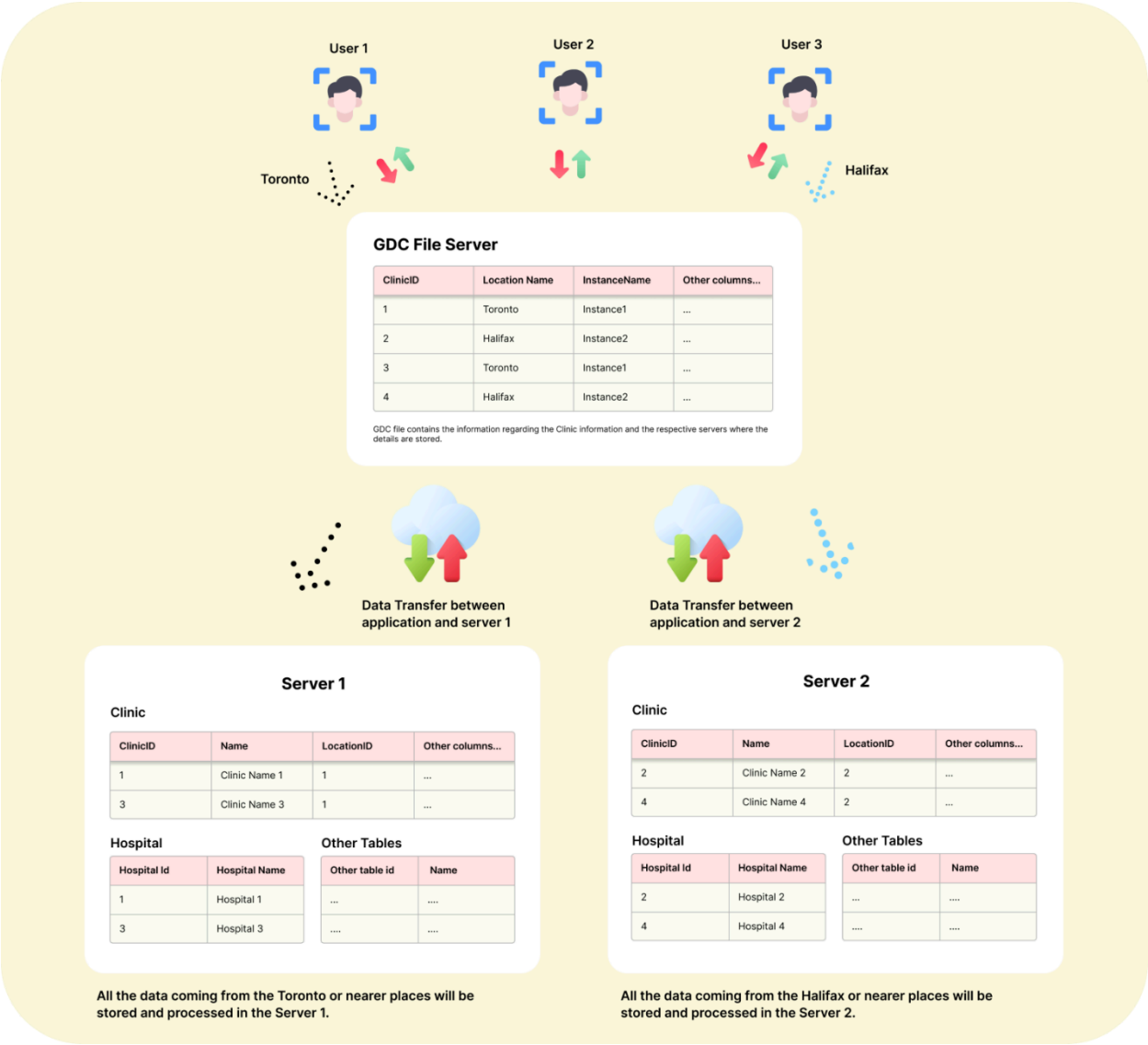


Fig 48: Horizontal fragmentation implementation using GDC with multiple server instances.

Methodology

- In our database, we introduced the **horizontal fragmentation** strategy which divides a table horizontally by selecting a subset of rows by values of one or more fields.
- After partition, these data fragments are assigned to different servers of a Distributed Database System.
- As per the GDC, our tables will be selected and will fetch the records from respected instances or servers of the distributed database system.

- We have **placed the same tables on both the server instances and our GDC will decide how the data will be stored on which tables**.
- Here we assume that all our queries will be passed from the GDC and based on the server selection of the GDC the queries will be processed.

- GDC will decide the server selection based on the requested locations. As shown in the above diagram, our GDC and server selection process will be implemented.

Data Entry Point

- Firstly, all the requests will come to the GDC.
- Based on the location mapping inside the file of the GDC where all the hospitals, clinics, and servers will be placed.
- GDC first evaluates the query and decides whether the query execution will be performed on server 1 or server 2 and the result will be returned to the origin user.

Sample Scenario

Let's take one example of how fragmentation is implemented.

- As shown in the diagram figure above (Fig 48), all our users are requesting the data from our GDC.
- Our GDC file contains details regarding all the hospitals and clinics available on both server 1 and server 2 with which server contains clinic or hospital data.
- Now GDC will read the query from the users. Parse that query and based on the mapping it will get the details regarding which server instance to choose.
- Assume our user wanted the query something like select clinic details for the clinic id "1" now when a request comes to GDC.
    - GDC search inside the file where this clinic id "1" is mapped with which server instance.
    - Once the GDC returns the server instance we will send the query to that server and once the query execution is completed then the fetched records will be returned to the user.

From the diagram take one example as our GDC file contains the details such as clinic IDs, locations, and server details. This shows that ClinicID "1" and "3" will be stored in "Instance1" while ClinicID "2" and "4" will be stored in the "Instance2". So, when requests come from a particular location such as Toronto will be redirected to "Instance1" as per our mapping file and the requests from Halifax will be redirected to "Instance2".

As given in the diagram above, User 1 is from the Toronto location which requested the clinic details. With the dotted black arrow direction shown in the diagram, the request flow will be from the User to the location where our application code and GDC file are stored. Then the GDC file got the server 1 based on the requested user's location which is Toronto. So, GDC chose server 1 and forwarded the query to server 1.

For users from Halifax as shown in the diagram, User 3 requested the details of the clinic from Halifax. Hence our mapping file stored the Halifax clinic details in server 2 the request was redirected to server 2.

Horizontal fragmentation explanation using the multiple server instance and the Java Query execution based on the data of the GDC.

In our GDC file the content will be stored as comma-separated details of the clinic, location, and server details such as URL and user details to access the data.

```
GDC File content which stores the information of the clinic, hospital, location and the server instance details
HospitalID,ClinicID,LocationID,Latitude,Longitude,City,State,Country,PINCode,InstanceName,InstanceStatus,ServerURL,Port,Username
1,1,1,43.65107,-79.347015,Kitchener,Ontario,Canada,123456,Instance1,Active,jdbc:mysql://34.41.125.106:3306,3306,root
1,3,1,43.65107,-79.347015,Kitchener,Ontario,Canada,123456,Instance1,Active,jdbc:mysql://34.41.125.106:3306,3306,root
2,2,2,44.6488,-63.5752,Halifax,Nova Scotia,Canada,123456,Instance2,Active,jdbc:mysql://34.106.240.4:3306,3306,root
2,4,2,44.6488,-63.5752,Halifax,Nova Scotia,Canada,123456,Instance2,Active,jdbc:mysql://34.106.240.4:3306,3306,root
```

Fig 49: GDC File content stored on Server1.

As per the above mapping, our clinics are divided based on the locations such as few clinics are available on server1 and few on server2. Below are the details available on both the different servers. Also, please take note that we use our **ClinicID** as a **Partition Key** for more simplification of the data separation rather than introducing a new key. Because it is a more suitable context rather than a random shared key. All the data related to patient and health services are directly and indirectly binded with the clinic hence we used ClinicID as our Partition key. The below figure shows all the available clinics.

```
Clinics from the server 1
ClinicID    Name    LocationID  PhoneNumber Email   HospitalID
1   THS Clinic Kitchener    1   1234567890  clinic1@example.com 1
3   THS Clinic Ontario  1   1234567890  clinic3@example.com 1
Clinics from the server 2
ClinicID    Name    LocationID  PhoneNumber Email   HospitalID
2   Halifax Clinic  2   1234567890  clinic2@example.com 2
4   Halifac Scotia bank Clinic  2   1234567890  clinic4@example.com 2
```

Fig 50: Clinic details stored on Server1 and Server2.

Now, Our GDC is deployed on server1 with our Java program code. Our program or query execution starts from here. Firstly we fetch the details of the server based on the ClinicID as our Partition Key and based on that server we are redirecting our request to that server. As shown in all the below figures our execution will be working based on the instance provided and if instance details are not provided in the query then we fetch those details from the GDC file and then based on that we return the query results.

```
Enter the SQL query:
Select * from Clinic where ClinicID = 3;
Execution of the query on the instance Server1 and the query which will be processed is Select * from Clinic where ClinicID = 3;
ClinicID    Name    LocationID  PhoneNumber Email   HospitalID
3   THS Clinic Ontario  1   1234567890  clinic3@example.com 1
Enter the SQL query:
Select * from Clinic where ClinicID = 4;
Execution of the query on the instance Server2 and the query which will be processed is Select * from Clinic where ClinicID = 4;
ClinicID    Name    LocationID  PhoneNumber Email   HospitalID
4   Halifac Scotia bank Clinic  2   1234567890  clinic4@example.com 2
```

Fig 51: Query execution without an instance specified on both servers.

In This example, we searched for the details of ClinicID 3 as we did not specify the instance the GDC gives us the details such as Server1, and then execution of query processed on Server1 and returned the clinic details. On the other one, we asked for clinic details for ClinicID 4 which is available on Server2.

```
Enter the SQL query:
Select * from Instance1.Clinic where ClinicID = 1;
Execution of the query on the instance Server1 and the query which will be processed is Select * from Instance1.Clinic where ClinicID = 1;
ClinicID    Name    LocationID  PhoneNumber Email   HospitalID
1   THS Clinic Kitchener    1   1234567890  clinic1@example.com 1
Enter the SQL query:
Select * from Instance2.Clinic where ClinicID = 2;
Execution of the query on the instance Server2 and the query which will be processed is Select * from Instance2.Clinic where ClinicID = 2;
ClinicID    Name    LocationID  PhoneNumber Email   HospitalID
2   Halifax Clinic  2   1234567890  clinic2@example.com 2
```

Fig 52: Query execution with an instance specified on both servers.

In the above example, we also provided the Instance details to our query so rather than going to GDC our query execution will be directly performed on the given Server instance as shown above we provided the Instance1. Clinic and Instance2.clinic which refers to the different server instances and both returns the respected data from those servers.

Steps for Cloud Deployment

Horizontal fragmentation is a database design technique used to divide a table's rows into subsets, each stored on a separate database instance. In this scenario, we'll create two SQL instances on Google Cloud Platform (GCP) named "vmysql1" and "vmysql2" to store different fragments of data.

