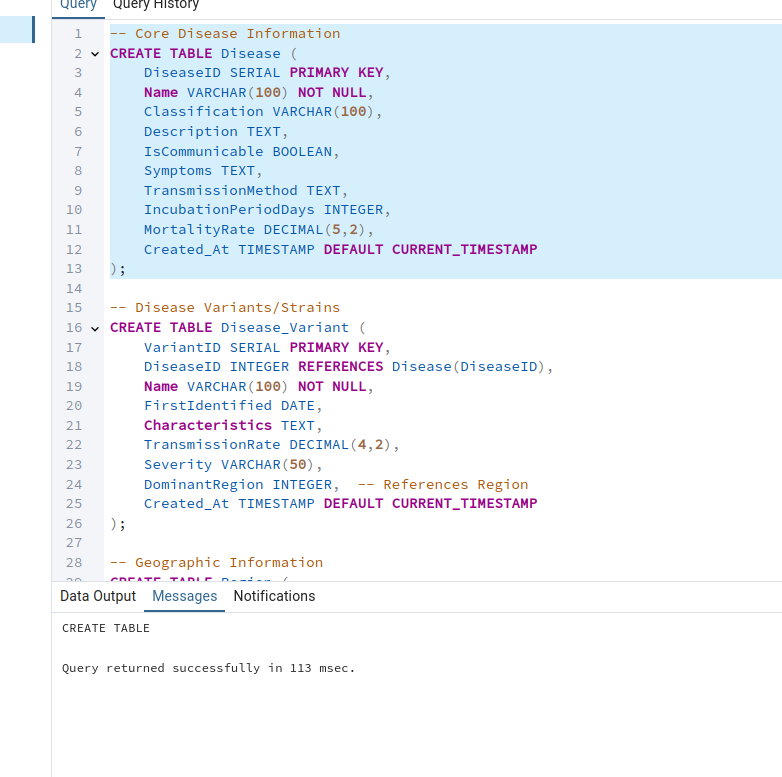
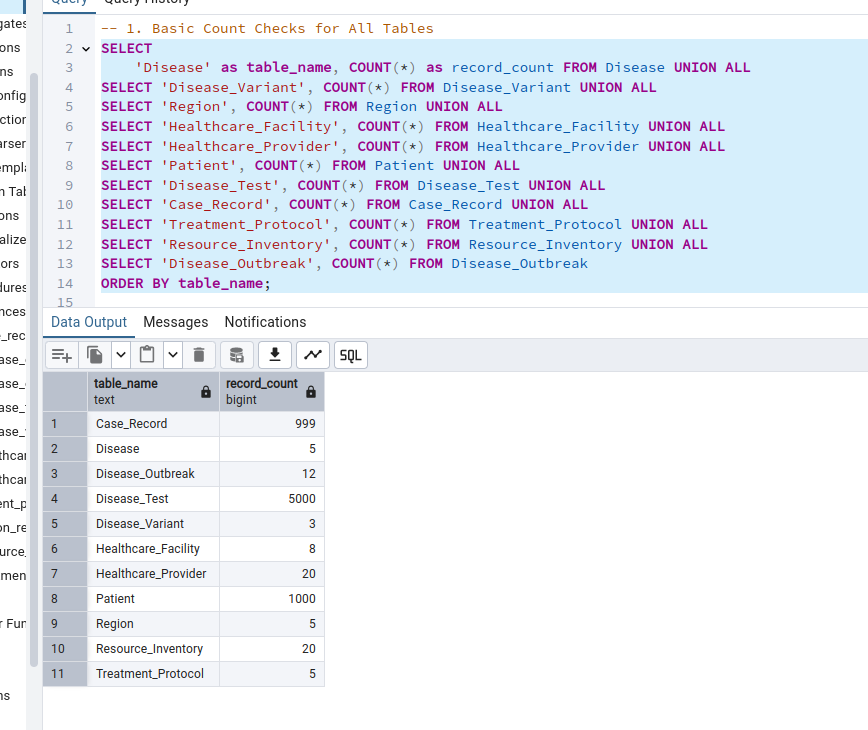
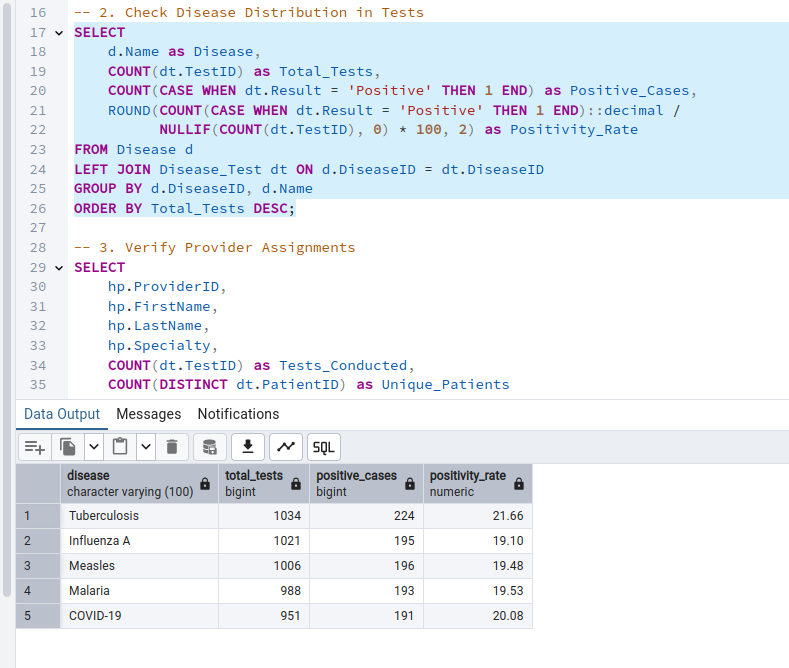
