ERD

## General Notes

- **UUIDs as Strings**: In these examples, UUIDs are represented as strings (`STRING` type) since Hive does not have a native UUID type.
- **ORC File Format**: The ORC file format is used for its efficient, columnar storage and performance benefits.
- **Transactional Tables**: The `TBLPROPERTIES` is set to `"transactional"="true"` to enable ACID (Atomicity, Consistency, Isolation, Durability) properties on the table, which is useful for updates and deletes.
- **Bucketing**: Each table is clustered into a number of buckets (10 in these examples), which can help in distributing the data evenly across the cluster and improving query performance.

## 1. Patient Information

```sql
CREATE TABLE PatientInformation (
    PatientID STRING,
    Name STRING,
    DateOfBirth DATE,
    Gender CHAR(1),
    Address STRING,
    ContactNumber STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
CLUSTERED BY (PatientID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Indexes**: Hive doesn't support traditional indexes. Instead, you use columnar file formats like ORC to optimize for read performance.
- **Clustering**: This table is clustered by `PatientID` to ensure that data for each patient is stored together, which can improve query performance.
- **Partitioning**: Not applied here as patient data is typically not time-based. However, if there's a specific query pattern, partitioning can be considered.

## 2. Doctor Information

```sql
CREATE TABLE DoctorInformation (
    DoctorID STRING,
    Name STRING,
    Specialty STRING,
    ContactNumber STRING,
    Department STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
CLUSTERED BY (DoctorID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Clustering**: Clustering by `DoctorID` is used for similar reasons as in the patient table.
- **Partitioning**: Like the Patient table, partitioning is not added here but can be considered based on specific query needs.

### 3. Appointment

```
CREATE TABLE Appointment (
    AppointmentID STRING,
    PatientID STRING,
    DoctorID STRING,
    AppointmentDate TIMESTAMP,
    ReasonForVisit STRING,
    AppointmentStatus STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
PARTITIONED BY (year INT, month INT, day INT)
CLUSTERED BY (AppointmentID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Partitioning**: This table is partitioned by year, month, and day of the `AppointmentDate`. This approach is beneficial for queries that are filtered by date ranges.
- **Clustering**: Clustering by `AppointmentID` to group all data related to a particular appointment together.

Creating tables for “Medical Records”, “Medication”, and “Prescription” in a Hue environment, which typically interfaces with Hive, involves considering specific aspects such as partitioning and clustering for optimized query performance. Hive does not support traditional indexing like relational databases, so the design focuses more on how data is stored and accessed.

### 1. Medical Records

```
CREATE TABLE MedicalRecords (
    RecordID STRING,
    PatientID STRING,
    VisitDate TIMESTAMP,
    Diagnosis STRING,
    Treatment STRING,
    DoctorID STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
PARTITIONED BY (year INT, month INT, day INT)
CLUSTERED BY (RecordID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Partitioning**: Partitioned by year, month, and day of `VisitDate` for efficient querying on date ranges.
- **Clustering**: Clustering by `RecordID` to keep all information about a particular medical record together.
- **ORC Format**: Using ORC for efficient, columnar storage.

### 2. Medication

```
CREATE TABLE Medication (
    MedicationID STRING,
    Name STRING,
    Type STRING,
    Manufacturer STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
CLUSTERED BY (MedicationID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Clustering**: Clustering by `MedicationID`. Since the medication table is less likely to be queried by date, it does not have a partition.
- **ORC Format**: ORC is used for storage efficiency.

### 3. Prescription

```
CREATE TABLE Prescription (
    PrescriptionID STRING,
    PatientID STRING,
    MedicationID STRING,
    Dosage STRING,
    Frequency STRING,
    StartDate DATE,
    EndDate DATE,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
```

```
PARTITIONED BY (startDateYear INT, startDateMonth INT, startDateDay INT)
CLUSTERED BY (PrescriptionID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Partitioning**: Partitioned by year, month, and day of `StartDate` to optimize queries related to prescription timings.
- **Clustering**: Clustering by `PrescriptionID` ensures that all data about a particular prescription is stored together.
- **ORC Format**: The table uses the ORC format for better performance in read-heavy operations.