Creating SQL Instances

Step 1: Access to Google Cloud Platform

1. Initiate access by logging into your Google Cloud Platform (GCP) account.

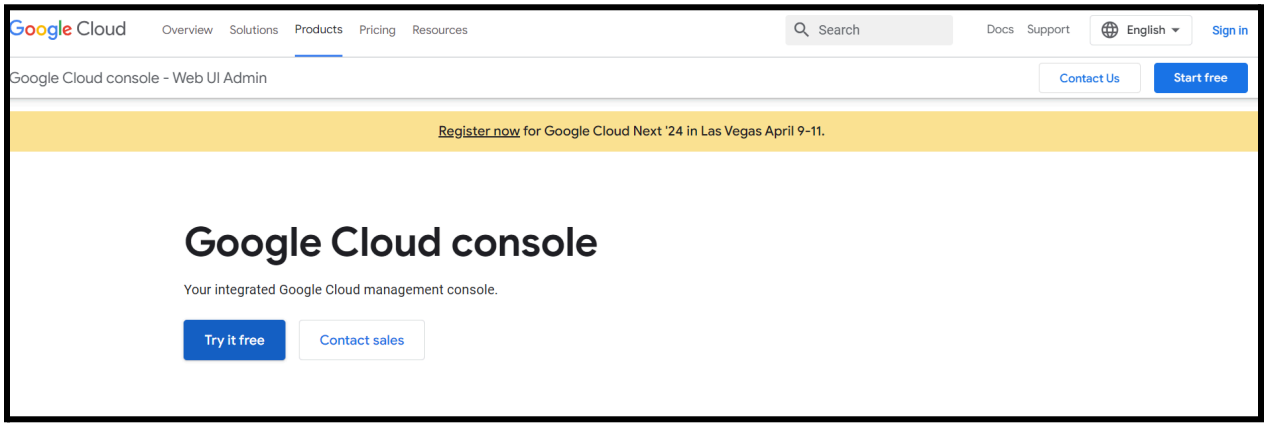


Fig 53: Homepage of Google Cloud Console

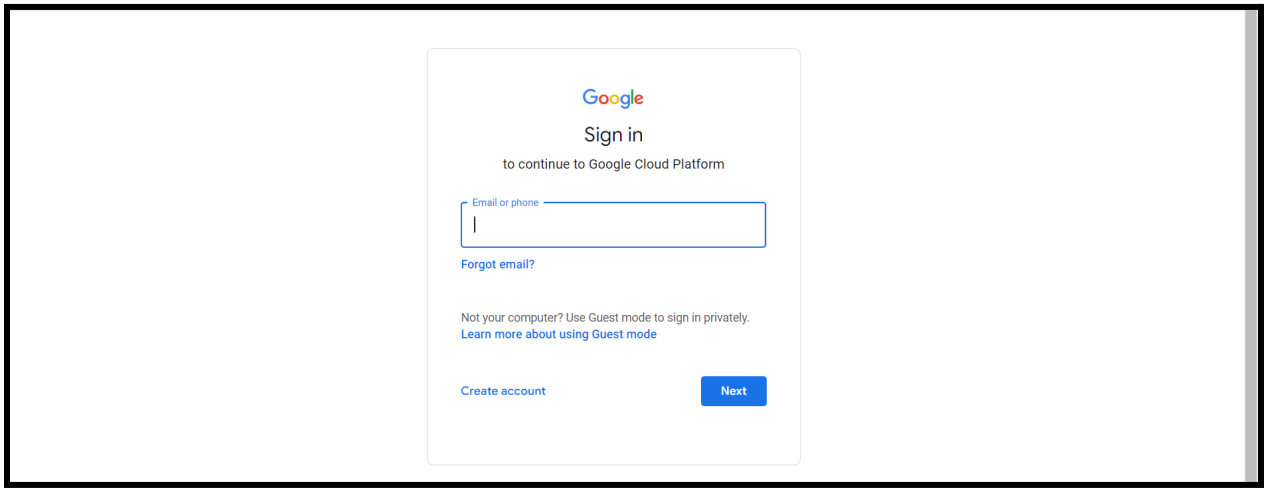


Fig 54: SignIn Page for Google Cloud Platform

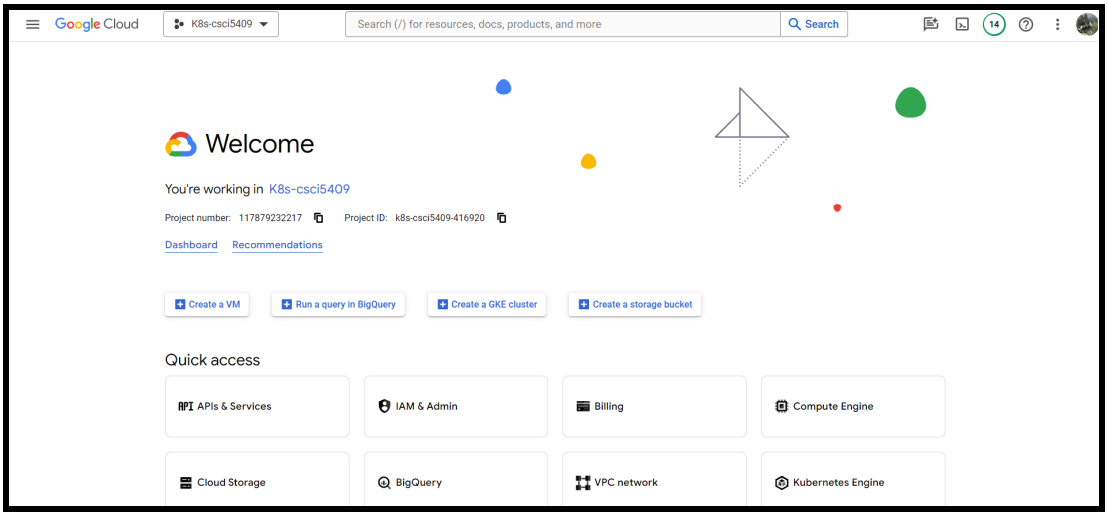


Fig 55: Dashboard for Google Cloud Platform after LogIn

Step 2: Navigation to SQL Instances Management

Proceed to the SQL Instances management screen by selecting the corresponding option from the GCP console's navigation menu.

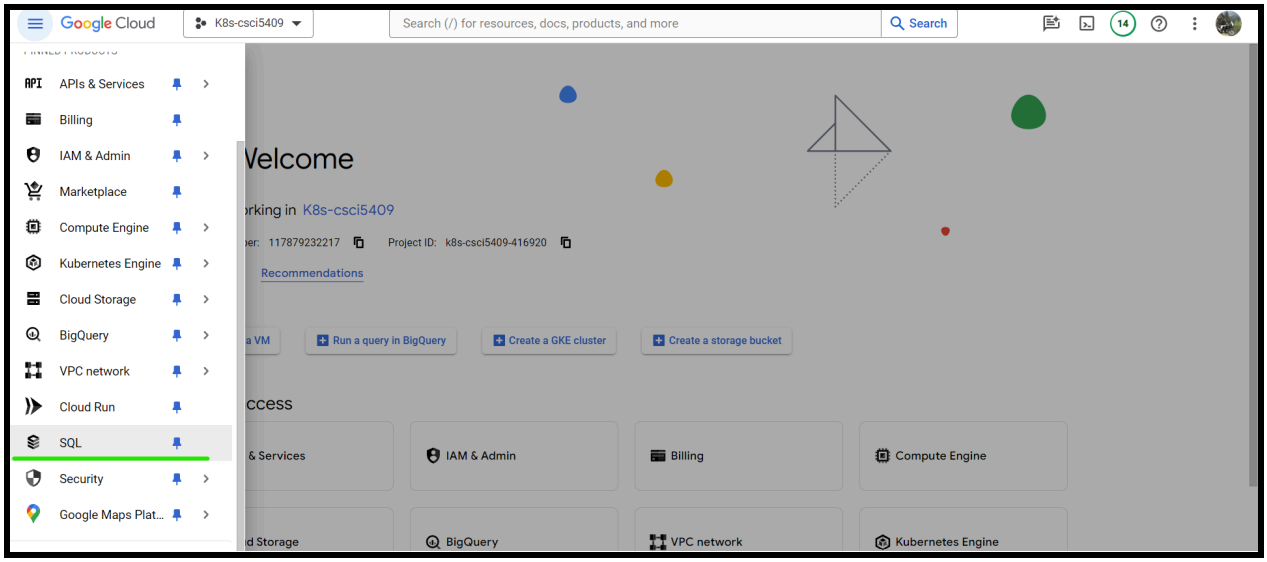


Fig 56: Cloud SQL Option to Manage Database

Step 3: Deployment of MySQL Instance 1 "vmysql1":

1. Initiate the creation process by clicking the "Create Instance" button.

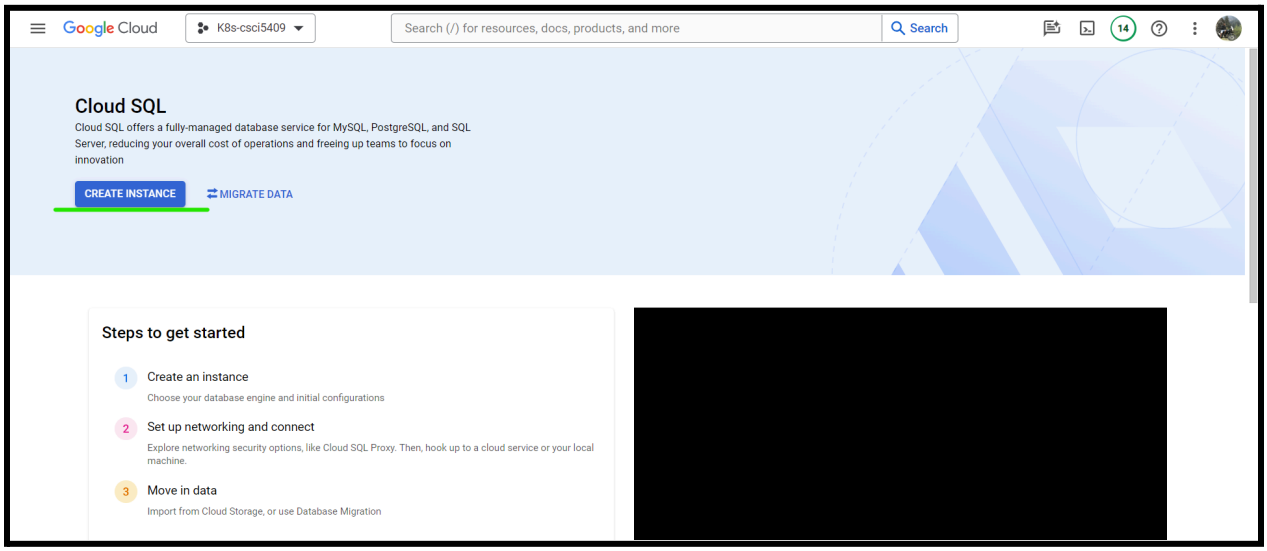


Fig57: Create Instance Button to Create SQL Instance

2. Select MySQL as the database engine of choice.

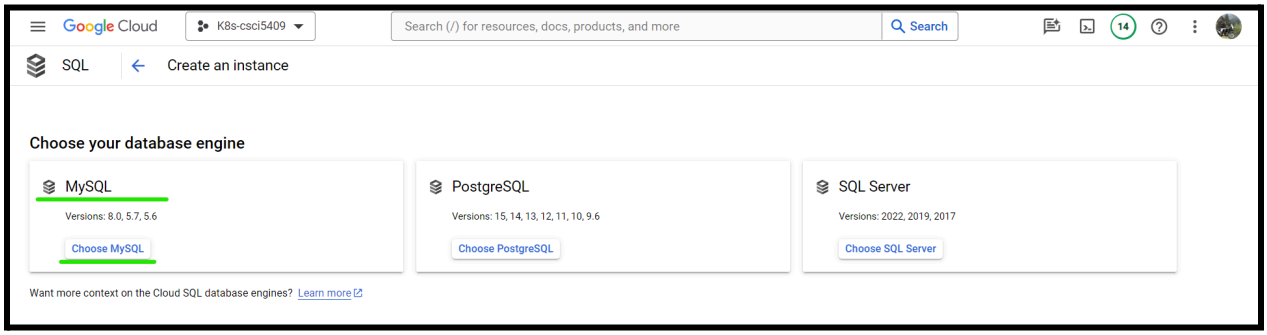


Fig 58: Cloud SQL Database Engine Options

3. Assign a distinctive identifier to the instance, such as "vmysql1".

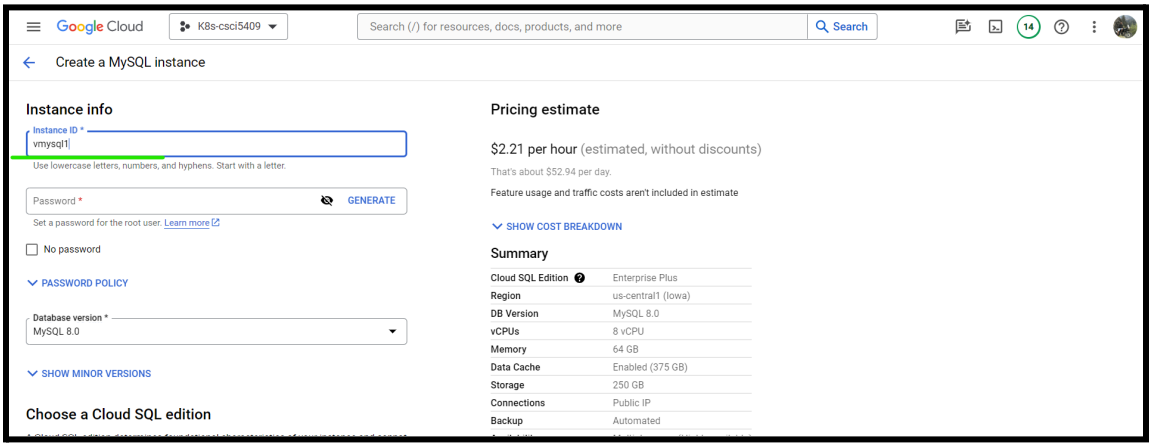


Fig 59: Instance ID “vmysql1” provided

4. Configure the instance according to project requirements, including selecting the appropriate region, and machine type, and specifying the storage capacity.