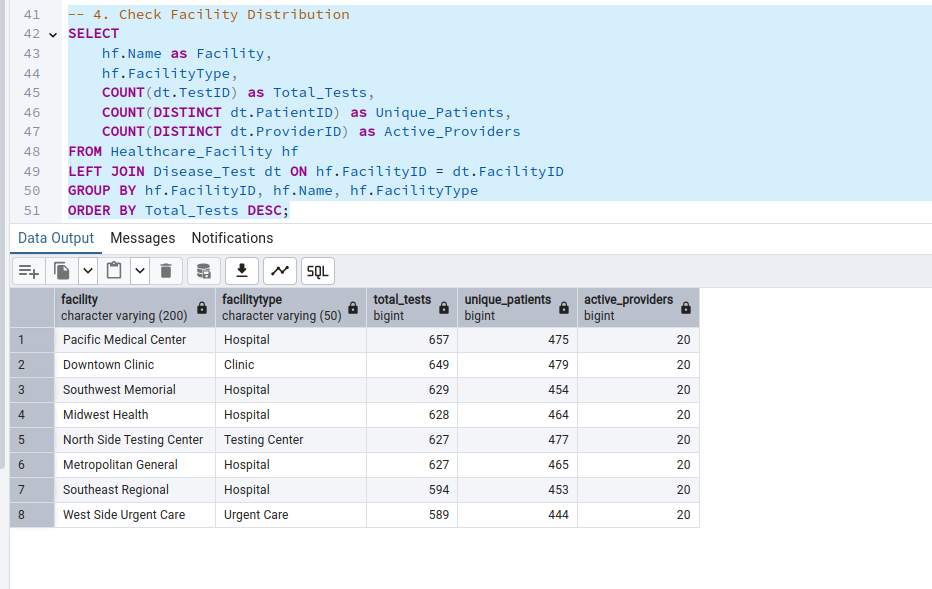
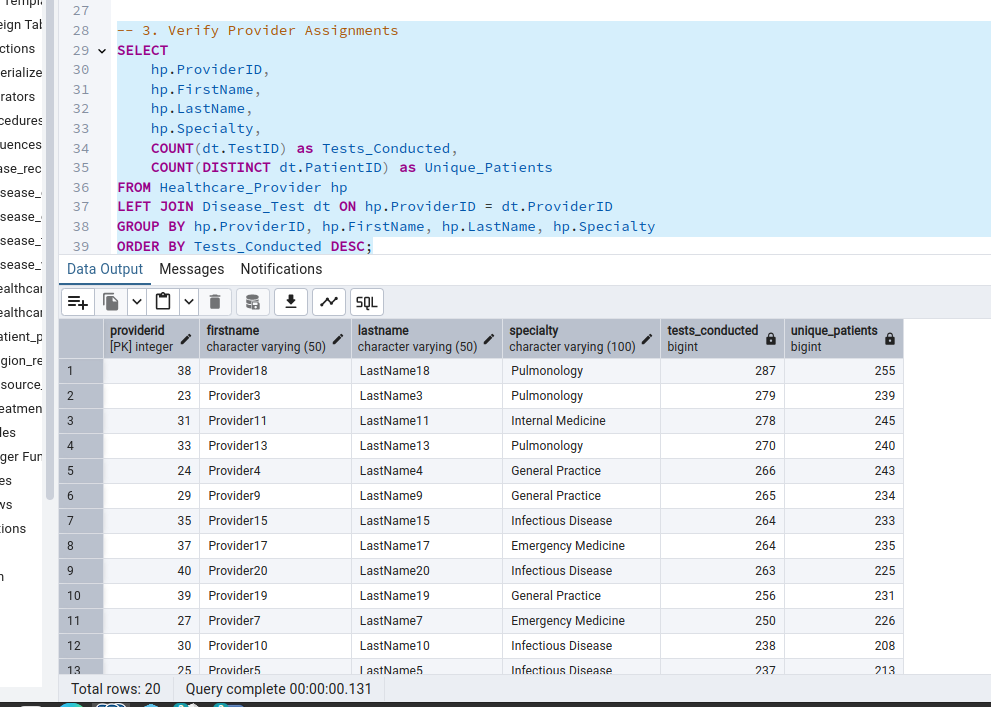
Creating tabels and indexes  
  