Creating tables for “Lab Tests”, “Billing Information”, and “Insurance Information” in a Hue environment, which typically interfaces with Hive, can be optimized through careful considerations of partitioning and clustering. Here are the SQL queries for creating these tables, keeping in mind that Hive does not support traditional SQL indexing:

### 1. Lab Tests

```
CREATE TABLE LabTests (
    TestID STRING,
    PatientID STRING,
    TestType STRING,
    TestDate TIMESTAMP,
    Result STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
PARTITIONED BY (year INT, month INT, day INT)
CLUSTERED BY (TestID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Partitioning**: Partitioned by year, month, and day of `TestDate`. This is beneficial for queries filtering by specific test dates.
- **Clustering**: Clustering by `TestID` to keep related data together, enhancing data retrieval performance.
- **ORC Format**: Using ORC for efficient storage and faster read operations.

### 2. Billing Information

```
CREATE TABLE BillingInformation (
    BillID STRING,
    PatientID STRING,
    DateIssued TIMESTAMP,
    TotalAmount DECIMAL(10,2),
    PaymentStatus STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
PARTITIONED BY (year INT, month INT)
CLUSTERED BY (BillID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Partitioning**: Partitioned by year and month of `DateIssued` to improve performance for queries based on billing periods.
- **Clustering**: Clustering by `BillID` for better data organization and retrieval.
- **ORC Format**: ORC is chosen for its efficient, columnar storage.

### 3. Insurance Information

```
CREATE TABLE InsuranceInformation (
    InsuranceID STRING,
    PatientID STRING,
    Provider STRING,
    PolicyNumber STRING,
    CoverageDetails STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
CLUSTERED BY (InsuranceID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Clustering**: Clustering by `InsuranceID`. Since this table is less likely to be queried by date, it does not have a partition.
- **ORC Format**: Using ORC for its storage efficiency and read performance.

Certainly, here are SQL queries for creating tables for “Emergency Contact Information,” “Staff Information,” and “Equipment” in a Hue environment, with considerations for indexes, clustering, and partitioning:

### 1. Emergency Contact Information

```
CREATE TABLE EmergencyContactInformation (
    EmergencyContactID STRING,
    PatientID STRING,
    Name STRING,
    Relationship STRING,
    ContactNumber STRING,
    Address STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
CLUSTERED BY (EmergencyContactID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Clustering**: Clustering by `EmergencyContactID` to group related emergency contact information together, improving query performance.
- **ORC Format**: Utilizing ORC for efficient storage and retrieval.

**2. Staff Information**

```
CREATE TABLE StaffInformation (
    StaffID STRING,
    Name STRING,
    Position STRING,
    ContactNumber STRING,
    Department STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
CLUSTERED BY (StaffID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Clustering**: Clustering by `StaffID` for organized data storage and faster access.
- **ORC Format**: Employing ORC for optimized columnar storage.

**3. Equipment**

```
CREATE TABLE Equipment (
    EquipmentID STRING,
    Name STRING,
    Type STRING,
    Status STRING,
    Department STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
CLUSTERED BY (EquipmentID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Clustering**: Clustering by `EquipmentID` to maintain related equipment data in proximity.
- **ORC Format**: Using ORC for efficient, columnar storage.

Certainly, here are SQL queries for creating tables for “Room and Facility Usage,” “Procedure Information,” and “Patient Procedures” in a Hue environment, with considerations for indexes, clustering, and partitioning:

### 1. Room and Facility Usage

```
CREATE TABLE RoomAndFacilityUsage (
    RoomID STRING,
    Type STRING,
    Status STRING,
    CurrentOccupant STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
PARTITIONED BY (year INT, month INT, day INT)
CLUSTERED BY (RoomID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Partitioning**: Partitioned by year, month, and day for efficient querying based on dates.
- **Clustering**: Clustering by `RoomID` to group related data together, improving query performance.
- **ORC Format**: Utilizing ORC for efficient storage and retrieval.

### 2. Procedure Information

```
CREATE TABLE ProcedureInformation (
    ProcedureID STRING,
    Name STRING,
    Description STRING,
    StandardRecoveryTime INT,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
CLUSTERED BY (ProcedureID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Clustering**: Clustering by `ProcedureID` to keep related procedure information together for faster access.
- **ORC Format**: Employing ORC for optimized columnar storage.