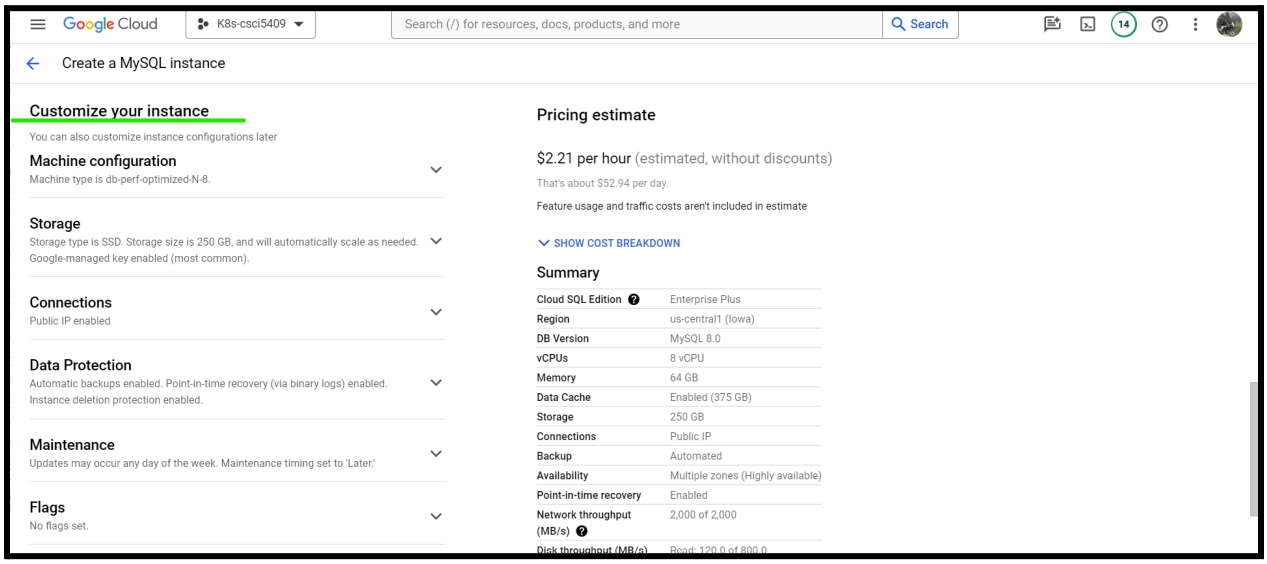


Fig 60: Various configuration options

5. Finalize the instance creation by clicking the "Create" button, thus deploying the "vmysql1" instance.

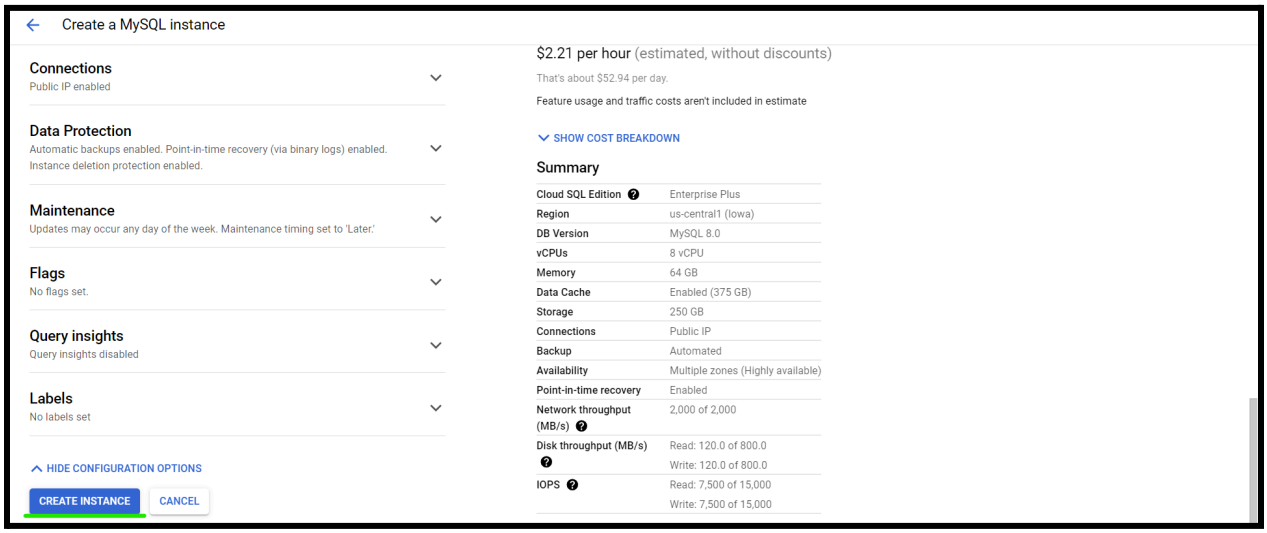


Fig 61: Creating vmysql1

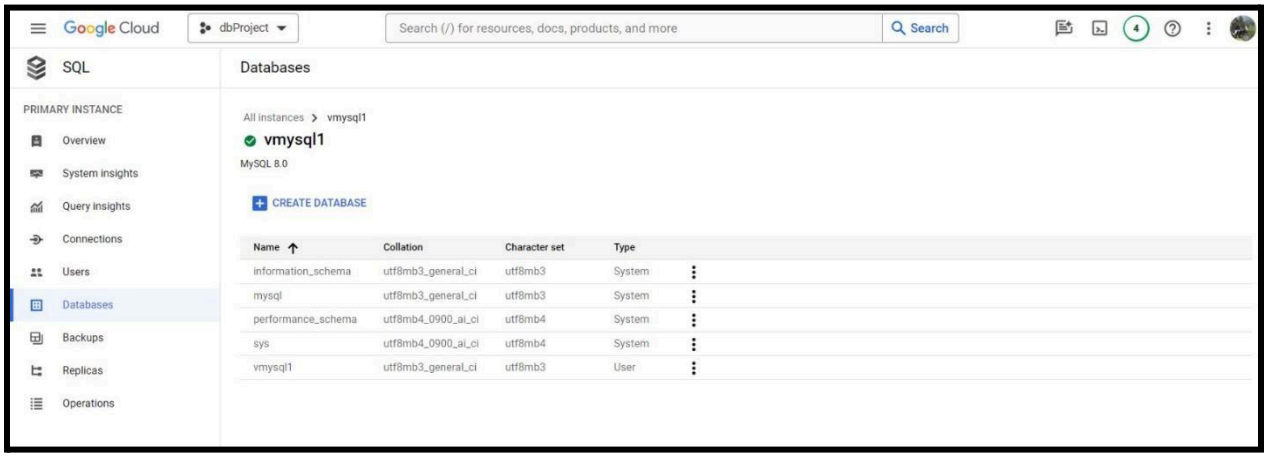


Fig 62: “vmysql1” Cloud SQL instance created

Step 4: Creating "vmysql2" Instance:

1. Follow the procedure outlined in Step 3 to commence the creation of the "vmysql2" instance.
2. Ensure the instance is uniquely identified by naming it "vmysql2".
3. Mirror the configuration settings applied to "vmysql1", adjusting the region or machine type as necessary to fulfill project specifications.

4. Complete the deployment of the "vmysql2" instance by clicking the "Create" button.

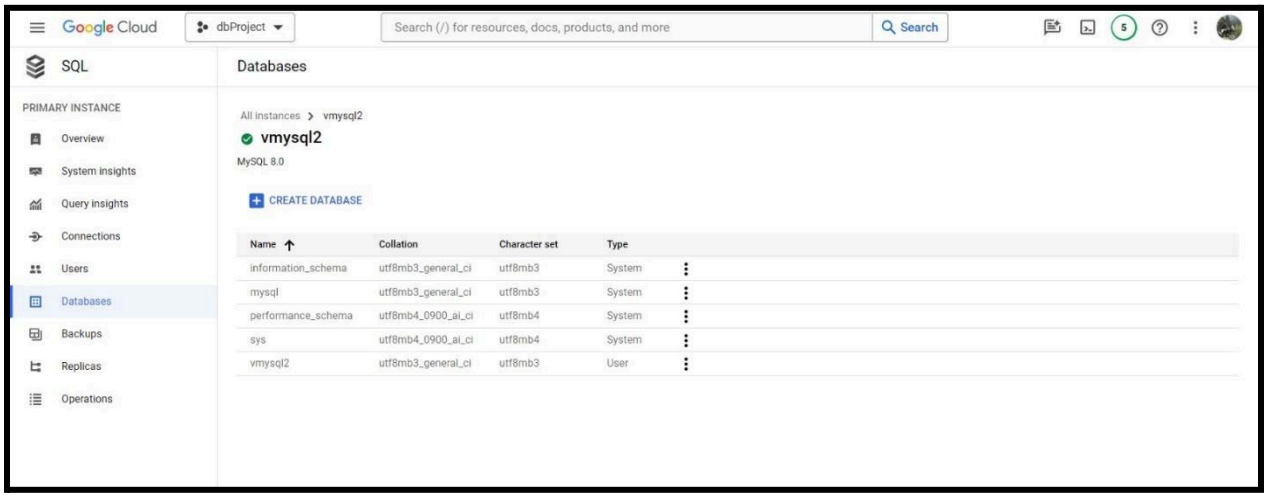


Fig 63: “vmysql2” Cloud SQL instance created

Introduction to Google Cloud Storage Buckets:

Google Cloud Storage (GCS) is a scalable, fully managed object storage service for storing and accessing data globally. Creating a GCS bucket and uploading SQL scripts into it is a fundamental step for managing data on Google Cloud Platform (GCP), supporting data accessibility and security[7].

Justification for Creating a Bucket and Uploading SQL Scripts:

- **Centralized Data Storage:** Utilizing a bucket as a centralized repository for SQL scripts, data files, and other essential resources significantly enhances database management efficiency.
- **Secure Data Management:** Google Cloud Storage is equipped with advanced security features, safeguarding data integrity and confidentiality. This is particularly vital for the secure storage of sensitive SQL scripts and related files.
- **Scalability and Availability:** The scalable and reliable nature of GCS buckets ensures the effortless accommodation of increasing data volumes, alongside maintaining high availability for the SQL scripts and resources needed for various operations.
- **Integration with Google Cloud Services:** The seamless integration of GCS with other services within the Google Cloud ecosystem allows for the efficient access and execution of SQL scripts stored in buckets, whether through Google Cloud SQL instances or other connected services.

Creating Bucket and Uploading SQL Scripts:

Step 1: Accessing Google Cloud Console

1. Begin by logging into your Google Cloud Platform (GCP) account.
2. Proceed to the Google Cloud Console to start the process.

Step 2: Creating a Bucket

1. Within the Cloud Console dashboard, navigate to the "Storage" section.

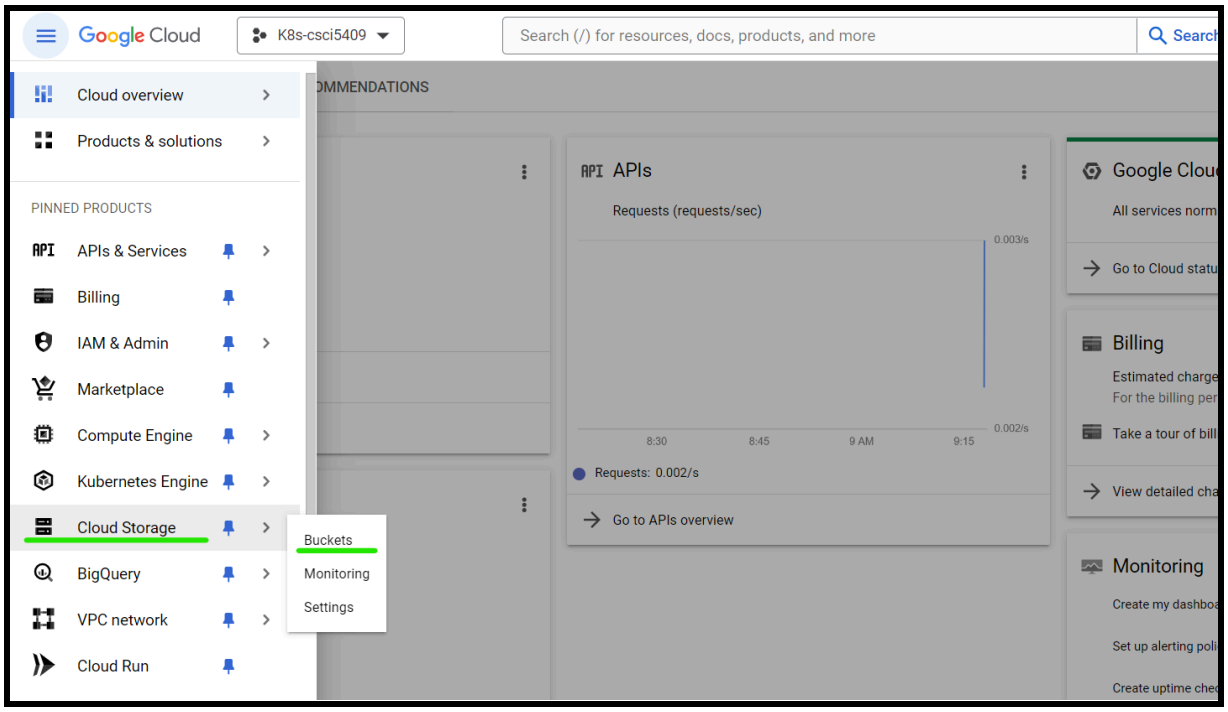


Fig 64: Buckets Option under Cloud Storage Section

2. Click on "Create Bucket" to commence the setup.

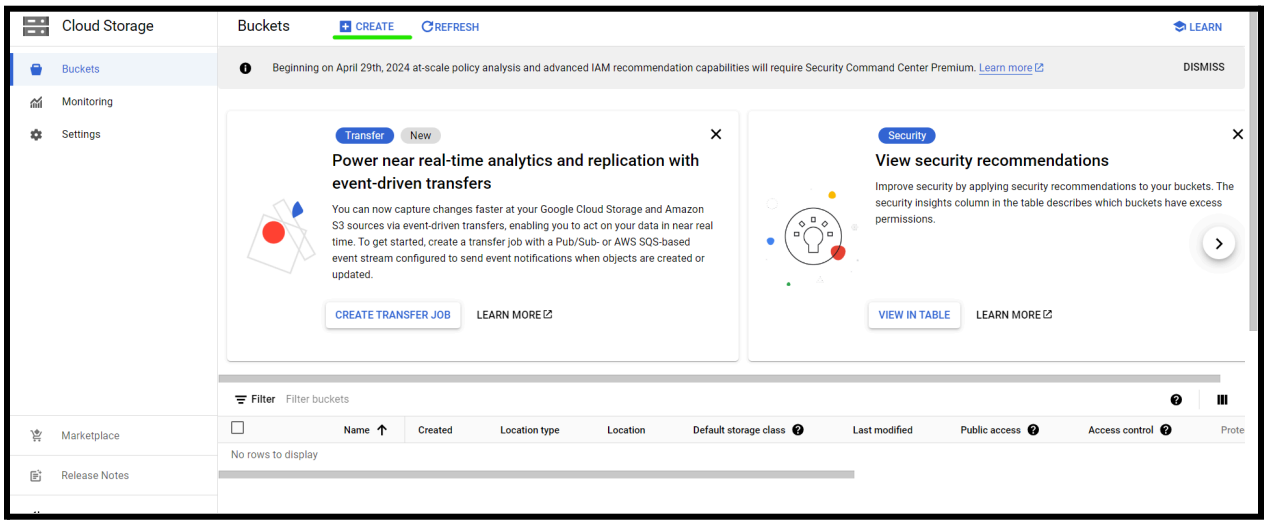


Fig 65: Cloud Storage with no Bucket and Create Option

3. Assign a globally unique identifier to your bucket, such as "hospital-management".

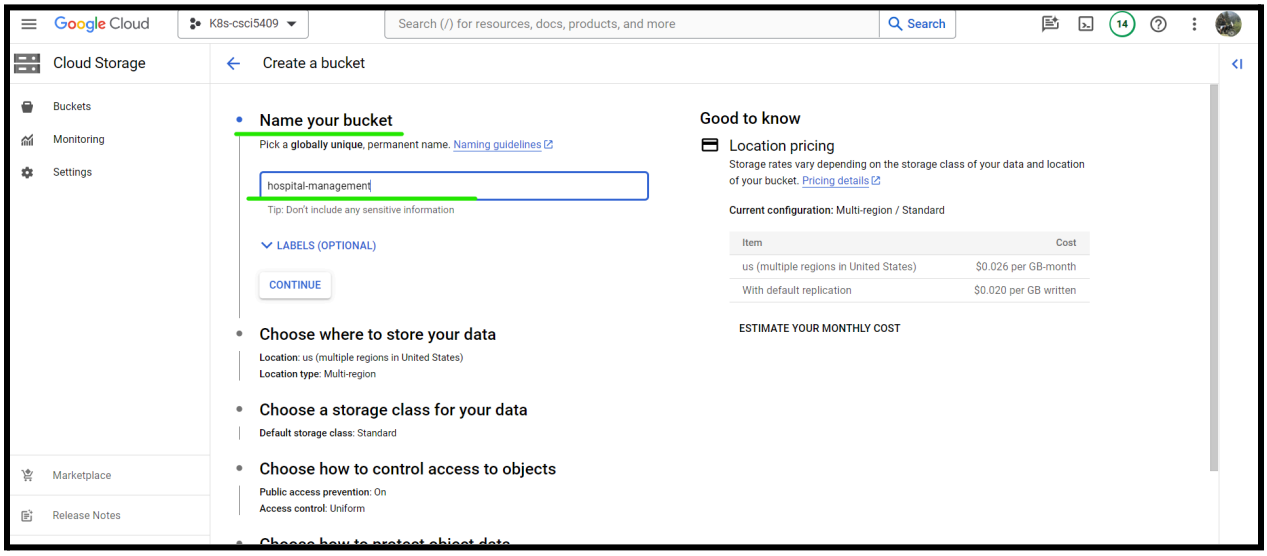


Fig 66: Bucket Name Given “hospital-management”