  
  
  
....  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
insyertong data  
  
  
  
  
  
  
  
generating data  
  
  
  
  
.....

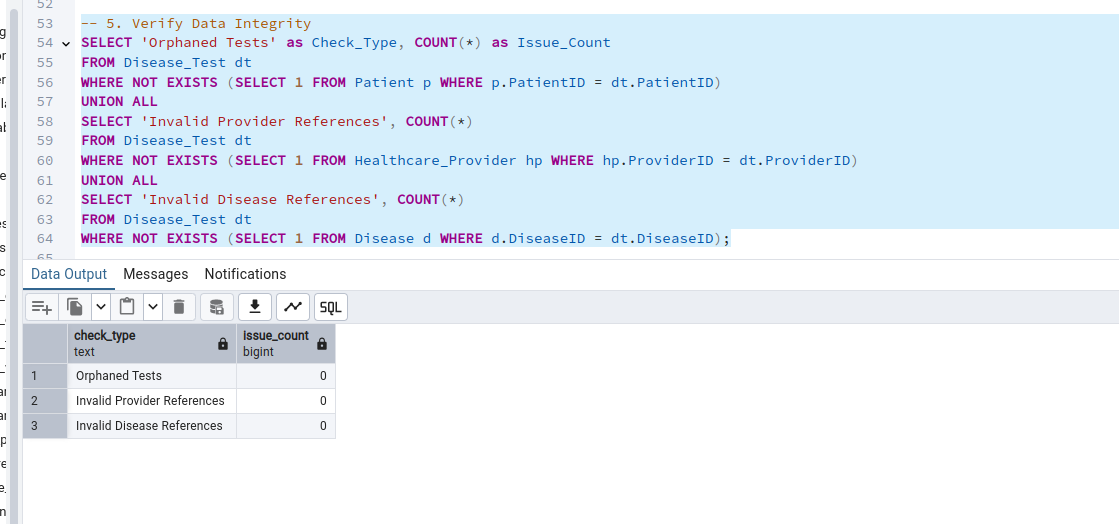