### 3. Patient Procedures

```
CREATE TABLE PatientProcedures (
    PatientProcedureID STRING,
    PatientID STRING,
    ProcedureID STRING,
    ProcedureDate TIMESTAMP,
    PerformingDoctorID STRING,
    Outcome STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
PARTITIONED BY (year INT, month INT)
CLUSTERED BY (PatientProcedureID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Partitioning**: Partitioned by year and month of `ProcedureDate` for efficient queries based on procedure dates.
- **Clustering**: Clustering by `PatientProcedureID` to group related patient procedure data together.
- **ORC Format**: Using ORC for efficient columnar storage.

Certainly, here are SQL queries for creating tables for "Immunization Records," "Disease Registry," "Healthcare Provider Information," and "Patient Visits History" in a Hue environment, with considerations for indexes, clustering, and partitioning:

### 1. Immunization Records

```
CREATE TABLE ImmunizationRecords (
    ImmunizationID STRING,
    PatientID STRING,
    VaccineName STRING,
    DateAdministered DATE,
    AdministeredBy STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
PARTITIONED BY (year INT, month INT)
CLUSTERED BY (ImmunizationID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Partitioning**: Partitioned by year and month of `DateAdministered` for efficient queries based on immunization dates.
- **Clustering**: Clustering by `ImmunizationID` to group related immunization records together for faster access.
- **ORC Format**: Utilizing ORC for efficient storage and retrieval.

### 2. Disease Registry

```sql
CREATE TABLE DiseaseRegistry (
    DiseaseID STRING,
    Name STRING,
    Description STRING,
    Category STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
CLUSTERED BY (DiseaseID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Clustering**: Clustering by `DiseaseID` to maintain related disease registry data in proximity for better query performance.
- **ORC Format**: Employing ORC for optimized columnar storage.

### 3. Healthcare Provider Information

```sql
CREATE TABLE HealthcareProviderInformation (
    ProviderID STRING,
    Name STRING,
    Type STRING,
    Address STRING,
    ContactNumber STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
CLUSTERED BY (ProviderID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Clustering**: Clustering by `ProviderID` to group healthcare provider information for improved data retrieval.
- **ORC Format**: Using ORC for efficient columnar storage.

### 4. Patient Visits History

```sql
CREATE TABLE PatientVisitsHistory (
    VisitID STRING,
    PatientID STRING,
    VisitDate TIMESTAMP,
    VisitType STRING,
    Outcome STRING,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
)
```

```
PARTITIONED BY (year INT, month INT, day INT)
CLUSTERED BY (VisitID) INTO 10 BUCKETS
STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

- **Partitioning**: Partitioned by year, month, and day of `VisitDate` for efficient queries on visit dates.
- **Clustering**: Clustering by `VisitID` to group related patient visit data together for improved query performance.
- **ORC Format**: Employing ORC for efficient columnar storage.

Creating a complex Hue query to build a data warehouse involves combining data from multiple tables. Below is an example of a complex query that retrieves and joins data from all the tables mentioned above to create a comprehensive healthcare data warehouse. This query assumes that the tables have already been created in the database.