4. Choose a strategic location for your bucket to optimize data storage.

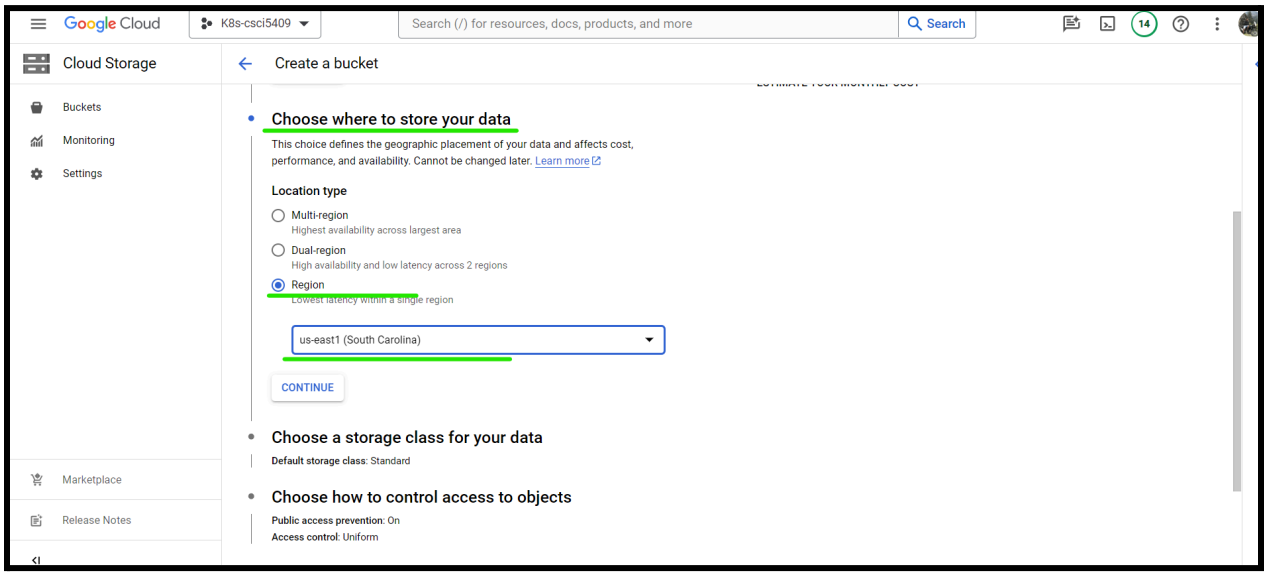


Fig 67: Strategic Location for Bucket chosen

5. Select a storage class that aligns with your needs, considering factors such as performance and cost.

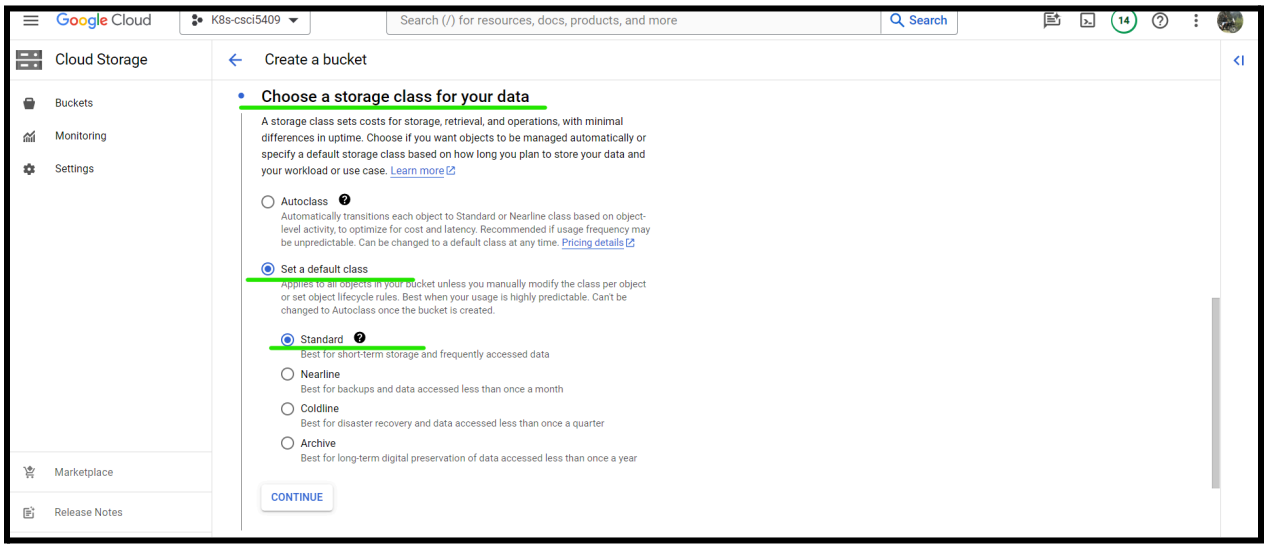


Fig 68: Storage Class Selected

6. Optionally, adjust access control settings and set retention policies to meet specific requirements.

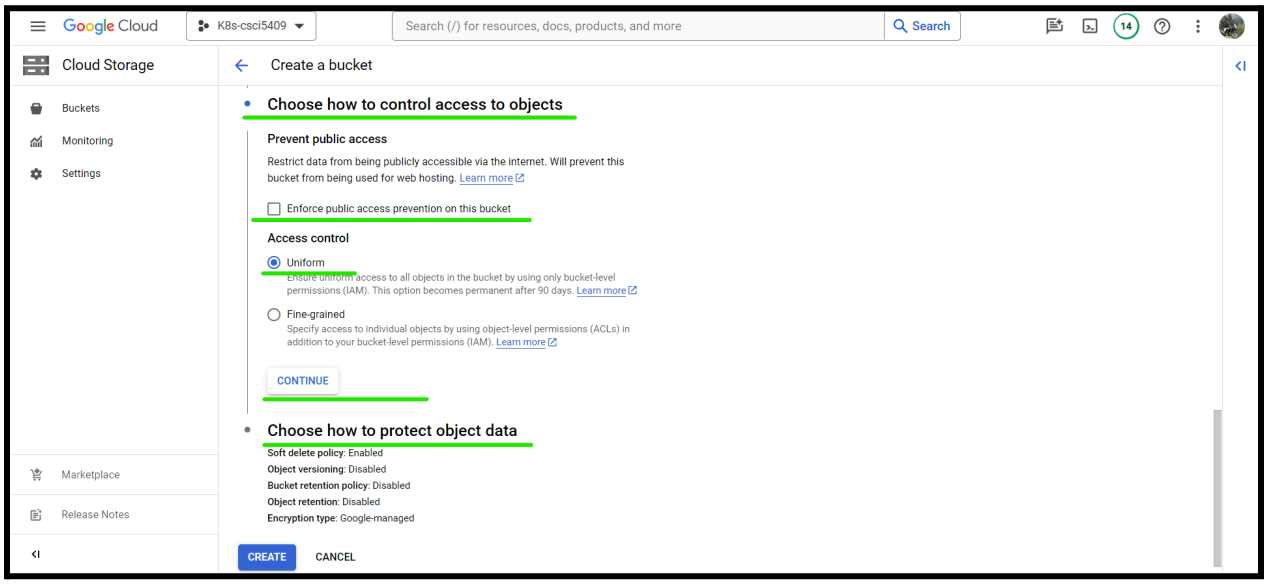


Fig 69: Control Settings and Retention Policies Selected

7. Finalize the creation by clicking on "Create".

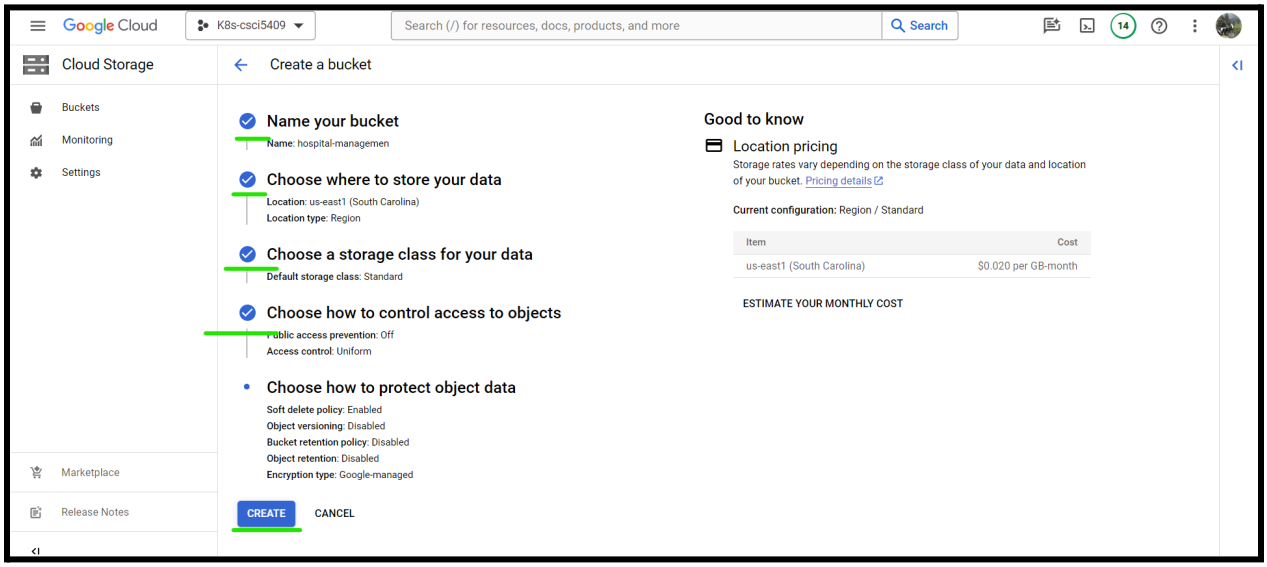


Fig 70: Finalized Configurations for bucket

Note: The execution of these steps not only facilitates the organization and management of SQL scripts and data files but also capitalizes on the robust features of Google Cloud Storage. This approach ensures that data management processes are both streamlined and secure, leveraging the scalability and integration capabilities of GCP to support a wide range of applications and services.

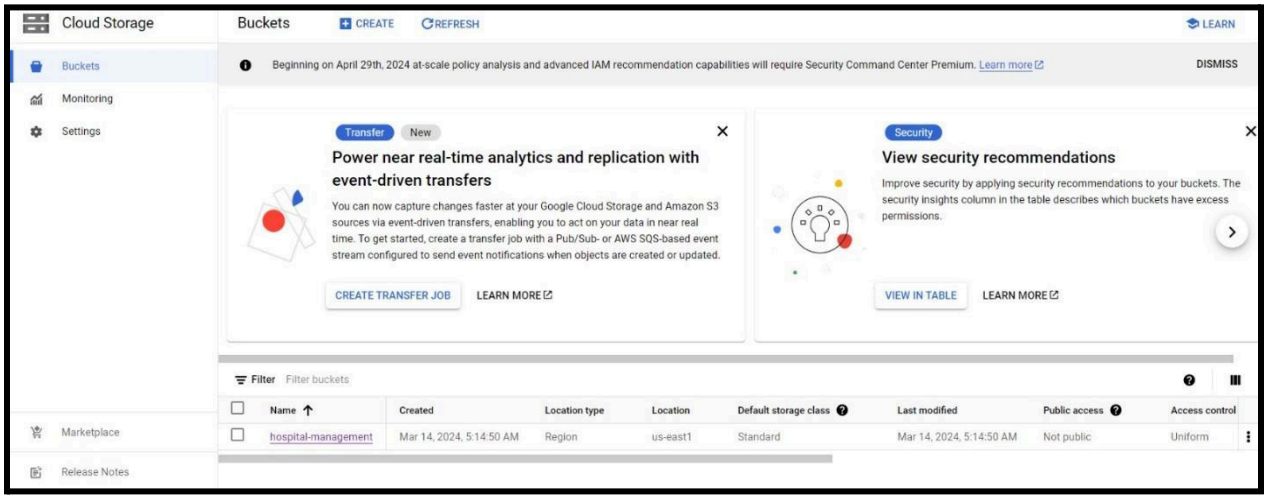


Fig 71: “hospital-management” bucket created

Step 3: Uploading SQL Scripts to the Newly Created Google Cloud Storage Bucket

After the successful creation of Google Cloud Storage bucket, the next crucial step is to populate it with the necessary SQL scripts. Below are the detailed instructions used to upload SQL script files to the bucket:

1. Accessing the Bucket:
    a. From the Google Cloud Console, locate and click on the name of your newly created bucket to view its details.