Verification

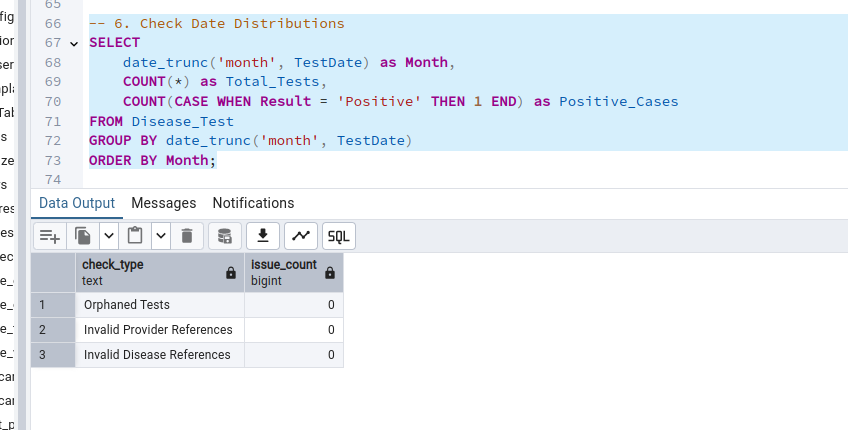
....

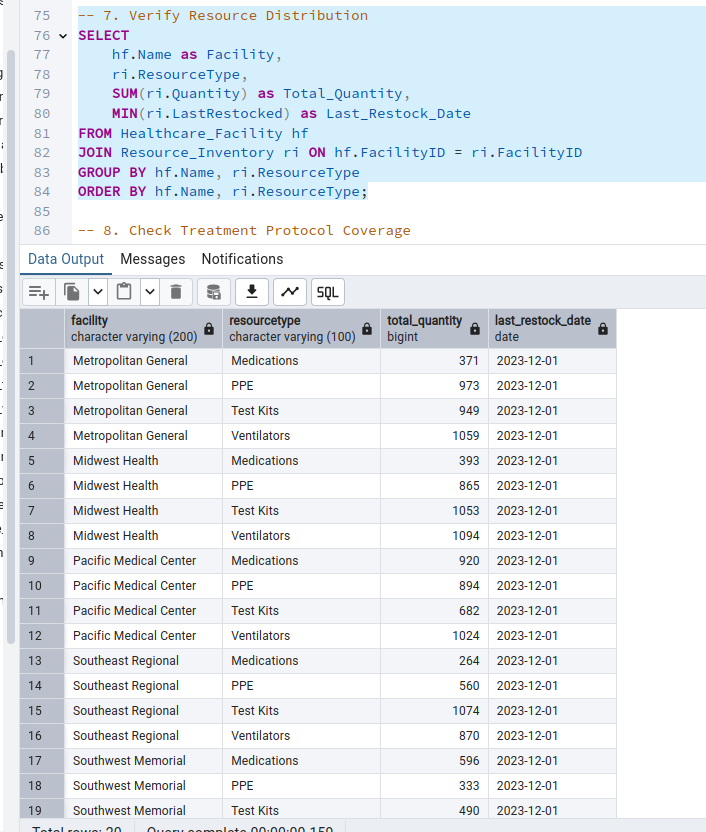
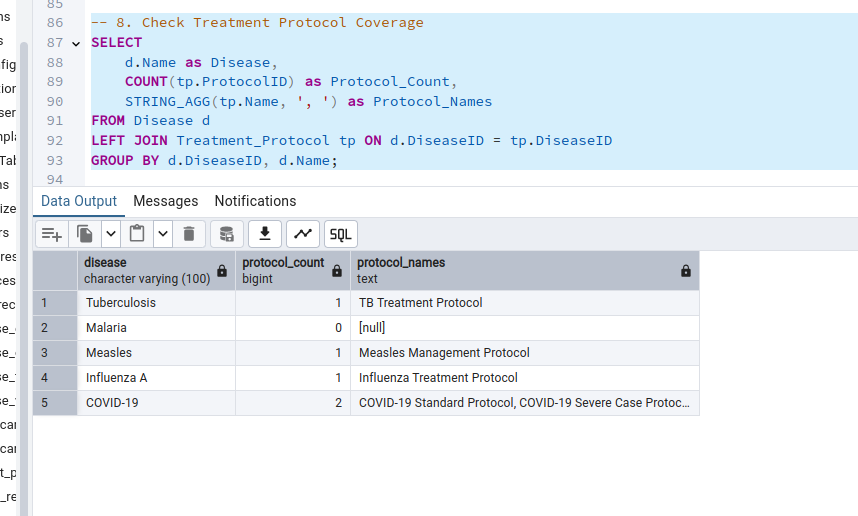
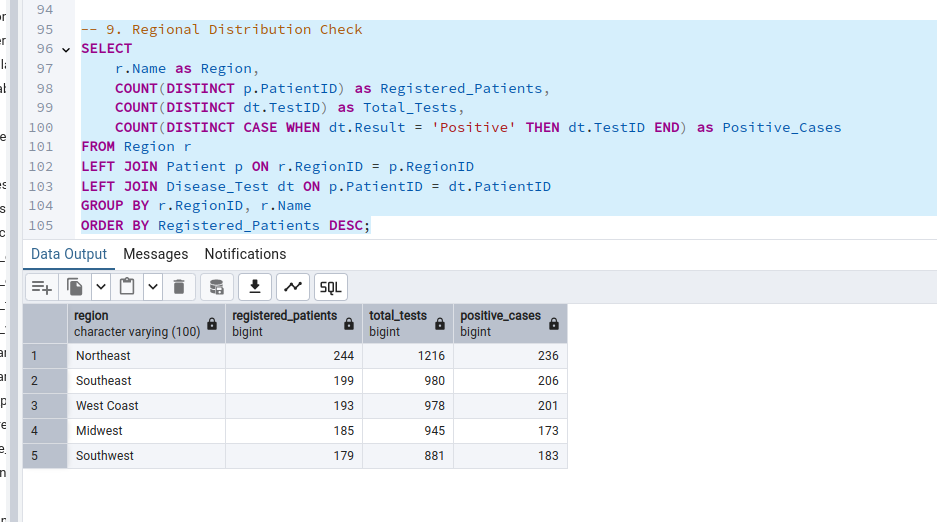