```
-- Create a healthcare data warehouse by joining data from multiple tables

-- Patient Information
WITH PatientInfo AS (
    SELECT
        PatientID,
        Name AS PatientName,
        DateOfBirth,
        Gender,
        Address AS PatientAddress,
        ContactNumber AS PatientContact
    FROM PatientInformation
),

-- Doctor Information
DoctorInfo AS (
    SELECT
        DoctorID,
        Name AS DoctorName,
        Specialty AS DoctorSpecialty,
        ContactNumber AS DoctorContact,
        Department AS DoctorDepartment
    FROM DoctorInformation
),

-- Medical Records
MedicalRecordsInfo AS (
    SELECT
        PatientID,
        VisitDate,
        Diagnosis,
        Treatment,
        DoctorID AS AttendingDoctorID
    FROM MedicalRecords
),

-- Room and Facility Usage
RoomFacilityInfo AS (
    SELECT
        RoomID,
        Type AS RoomType,
        Status AS RoomStatus,
        CurrentOccupant AS OccupantPatientID
    FROM RoomAndFacilityUsage
),

-- Procedure Information
ProcedureInfo AS (
    SELECT
        ProcedureID,
        Name AS ProcedureName,
        Description AS ProcedureDescription,
        StandardRecoveryTime AS ProcedureRecoveryTime
    FROM ProcedureInformation
),

-- Patient Procedures
PatientProceduresInfo AS (
    SELECT
        PatientID AS ProcedurePatientID,
        ProcedureID AS ProcedurePerformedID,
        ProcedureDate AS ProcedurePerformedDate,
        PerformingDoctorID AS ProcedurePerformingDoctorID,
        Outcome AS ProcedureOutcome
    FROM PatientProcedures
),

-- Immunization Records
ImmunizationInfo AS (
    SELECT
        PatientID AS ImmunizationPatientID,
        VaccineName AS ImmunizationVaccineName,
        DateAdministered AS ImmunizationDateAdministered,
        AdministeredBy AS ImmunizationAdministeredBy
    FROM ImmunizationRecords
),
```

```sql
-- Disease Registry
DiseaseRegistryInfo AS (
    SELECT
        DiseaseID,
        Name AS DiseaseName,
        Description AS DiseaseDescription,
        Category AS DiseaseCategory
    FROM DiseaseRegistry
),

-- Healthcare Provider Information
ProviderInfo AS (
    SELECT
        ProviderID,
        Name AS ProviderName,
        Type AS ProviderType,
        Address AS ProviderAddress,
        ContactNumber AS ProviderContact
    FROM HealthcareProviderInformation
),

-- Patient Visits History
PatientVisitsInfo AS (
    SELECT
        PatientID AS VisitPatientID,
        VisitDate AS VisitDate,
        VisitType AS VisitType,
        Outcome AS VisitOutcome
    FROM PatientVisitsHistory
)

-- Final Query: Joining all the information together
SELECT
    PI.*,
    DI.*,
    MR.*,
    RF.*,
    PR.*,
    PP.*,
    II.*,
    DR.*,
    HI.*,
    PV.*
FROM PatientInfo PI
JOIN DoctorInfo DI ON PI.PatientID = MR.PatientID
JOIN MedicalRecordsInfo MR ON DI.DoctorID = MR.AttendingDoctorID
LEFT JOIN RoomFacilityInfo RF ON MR.PatientID = RF.OccupantPatientID
LEFT JOIN ProcedureInfo PR ON MR.Diagnosis = PR.ProcedureName
LEFT JOIN PatientProceduresInfo PP ON MR.PatientID = PP.ProcedurePatientID
LEFT JOIN ImmunizationInfo II ON MR.PatientID = II.ImmunizationPatientID
LEFT JOIN DiseaseRegistryInfo DR ON MR.Diagnosis = DR.DiseaseName
LEFT JOIN ProviderInfo HI ON DI.DoctorID = HI.ProviderID
LEFT JOIN PatientVisitsInfo PV ON MR.PatientID = PV.VisitPatientID;
```

This query combines data from all the tables and performs JOIN operations to create a comprehensive healthcare data warehouse. Make sure to replace table and column names with the actual names used in your database. Additionally, you can further customize this query to meet your specific reporting and analysis requirements.

Certainly, let's create more advanced and robust SQL queries for generating dimension tables and a fact table. We will enhance the queries with additional features such as handling NULL values, adding computed columns, and providing more advanced data transformations.

## 1. Dimension Tables:

### Dimension Patient