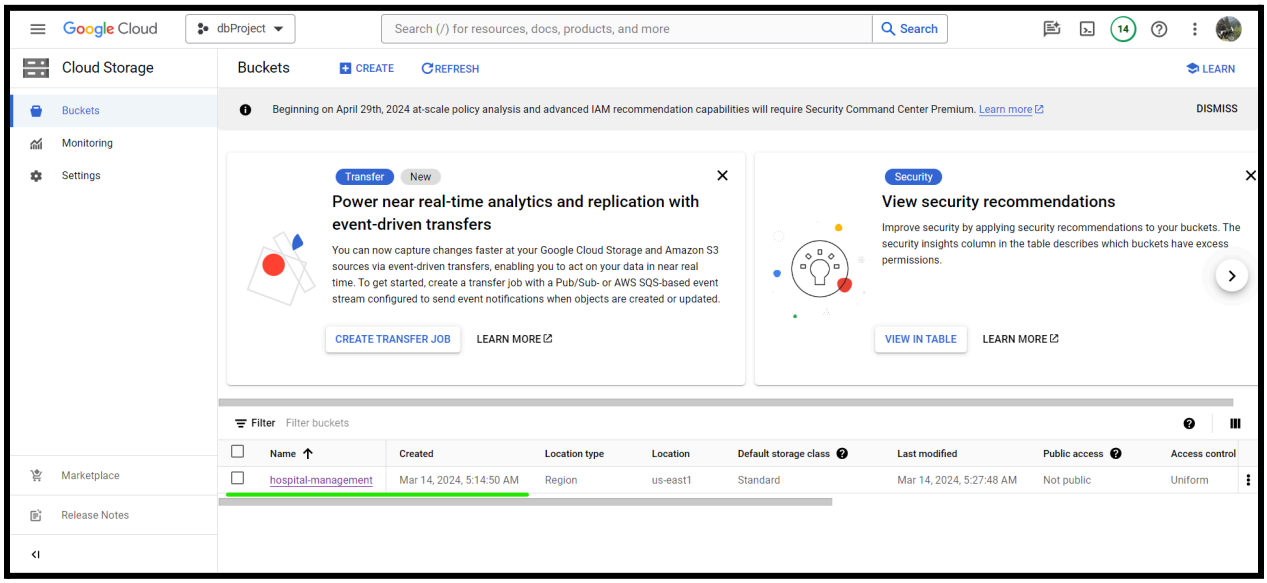


Fig 72: Newly Created Bucket

2. Navigating to Upload Section:
    a. Within the bucket's details page, look for the "Files" tab or the "Upload" section. This is typically part of the bucket's overview page.

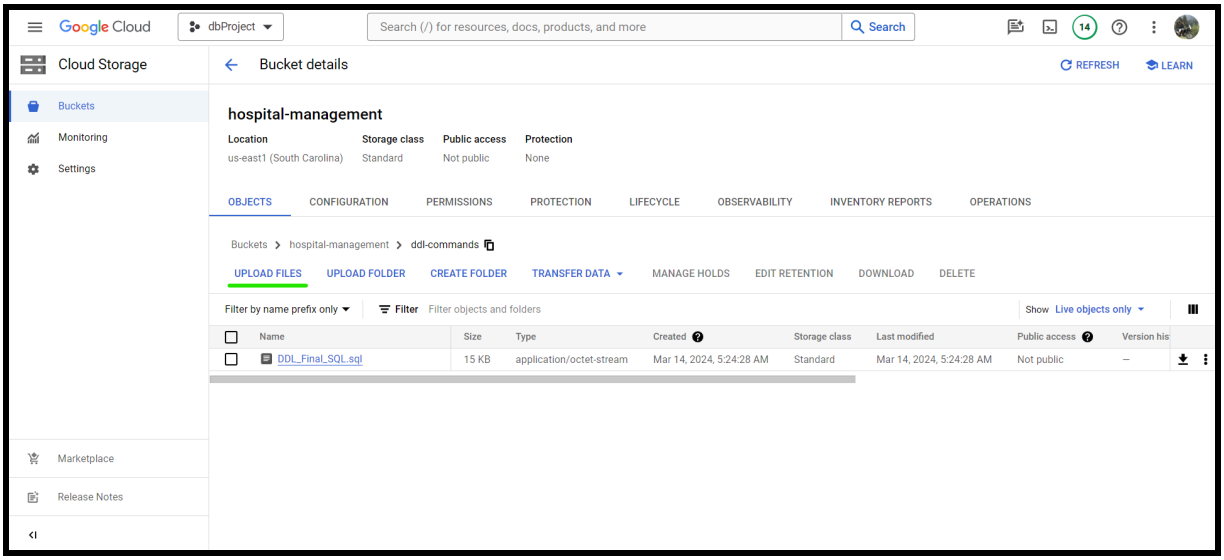


Fig 73: Upload File Option

3. Initiating the Upload:
    a. Click on the "Upload Files" button. Alternatively, you can drag and drop files directly into the browser window if this feature is supported.
4. Selecting SQL Scripts:
    a. From the file selector dialog, navigate to the folder on your local machine named "ddl-commands".
    b. Locate and select the "DDL_Final_SQL.sql" file, or any other SQL script files you intend to upload.

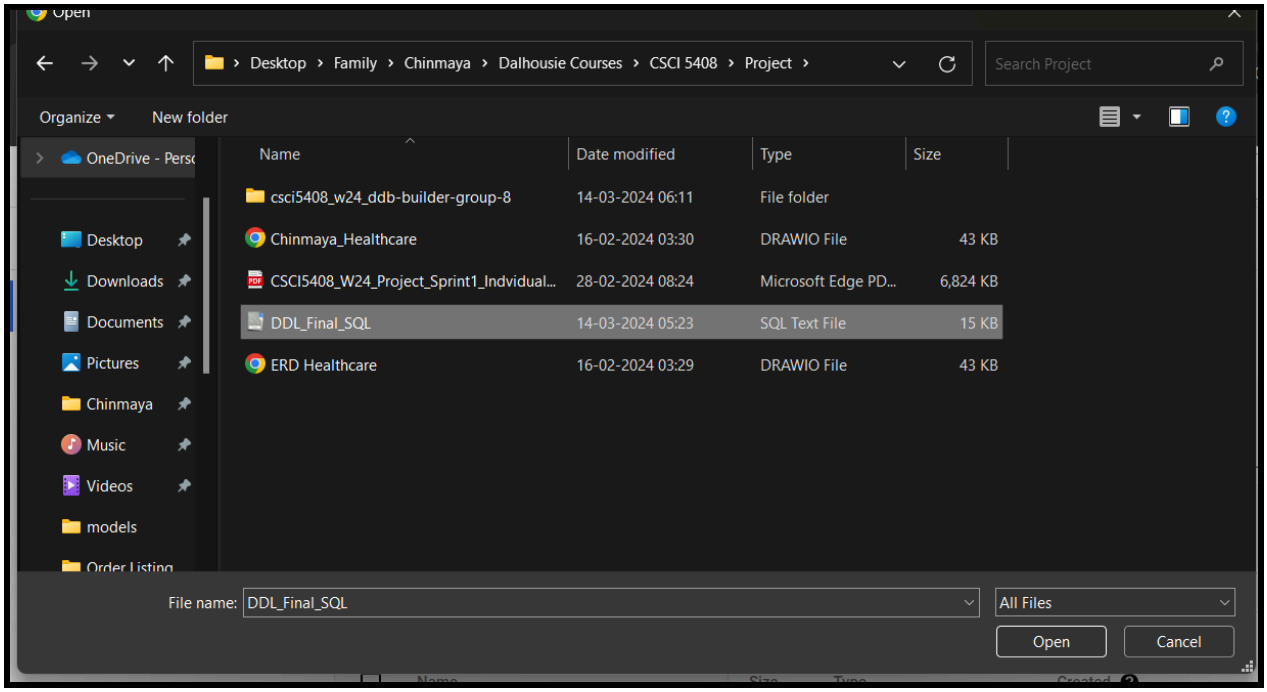


Fig 73: Select the SQL File to be Uploaded

5. Uploading the File:
    a. Confirm your file selection and proceed by clicking the "Upload" button to initiate the transfer of the SQL script file to your GCS bucket.
6. Upload Monitoring and Verification:
    a. Observe the upload progress, typically indicated by a progress bar or status messages within the Cloud Console.
    b. Once the upload is complete, verify that the "DDL_Final_SQL.sql" file appears in the bucket's file list and there are no error messages.

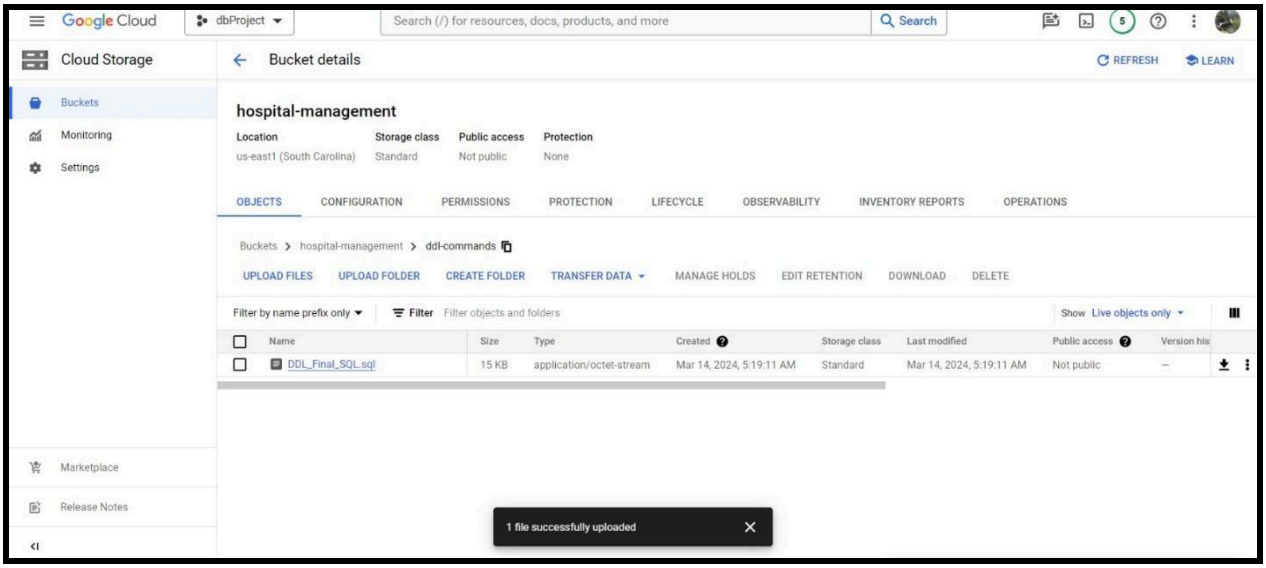


Fig 74: SQL File successfully uploaded

Importing Data from Google Cloud Storage Bucket to Google Cloud SQL

Overview

The process of importing data from a Google Cloud Storage (GCS) bucket into Google Cloud SQL instances is an essential component of database management. This enables the migration of vital resources such as SQL scripts, database backups, and data files from a secure cloud storage location to SQL instances for execution and data management purposes.

1. Select the Google Cloud SQL instance

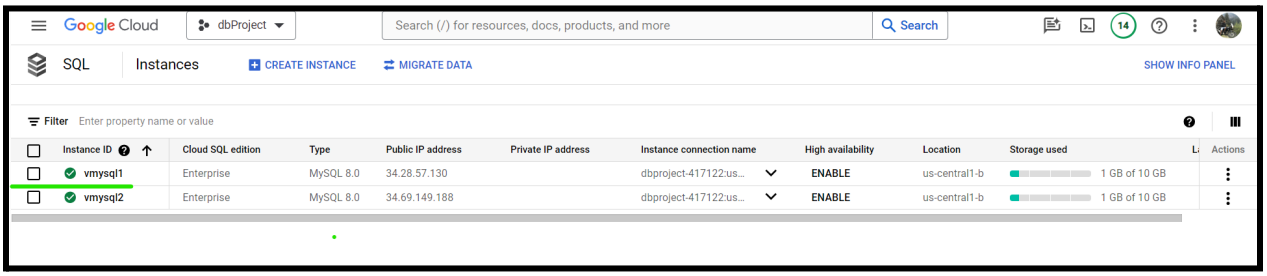


Fig 75: Select the instance

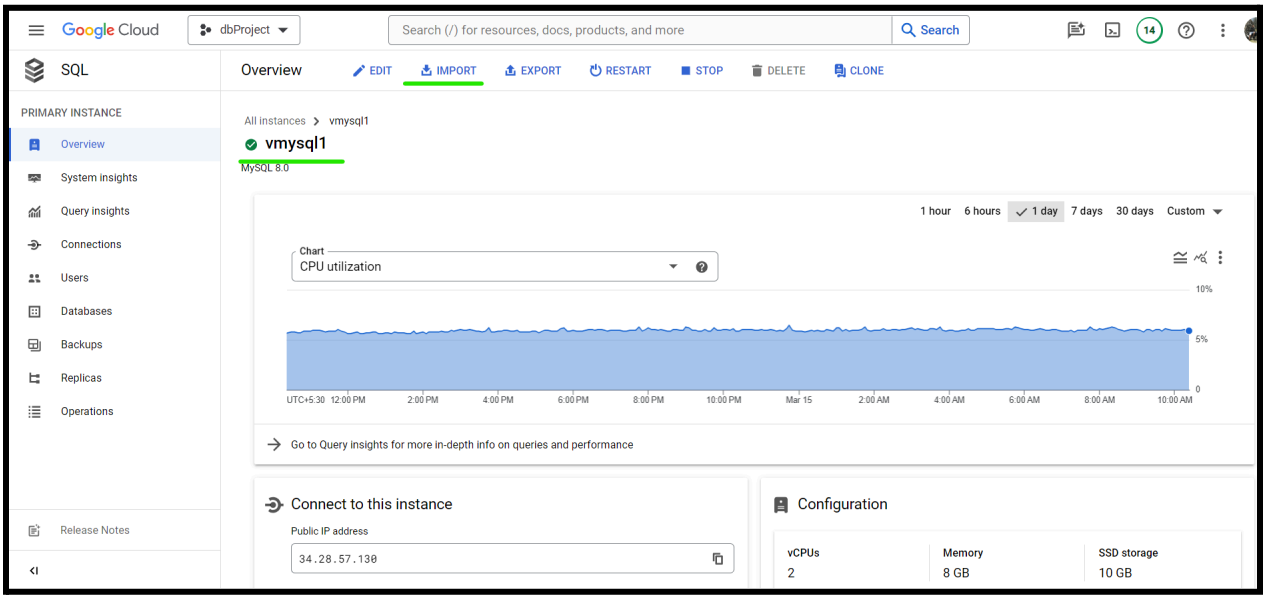


Fig 76: Import Button for “vmysql1” Cloud SQL Instance

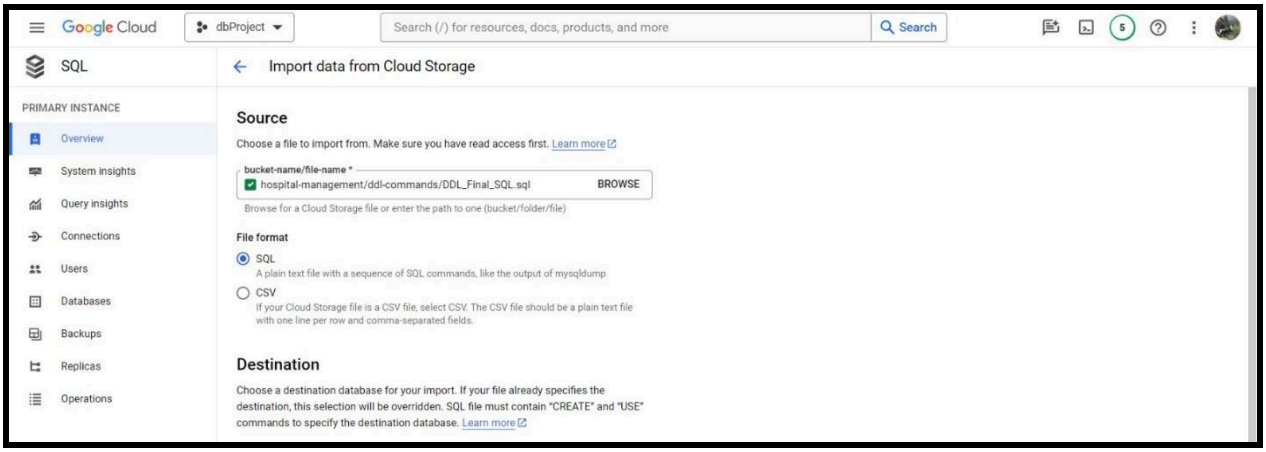


Fig 77: Landing Page to Import ".sql" file

2. Setting the Source
    a. Configure GCS Bucket "hospital-management": Identify and select the GCS bucket "hospital-management" that contains the "ddl-commands" folder.

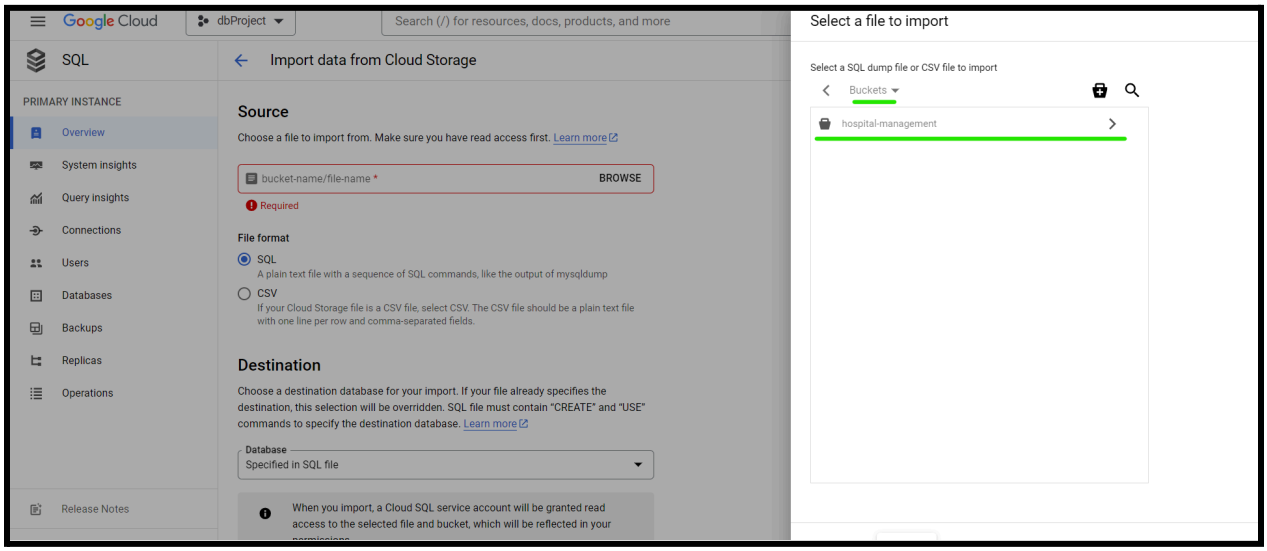


Fig 78: Select Bucket

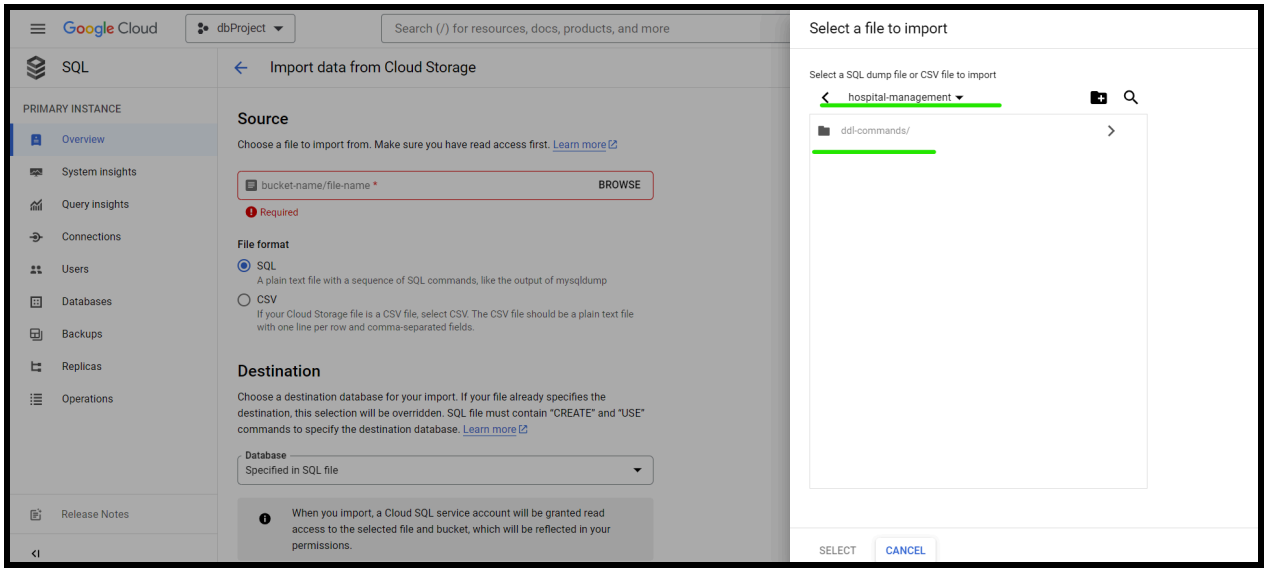


Fig 79: Select Folder

b. Prepare DDL Commands: The DDL commands within this folder provide the necessary instructions for establishing the database schema within Cloud SQL instances, creating tables, views, indexes, and constraints.

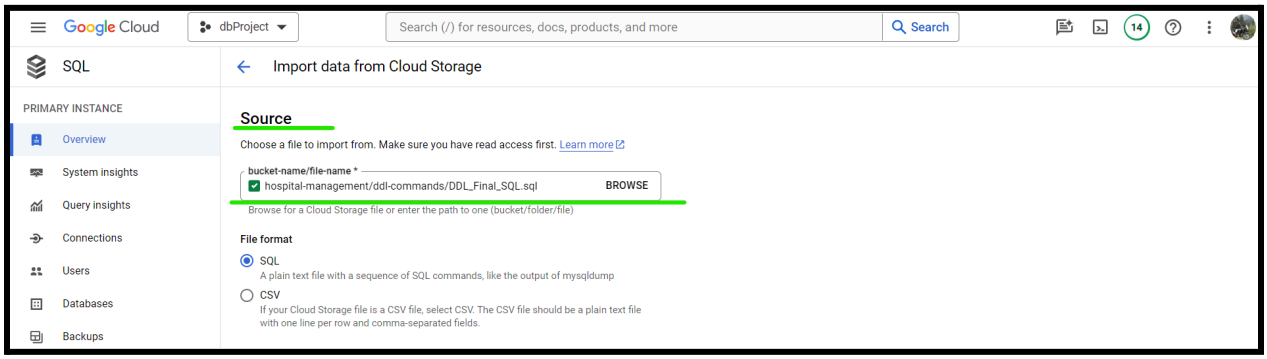


Fig 80: Source file containing SQL statements selected

3. Setting the Destination
    a. Specify Cloud SQL Instance Database: Determine the target Google Cloud SQL instance's database or specify DDL command "CREATE" and "USE" in the ".sql" file.

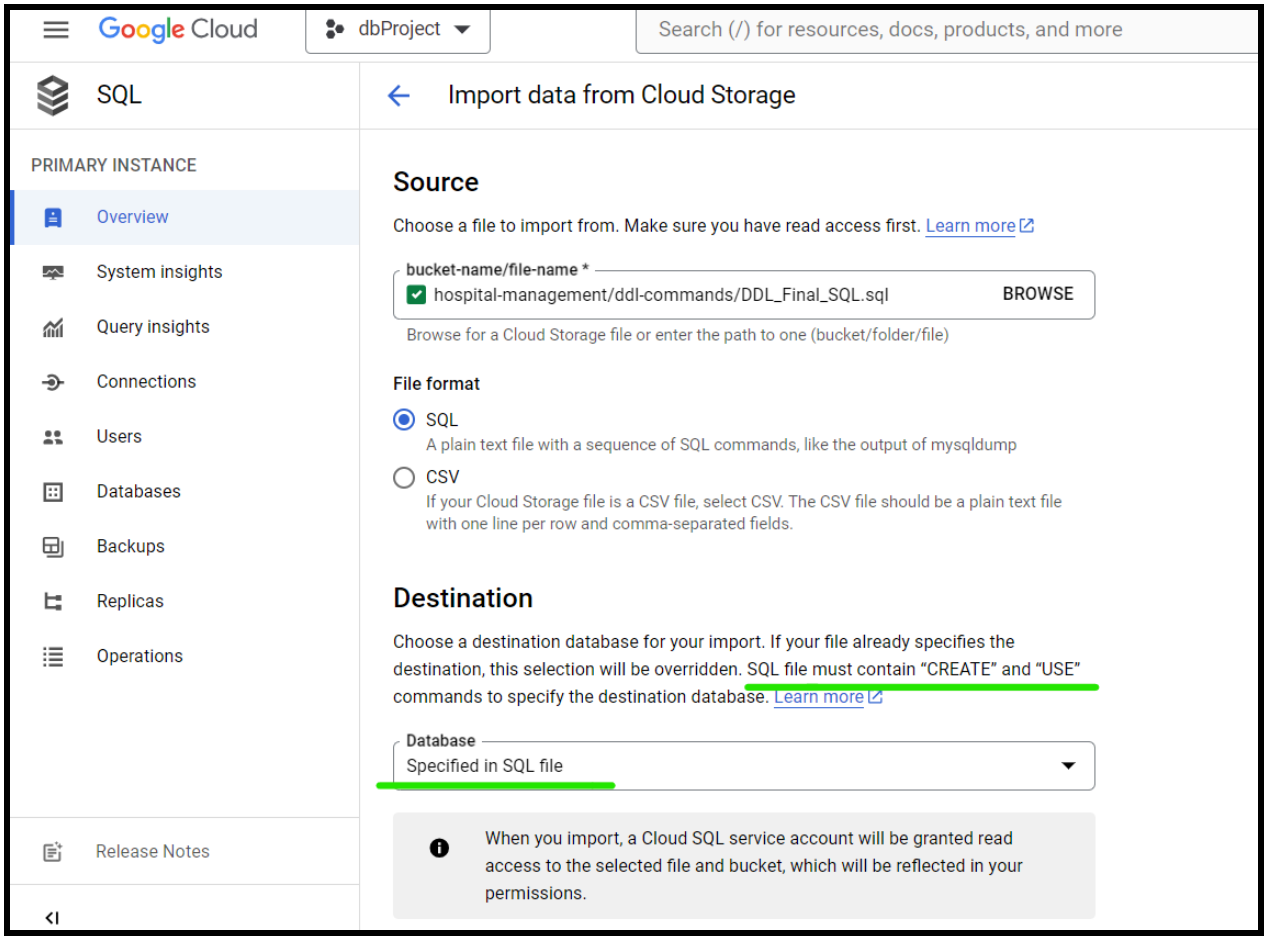


Fig 81: Destination Specified

4. Execute DDL Commands: Import the DDL commands into the designated Cloud SQL instances to create and implement the database schema uniformly.