```sql
CREATE TABLE DimensionPatient AS
SELECT
    PatientID,
    Name AS PatientName,
    DateOfBirth,
    Gender,
    Address AS PatientAddress,
    ContactNumber AS PatientContactNumber,
    COALESCE(created_at, NOW()) AS PatientCreatedDate,
    COALESCE(updated_at, NOW()) AS PatientUpdatedDate
FROM
    PatientInformation;
```

### Dimension Doctor

```sql
CREATE TABLE DimensionDoctor AS
SELECT
    DoctorID,
    Name AS DoctorName,
    Specialty AS DoctorSpecialty,
    ContactNumber AS DoctorContactNumber,
    Department AS DoctorDepartment,
    COALESCE(created_at, NOW()) AS DoctorCreatedDate,
    COALESCE(updated_at, NOW()) AS DoctorUpdatedDate
```

```sql
FROM
    DoctorInformation;
```

### Dimension Medical

```sql
CREATE TABLE DimensionMedical AS
SELECT
    RecordID AS MedicalRecordID,
    PatientID AS PatientIDMedical,
    VisitDate AS MedicalVisitDate,
    Diagnosis AS MedicalDiagnosis,
    Treatment AS MedicalTreatment,
    DoctorID AS DoctorIDMedical,
    COALESCE(created_at, NOW()) AS MedicalRecordCreatedDate,
    COALESCE(updated_at, NOW()) AS MedicalRecordUpdatedDate
FROM
    MedicalRecords;
```

### Dimension Billing

```sql
CREATE TABLE DimensionBilling AS
SELECT
    BillID AS BillingBillID,
    PatientID AS PatientIDBilling,
    DateIssued AS BillingDateIssued,
    TotalAmount AS BillingTotalAmount,
    PaymentStatus AS BillingPaymentStatus,
    COALESCE(created_at, NOW()) AS BillingCreatedDate,
    COALESCE(updated_at, NOW()) AS BillingUpdatedDate
FROM
    BillingInformation;
```

### Dimension Finance

```sql
CREATE TABLE DimensionFinance AS
SELECT
    BillID AS FinanceBillID,
    PatientID AS FinancePatientID,
    DateIssued AS FinanceDateIssued,
    TotalAmount AS FinanceTotalAmount,
    PaymentStatus AS FinancePaymentStatus,
    COALESCE(created_at, NOW()) AS FinanceCreatedDate,
    COALESCE(updated_at, NOW()) AS FinanceUpdatedDate
FROM
    BillingInformation;
```

**Dimension Insurance**

```
CREATE TABLE DimensionInsurance AS
SELECT
    InsuranceID AS InsuranceInsuranceID,
    PatientID AS InsurancePatientID,
    Provider AS InsuranceProvider,
    PolicyNumber AS InsurancePolicyNumber,
    CoverageDetails AS InsuranceCoverageDetails,
    COALESCE(created_at, NOW()) AS InsuranceCreatedDate,
    COALESCE(updated_at, NOW()) AS InsuranceUpdatedDate
FROM
    InsuranceInformation;
```

**Dimension Provider**

```
CREATE TABLE DimensionProvider AS
SELECT
    ProviderID AS ProviderProviderID,
    Name AS ProviderName,
    Type AS ProviderType,
    Address AS ProviderAddress,
    ContactNumber AS ProviderContactNumber,
    COALESCE(created_at, NOW()) AS ProviderCreatedDate,
    COALESCE(updated_at, NOW()) AS ProviderUpdatedDate
FROM
    HealthcareProviderInformation;
```

**Additional Dimension (You can adapt this for any other potential dimension)**

```
CREATE TABLE DimensionAdditional AS
SELECT
    AdditionalID,
    AdditionalField1 AS AdditionalField1Name,
    AdditionalField2 AS AdditionalField2Name,
    COALESCE(created_at, NOW()) AS AdditionalCreatedDate,
    COALESCE(updated_at, NOW()) AS AdditionalUpdatedDate
FROM
    AdditionalTable;
```

### 2. Fact Table

Assuming you want to create a fact table that combines data from multiple dimension tables:

```
CREATE TABLE FactTable AS
SELECT
```

```
    M.MedicalRecordID,
    P.PatientName,
    D.DoctorName,
    M.MedicalVisitDate,
    M.MedicalDiagnosis,
    M.MedicalTreatment,
    B.BillingTotalAmount,
    F.FinanceTotalAmount,
    I.InsuranceCoverageDetails,
    PR.ProviderName,
    A.AdditionalField1Name,
    A.AdditionalField2Name,
    COALESCE(M.MedicalRecordCreatedDate, NOW()) AS MedicalRecordCreatedDate,
    COALESCE(M.MedicalRecordUpdatedDate, NOW()) AS MedicalRecordUpdatedDate
FROM
    DimensionMedical AS M
JOIN
    DimensionPatient AS P ON M.PatientIDMedical = P.PatientID
JOIN
    DimensionDoctor AS D ON M.DoctorIDMedical = D.DoctorID
LEFT JOIN
    DimensionBilling AS B ON M.MedicalRecordID = B.BillingBillID
LEFT JOIN
    DimensionFinance AS F ON M.MedicalRecordID = F.FinanceBillID
LEFT JOIN
    DimensionInsurance AS I ON M.MedicalRecordID = I.InsuranceInsuranceID
LEFT JOIN
    DimensionProvider AS PR ON M.DoctorIDMedical = PR.ProviderProviderID
LEFT JOIN
    DimensionAdditional AS A ON M.MedicalRecordID = A.AdditionalID;
```

These enhanced queries include features like handling NULL values and computing date columns for better data quality and accuracy. You can further customize and optimize these queries based on your specific data requirements and schema.

Certainly, let's create an even more detailed, advanced, complex, and robust Hue query for generating a data warehouse. In this query, we'll include advanced features like data transformations, aggregation, and the creation of advanced analytical views. We'll also optimize the query for performance. Please note that this is a comprehensive example, and you can further tailor it to your specific use case.

```
-- Create and populate dimension tables
CREATE TABLE IF NOT EXISTS DimensionPatient AS
SELECT
    PatientID,
    Name AS PatientName,
    DateOfBirth,
    Gender,
    Address AS PatientAddress,
    ContactNumber AS PatientContactNumber,
    COALESCE(created_at, NOW()) AS PatientCreatedDate,
    COALESCE(updated_at, NOW()) AS PatientUpdatedDate
FROM
    PatientInformation;

-- Create and populate other dimension tables similarly

-- Create and populate the fact table with aggregations
CREATE TABLE IF NOT EXISTS FactTable AS
SELECT
    M.MedicalRecordID,
    P.PatientName,
    D.DoctorName,
    M.MedicalVisitDate,
    M.MedicalDiagnosis,
    M.MedicalTreatment,
    B.BillingTotalAmount,
    F.FinanceTotalAmount,
    I.InsuranceCoverageDetails,
    PR.ProviderName,
    A.AdditionalField1Name,
    A.AdditionalField2Name,
    COALESCE(M.MedicalRecordCreatedDate, NOW()) AS MedicalRecordCreatedDate,
    COALESCE(M.MedicalRecordUpdatedDate, NOW()) AS MedicalRecordUpdatedDate
FROM
    DimensionMedical AS M
JOIN
    DimensionPatient AS P ON M.PatientIDMedical = P.PatientID
JOIN
    DimensionDoctor AS D ON M.DoctorIDMedical = D.DoctorID
LEFT JOIN
    DimensionBilling AS B ON M.MedicalRecordID = B.BillingBillID
LEFT JOIN
    DimensionFinance AS F ON M.MedicalRecordID = F.FinanceBillID
LEFT JOIN
    DimensionInsurance AS I ON M.MedicalRecordID = I.InsuranceInsuranceID
LEFT JOIN
    DimensionProvider AS PR ON M.DoctorIDMedical = PR.ProviderProviderID
LEFT JOIN
    DimensionAdditional AS A ON M.MedicalRecordID = A.AdditionalID
GROUP BY
```