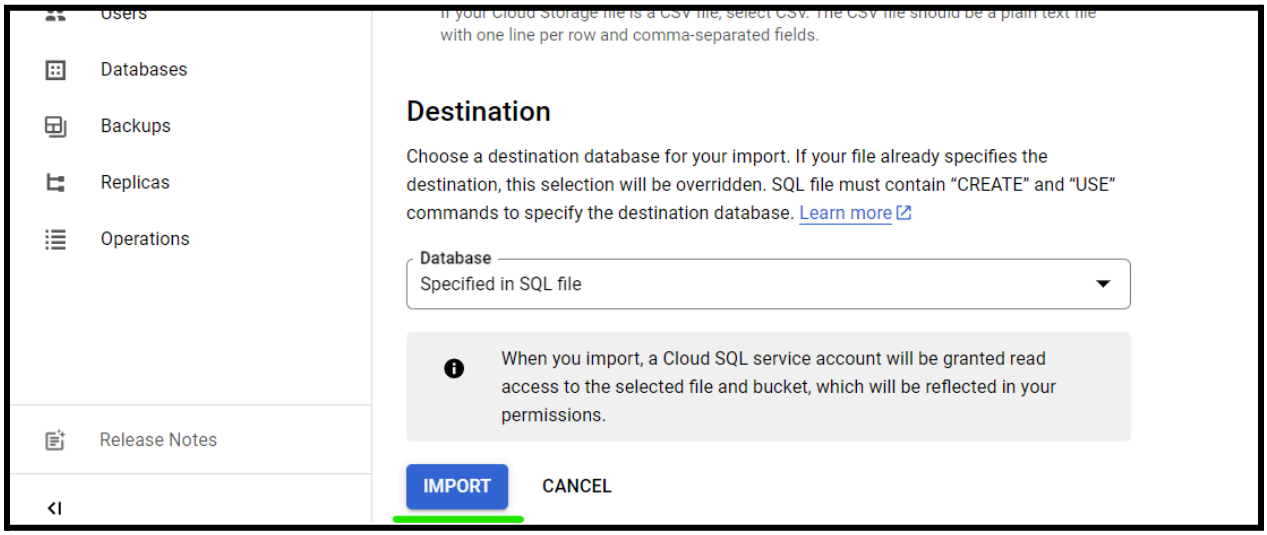


Fig 82: Import Button to run SQL DDL Command Present in ".sql" File

Snippets of the Database Created

- **VMySQL1 Server Setup in MySQL Workbench**

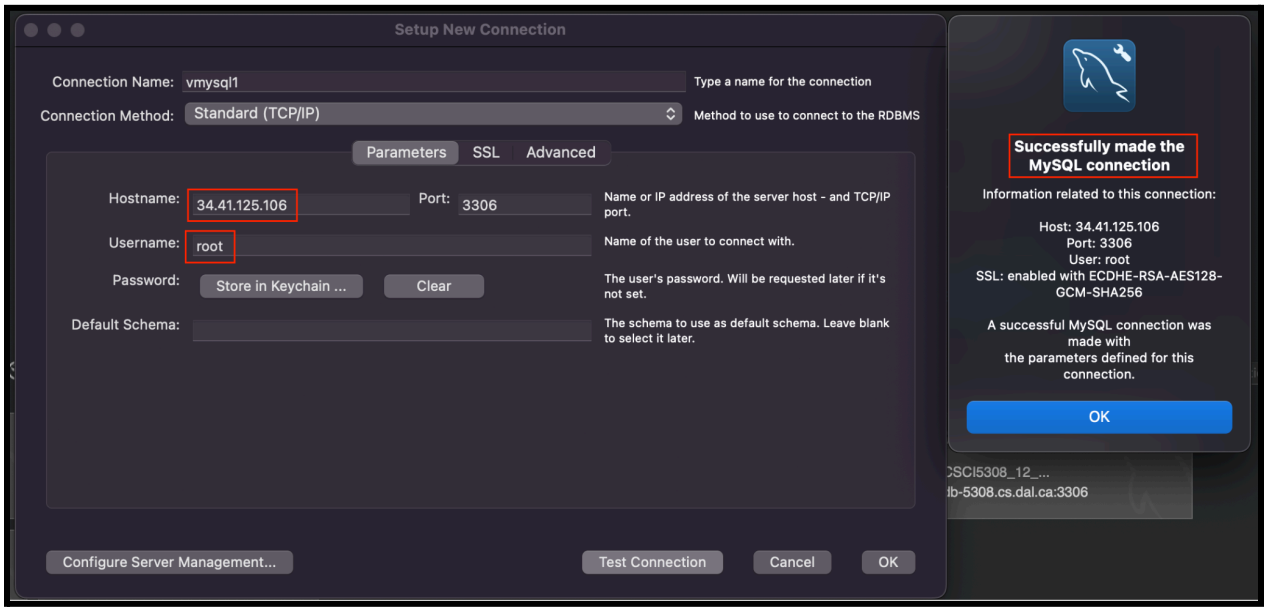


Fig 83: MySQL Workbench Connection made with vmysql1

- **VMySQL2 Server Setup in MySQL Workbench**

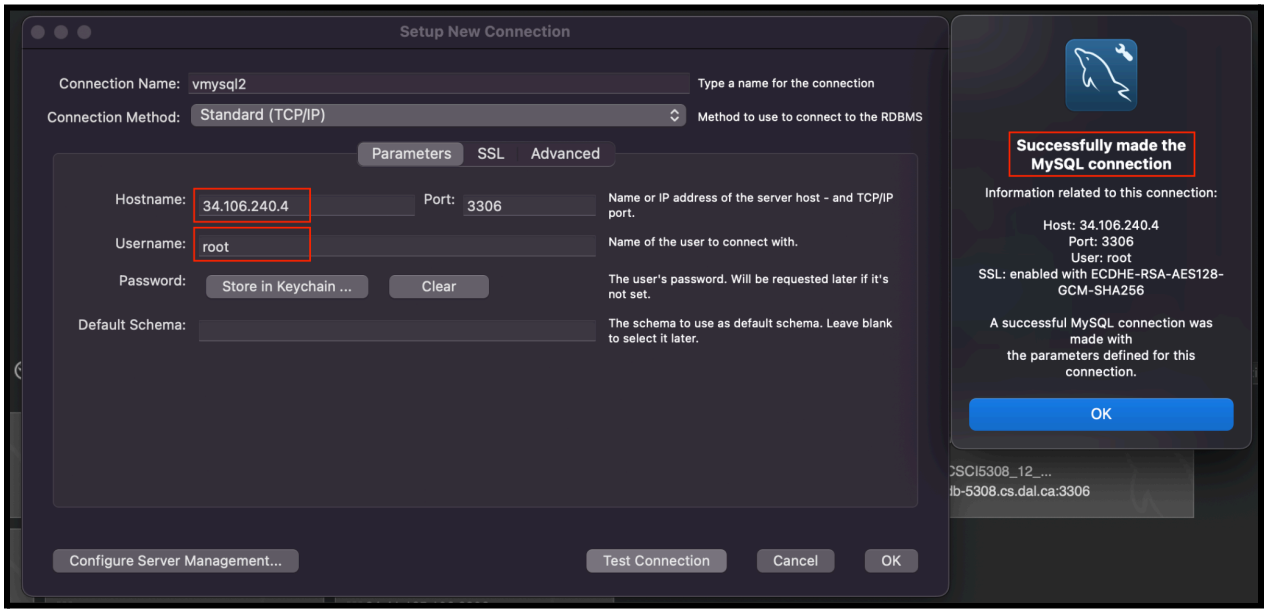
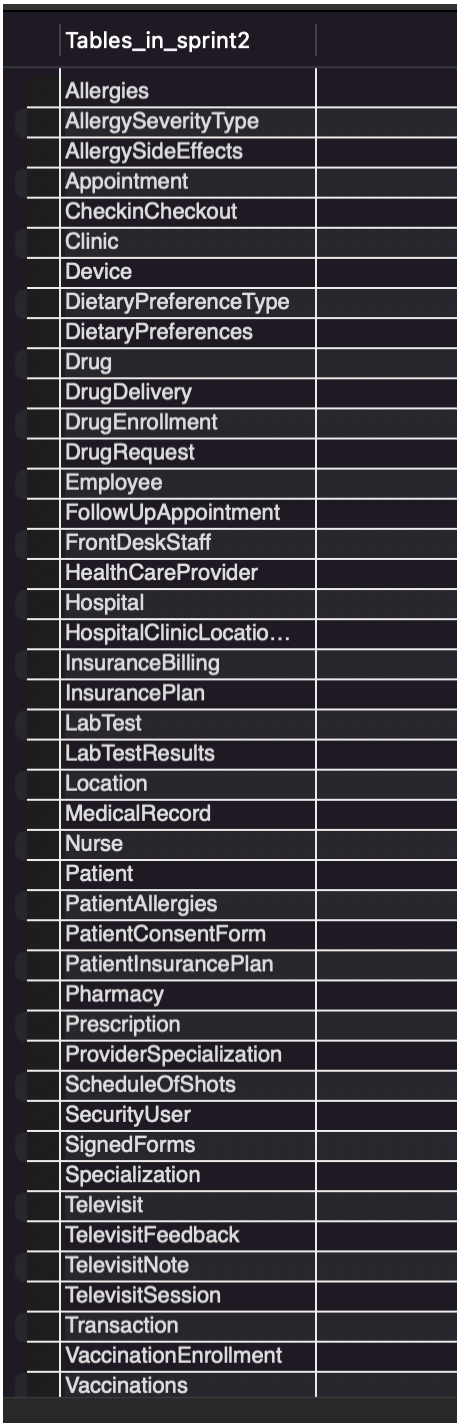


Fig 84: MySQL Workbench Connection made with vmysql2

- **VMySQL1 Server Tables**



Tables_in_sprint2
Allergies
AllergySeverityType
AllergySideEffects
Appointment
CheckinCheckout
Clinic
Device
DietaryPreferenceType
DietaryPreferences
Drug
DrugDelivery
DrugEnrollment
DrugRequest
Employee
FollowUpAppointment
FrontDeskStaff
HealthCareProvider
Hospital
HospitalClinicLocatio…
InsuranceBilling
InsurancePlan
LabTest
LabTestResults
Location
MedicalRecord
Nurse
Patient
PatientAllergies
PatientConsentForm
PatientInsurancePlan
Pharmacy
Prescription
ProviderSpecialization
ScheduleOfShots
SecurityUser
SignedForms
Specialization
Televisit
TelevisitFeedback
TelevisitNote
TelevisitSession
Transaction
VaccinationEnrollment
Vaccinations

Fig 85: Tables present in vmysql1

- **VMySQL2 Server Tables**

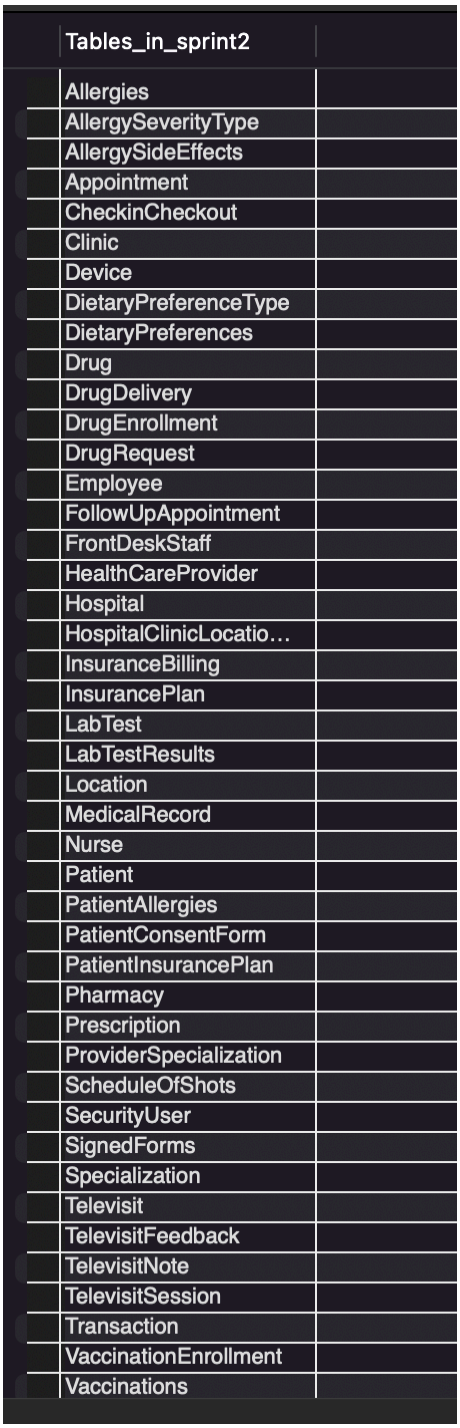


Fig 86: Tables present in vmysql2

Importance of Fragmentation in the Builder Project

- **Database Schema Establishment:** Ensuring the database schema is uniformly applied across multiple instances facilitates consistent schema definition and data organization.
- **Horizontal Fragmentation Support:** This process is integral to the project's horizontal fragmentation strategy, promoting scalability and data consistency.
- **Data Management Efficiency:** Centralization of DDL commands in a GCS bucket and execution on Cloud SQL instances optimizes data management, system reliability, and scalability.

Repetition for Each Instance:

1. **Uniform Schema Application:** The import process must be replicated for each SQL instance, like "vmysql1" and "vmysql2", to maintain schema uniformity across the distributed database system.
2. **Consistency Across Instances:** Repeating the import process ensures that all instances have a consistent database structure, critical for the system's horizontal fragmentation and overall integrity.

Monitoring CPU Utilization

1. **Assessing System Load**: Screenshots of CPU utilization provide a snapshot of the processing demand placed on instances "vmysql1" and "vmysql2" during DDL command execution.
2. **Database Initialization Indication**: Elevated CPU usage during this phase suggests active processing of the DDL commands, a necessary step in establishing the database's structural components.