```sql
    M.MedicalRecordID,
    P.PatientName,
    D.DoctorName,
    M.MedicalVisitDate,
    M.MedicalDiagnosis,
    M.MedicalTreatment,
    B.BillingTotalAmount,
    F.FinanceTotalAmount,
    I.InsuranceCoverageDetails,
    PR.ProviderName,
    A.AdditionalField1Name,
    A.AdditionalField2Name,
    MedicalRecordCreatedDate,
    MedicalRecordUpdatedDate;

-- Create advanced analytical views
CREATE VIEW IF NOT EXISTS PatientVisitSummary AS
SELECT
    PatientName,
    COUNT(MedicalRecordID) AS VisitCount,
    AVG(BillingTotalAmount) AS AvgBillingAmount
FROM
    FactTable
GROUP BY
    PatientName;

-- Create an additional aggregate view as needed

-- Create indexes for performance optimization
CREATE INDEX idx_fact_patient ON FactTable (PatientName);
CREATE INDEX idx_fact_doctor ON FactTable (DoctorName);
-- Create additional indexes as needed

-- Optimize the table for query performance
ANALYZE TABLE FactTable COMPUTE STATISTICS;

-- Optionally, you can create partitioning or clustering if needed
-- Example of partitioning by year and month
CREATE TABLE IF NOT EXISTS FactTablePartitioned
PARTITIONED BY (year INT, month INT)
AS
SELECT
    MedicalRecordID,
    PatientName,
    DoctorName,
    MedicalVisitDate,
    MedicalDiagnosis,
    MedicalTreatment,
    BillingTotalAmount,
    FinanceTotalAmount,
    InsuranceCoverageDetails,
    ProviderName,
    AdditionalField1Name,
    AdditionalField2Name,
    MedicalRecordCreatedDate,
    MedicalRecordUpdatedDate
FROM
    FactTable;

-- Example of clustering by MedicalRecordID
CREATE TABLE IF NOT EXISTS FactTableClustered
CLUSTERED BY (MedicalRecordID) INTO 10 BUCKETS
AS
SELECT
    MedicalRecordID,
    PatientName,
    DoctorName,
    MedicalVisitDate,
    MedicalDiagnosis,
    MedicalTreatment,
    BillingTotalAmount,
    FinanceTotalAmount,
    InsuranceCoverageDetails,
    ProviderName,
    AdditionalField1Name,
    AdditionalField2Name,
    MedicalRecordCreatedDate,
    MedicalRecordUpdatedDate
FROM
    FactTable;
```

In this advanced query:

- We aggregate data in the fact table to provide summary information, like visit counts and average billing amounts for patients.
- We create advanced analytical views (`PatientVisitSummary`) that can be used for reporting and analytics.
- We continue to optimize the data warehouse by creating indexes and optionally partitioning or clustering the data.
- We use `COALESCE` to handle NULL values in date columns, ensuring they have valid default values.
- We group the fact table data by relevant dimensions for aggregation.

Please adapt and customize this query based on your specific data warehouse requirements and performance considerations.

Creating advanced, complex, and robust data marts and cubes using materialized views in Hue involves optimizing query performance, adding more advanced analytics, and ensuring data freshness. Below is a more detailed and advanced Hue query for generating multiple deep data marts and cubes:

```sql
-- Create the first deep data mart using a materialized view
CREATE MATERIALIZED VIEW IF NOT EXISTS DataMart1 AS
SELECT
    P.PatientID,
    P.PatientName,
    D.DoctorID,
    D.DoctorName,
    M.MedicalVisitDate,
    M.MedicalDiagnosis,
    SUM(B.BillingTotalAmount) AS TotalBillingAmount,
    AVG(F.FinanceTotalAmount) AS AvgFinanceAmount,
    MAX(M.MedicalTreatment) AS MostCommonTreatment
FROM
    FactTable AS F
JOIN
    DimensionPatient AS P ON F.PatientName = P.PatientName
JOIN
    DimensionDoctor AS D ON F.DoctorName = D.DoctorName
JOIN
    DimensionMedical AS M ON F.MedicalVisitDate = M.MedicalVisitDate
GROUP BY
    P.PatientID,
    P.PatientName,
    D.DoctorID,
    D.DoctorName,
    M.MedicalVisitDate,
    M.MedicalDiagnosis;

-- Create the second deep data mart using a materialized view
CREATE MATERIALIZED VIEW IF NOT EXISTS DataMart2 AS
SELECT
    P.PatientID,
    P.PatientName,
    M.MedicalVisitDate,
    COUNT(DISTINCT D.DoctorID) AS DistinctDoctorCount,
    MAX(M.MedicalDiagnosis) AS MostFrequentDiagnosis,
    MIN(M.MedicalTreatment) AS LeastCommonTreatment
FROM
    FactTable AS F
```

```
JOIN
    DimensionPatient AS P ON F.PatientName = P.PatientName
JOIN
    DimensionMedical AS M ON F.MedicalVisitDate = M.MedicalVisitDate
GROUP BY
    P.PatientID,
    P.PatientName,
    M.MedicalVisitDate;

-- Create the first deep cube using a materialized view
CREATE MATERIALIZED VIEW IF NOT EXISTS Cube1 AS
SELECT
    P.PatientID,
    P.PatientName,
    D.DoctorID,
    D.DoctorName,
    M.MedicalVisitDate,
    M.MedicalDiagnosis,
    F.BillingPaymentStatus,
    COUNT(*) AS VisitCount,
    SUM(F.BillingTotalAmount) AS TotalBillingAmount,
    AVG(F.FinanceTotalAmount) AS AvgFinanceAmount
FROM
    FactTable AS F
JOIN
    DimensionPatient AS P ON F.PatientName = P.PatientName
JOIN
    DimensionDoctor AS D ON F.DoctorName = D.DoctorName
JOIN
    DimensionMedical AS M ON F.MedicalVisitDate = M.MedicalVisitDate
GROUP BY
    P.PatientID,
    P.PatientName,
    D.DoctorID,
    D.DoctorName,
    M.MedicalVisitDate,
    M.MedicalDiagnosis,
    F.BillingPaymentStatus
WITH CUBE;

-- Create the second deep cube using a materialized view
CREATE MATERIALIZED VIEW IF NOT EXISTS Cube2 AS
SELECT
    D.DoctorID,
    D.DoctorName,
    M.MedicalVisitDate,
    M.MedicalDiagnosis,
    P.Gender,
    COUNT(*) AS VisitCount,
    SUM(F.BillingTotalAmount) AS TotalBillingAmount
FROM
    FactTable AS F
JOIN
    DimensionDoctor AS D ON F.DoctorName = D.DoctorName
JOIN
    DimensionMedical AS M ON F.MedicalVisitDate = M.MedicalVisitDate
JOIN
    DimensionPatient AS P ON F.PatientName = P.PatientName
GROUP BY
    D.DoctorID,
    D.DoctorName,
    M.MedicalVisitDate,
    M.MedicalDiagnosis,
    P.Gender
WITH ROLLUP;

-- Refresh the materialized views to populate them with data
REFRESH MATERIALIZED VIEW DataMart1;
REFRESH MATERIALIZED VIEW DataMart2;
REFRESH MATERIALIZED VIEW Cube1;
REFRESH MATERIALIZED VIEW Cube2;
```

In this advanced query:

- We join the fact table with dimension tables to enrich the data marts and cubes with additional attributes.
- Aggregation functions, such as `SUM`, `AVG`, `MAX`, and `MIN`, are used to calculate various metrics.
- We use the `WITH CUBE` and `WITH ROLLUP` clauses to create advanced cubes with multiple dimensions.
- Data marts and cubes include more detailed attributes and metrics, providing a richer analytical environment.
- The `REFRESH MATERIALIZED VIEW` statements ensure that the materialized views are updated with the latest data.

Please adapt and customize this query to your specific data mart and cube requirements, including the dimensions, metrics, and aggregations that are relevant to your healthcare analytics.

Open in Colab