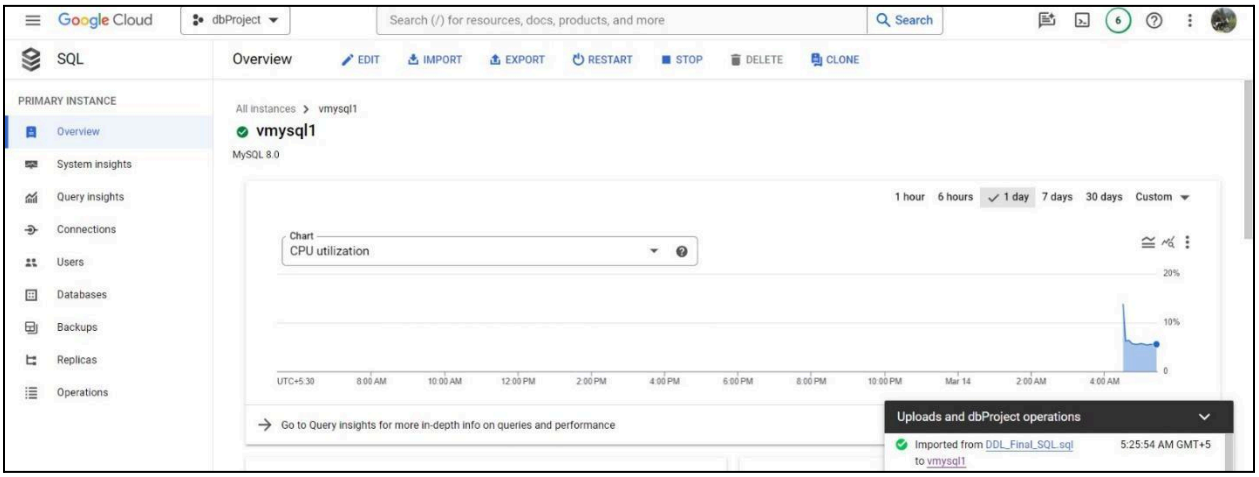


Fig 87: “vmysql1” CPU Utilization After Running DDL Queries to Create DB “sprint2” and Tables

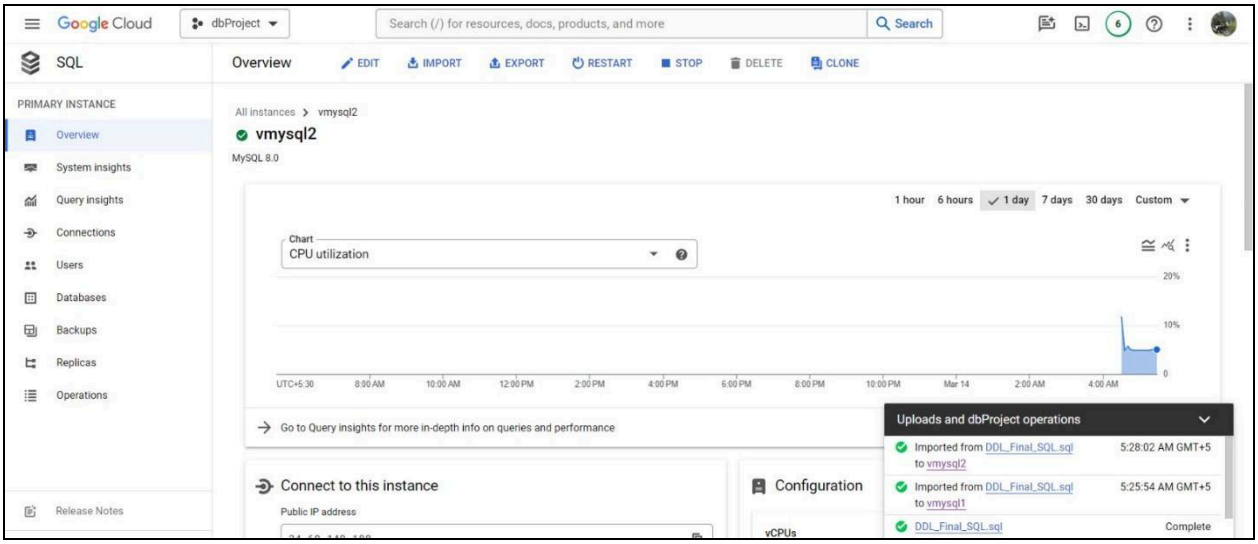


Fig 88: “vmysql2” CPU Utilization After Running DDL Queries to Create DB “sprint2” and Tables

3. **Performance Insights**: Monitoring CPU utilization is a key practice for evaluating system performance, identifying potential processing bottlenecks, and making informed decisions about resource management to optimize database operations.

In summary, the structured importation of DDL commands from a GCS bucket to Cloud SQL instances is a strategic approach to establishing a consistent and scalable database schema, which is essential for distributed database management systems. The methodical replication of this process across multiple instances ensures coherent schema deployment and supports the efficient operation of a horizontally fragmented database architecture. Monitoring CPU utilization during this process provides insights into the system's performance, guiding resource optimization and ensuring smooth database initialization.

Global Data Coordinator (GDC) Code Documentations

```
public static void main(String[] args) {
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");

        String query1 = "Select * from Clinic where ClinicID = 1;";
        String query2 = "Select * from Instance2.Clinic where ClinicID = 2;";
        executeClinicQuery(query1);
        executeClinicQuery(query2);
        // Create a Scanner object to read input from the command line
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("Enter the SQL query:");
            String query = scanner.nextLine();
            if (query.equalsIgnoreCase("exit")) {
                break;
            }
            executeClinicQuery(query);
        }
        System.out.println("Connection initiated...");
        Connection con = DriverManager.getConnection(vmysql1, cloud_username, cloud_password);
        storeDetailsToFile();
        String instanceName = getInstanceNameForClinic(1);
        System.out.println(instanceName);
        insertHospitalAndClinicData();
        see_hospital_location_mapping_server_1();
        see_hospital_server2();
        System.out.println("Connected to the database!");
        con.close();
    } catch (Exception e) {
        System.out.println(e.getMessage());
        System.out.println(e);
    }
}
```

1. **Main Method:**

The main method is the program's entry point. Here's a breakdown of its key functionalities:

- **Establishes Database Connections:** It defines connection details for two database instances (vmysql1 and vmysql2) and sets up credentials for username (cloud_username) and password (cloud_password).
- **Test Data Insertion (Optional):** It demonstrates how to insert dummy data (hospitals, clinics, and locations) into both database instances using helper methods like insertHospital, insertClinic, and insertLocation. This section is commented out by default but can be used for testing purposes.
- **User Input Loop:** It enters a loop where it prompts the user to enter an SQL query. The loop continues until the user enters "exit".
- **Query Execution:** The entered query is passed to the executeClinicQuery method for processing.

- **Database Connection and Result Display:** Upon successful query execution, the executeClinicQuery method establishes a connection to the relevant database and retrieves results. It then displays the retrieved data on the console.

```
public static void executeClinicQuery(String query) {
    try {
        String instanceName = "";
        // Parse the query to extract the instance name
        QueryInfo queryInfo = parseQuery(query);

        // If instance name not provided, retrieve it from the parser file based on clinic ID
        if (queryInfo.getInstanceName() == null) {
            instanceName = getInstanceNameForClinic(
                    Integer.parseInt(queryInfo.getKeyValue())); // You need to implement this method
        }else{
            instanceName = queryInfo.getInstanceName();
        }
        // Determine the connection details based on instance name
        String jdbcURL = null;
        if ("instance1".equalsIgnoreCase(instanceName)) {
            jdbcURL = vmysql1;
        } else if ("instance2".equalsIgnoreCase(instanceName)) {
            jdbcURL = vmysql2;
        } else {
            System.out.println("Invalid instance name.");
            return;
        }
```

```
        String queryToExecute = removeInstanceName(query);

        // Connect to the database
        Connection connection = DriverManager.getConnection(jdbcURL, cloud_username, cloud_password);

        // Execute the query
        Statement statement = connection.createStatement();
        statement.execute("use sprint2");
        ResultSet resultSet = statement.executeQuery(queryToExecute);
        ResultSetMetaData rsmd = resultSet.getMetaData();
        int columnCount = rsmd.getColumnCount();

        // Print column names
        for (int i = 1; i <= columnCount; i++) {
            System.out.print(rsmd.getColumnName(i) + "\t");
        }
        System.out.println(); // Move to next line for data

        // Print data
        while (resultSet.next()) {
            for (int i = 1; i <= columnCount; i++) {
                System.out.print(resultSet.getString(i) + "\t");
            }
            System.out.println(); // Move to next line for next row
        }

        // Close connections
        resultSet.close();
        statement.close();
        connection.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

2. executeClinicQuery(String query):

This method handles executing a clinic-related query, considering possible scenarios with multiple database instances:

- **Query Parsing:** It calls the parseQuery method to extract valuable information from the user-provided query string. This includes the table name, instance name (if specified), key used in the WHERE clause, and its value.
- **Instance Name Handling:** If the query doesn't explicitly mention an instance name, it retrieves the instance name associated with the clinic ID in the WHERE clause using getInstanceNameForClinic.
- **JDBC URL Selection:** Based on the extracted instance name (instance1 or instance2), it determines the appropriate JDBC URL for the database connection.
- **Query Modification:** It removes the instance name from the original query string (if present) as it's already considered for connection selection.
- **Database Connection:** It establishes a connection to the chosen database instance using the retrieved JDBC URL and credentials.

- **Query Execution and Result Display:** It executes the modified query on the connected database. It then retrieves and prints the resulting data on the console using JDBC methods.
- **Connection Closing:** After successful execution, it closes the database connection to release resources.

```
public static QueryInfo parseQuery(String query) {
    QueryInfo info = new QueryInfo();

    // Define the regex pattern to match table name, instance name, key, and value
    Pattern pattern = Pattern.compile("from\\s*(\\S+)\\s*where\\s*(\\S+)\\s*=\\s*(\\d+)");
    Matcher matcher = pattern.matcher(query);

    if (matcher.find()) {
        String tableName = matcher.group(1);
        String instanceName = null;

        // Extract instance name if available
        String[] parts = tableName.split("\\.");
        if (parts.length > 1) {
            instanceName = parts[0];
            tableName = parts[1];
        }

        String key = matcher.group(2);
        String value = matcher.group(3);

        info.setTableName(tableName);
        info.setInstanceName(instanceName);
        info.setKeyName(key);
        info.setKeyValue(value);

    } else {
        System.out.println("Invalid query: " + query);
    }
    return info;
}
```

3. parseQuery(String query):

This method takes a query string and parses it to identify relevant components:

- **Regular Expression Matching:** It uses a predefined regular expression pattern to match the expected structure of the query. This pattern ensures the query follows the format "FROM table_name WHERE key = value".
- **Information Extraction:** If a match is found, it extracts the following details from the matching groups in the regular expression:
    - Table name: The table being queried (e.g., "Clinic").

    - Instance name (optional): The name of the database instance if specified in the query (e.g., "instance1").
    - Key name: The column name used in the WHERE clause (e.g., "ClinicID").
    - Value: The value used for comparison in the WHERE clause (e.g., "1").
- **QueryInfo Object:** It creates a QueryInfo object containing the extracted information and returns it. This object encapsulates the parsed details from the query string.

```
private static String getInstanceNameForClinic(int clinicId) {
    String instanceName = null;
    try (Scanner scanner = new Scanner(new File("hospital_clinic_details.txt"))) {
        scanner.nextLine();
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            String[] parts = line.split(",");
            int currentClinicId = Integer.parseInt(parts[1]);
            if (currentClinicId == clinicId) {
                instanceName = parts[9];
                break;
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Error: File not found");
    }
    return instanceName;
}
```

4. getInstanceNameForClinic(int clinicId):

This method retrieves the instance name associated with a given clinic ID:

- **File Reading:** It opens a file named "hospital_clinic_details.txt" and reads its contents line by line. This file is assumed to contain clinic information, including clinic ID and corresponding instance name.
- **Line Processing:** It iterates through each line in the file.
- **Data Extraction:** For each line, it parses the data to extract the clinic ID and instance name.
- **Matching Clinic ID:** It compares the extracted clinic ID with the provided clinicId parameter.
- **Instance Name Return:** If a match is found (clinic IDs match), it returns the corresponding instance name retrieved from the file. If no match is found, it returns null.

Meeting Logs - Sprint 2

MeetingDate	Meeting Time (Start)	Meeting Time (End)	Meeting Place	Sprint	Outcomes	Duration	Attendees
2/28/2024	5:00 PM	7:00 PM	In-person	Sprint 2	Logical Diagram Discussion	2 hours	All present
3/1/2024	1:00 PM	3:00 PM	online	Sprint 2	Logical Diagram Finalized	2 hours	All present
3/3/2024	9:00 AM	11:00 PM	online	Sprint 2	ERD Dependency Discovery	2 hours	All present
3/5/2024	10:30 AM	12:30 PM	In-person	Sprint 2	ERD Dependency resolution	2 hours	All present
3/8/2024	9:00 AM	10:00 AM	online	Sprint 2	GCP Server Setup	1 hour	All present
3/12/2024	10:30 AM	12:30 PM	online	Sprint 2	Fragmentation	2 hours	All present
3/15/2024	9:30 AM	10:30 AM	online	Sprint 2	Submission Report Finalize	1 hour	All present

References

[1] C. J. Date, N. Lorentzos, and H. Darwen, *Temporal Data & the Relational Model*, 1st ed. Morgan Kaufmann, 2002. [Online]. Available: https://www.morganclaypool.com/doi/abs/10.2200/S00355ED1V01Y200901DTM004. [Accessed: 12-Mar-2024].

[2] C. J. Date, *Introduction to Database Systems*, 8th ed. Boston: Addison-Wesley, 2004.

[3] W. Kent, "A Simple Guide to Five Normal Forms in Relational Database Theory," *Communications of the ACM*, vol. 26, no. 2, pp. 120–125, 1983. doi:10.1145/358024.358054.

[4] E. F. Codd, "Further Normalization of the Data Base Relational Model," presented at Courant Computer Science Symposia Series 6, "Database Systems," New York City, May 24th-25th, 1971. Prentice-Hall, 1972, pp. 45-51.

[5] Initial Physical Diagram, Group 8, [Online]. Available: https://drive.google.com/file/d/1i5DsU2h8Mm9euKf302Qu-cHfYE_hW8FD/view?usp=sharing

[6] Final Physical Diagram, Group 8, [Online]. Available: https://drive.google.com/file/d/1zbkz-pQrv05OcfgKZTO31vQ3WMsghYGg/view?usp=sharing

[7] "Google Cloud Storage Documentation," Google Cloud. [Online]. Available: https://cloud.google.com/storage. [Accessed: 15-Mar-2024].