Creating Operational Queries and DML Operations.   
  
Disease Outbreak Tracking:  
  

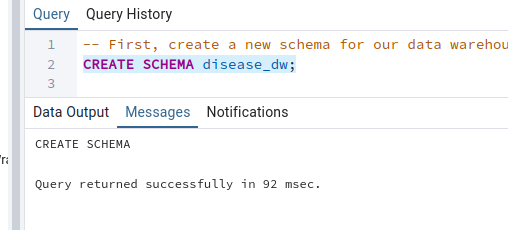
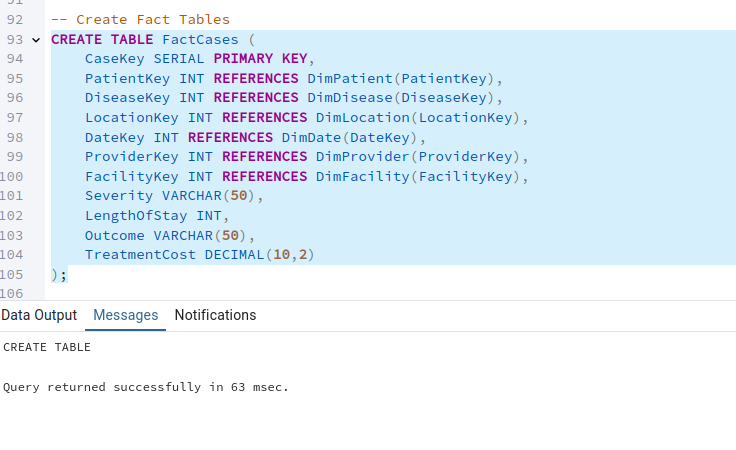

Patient Case Management:  


demonstrate referential integrity scenarios that show how our database handles related records across tables, including cascade effects and constraint enforcement.

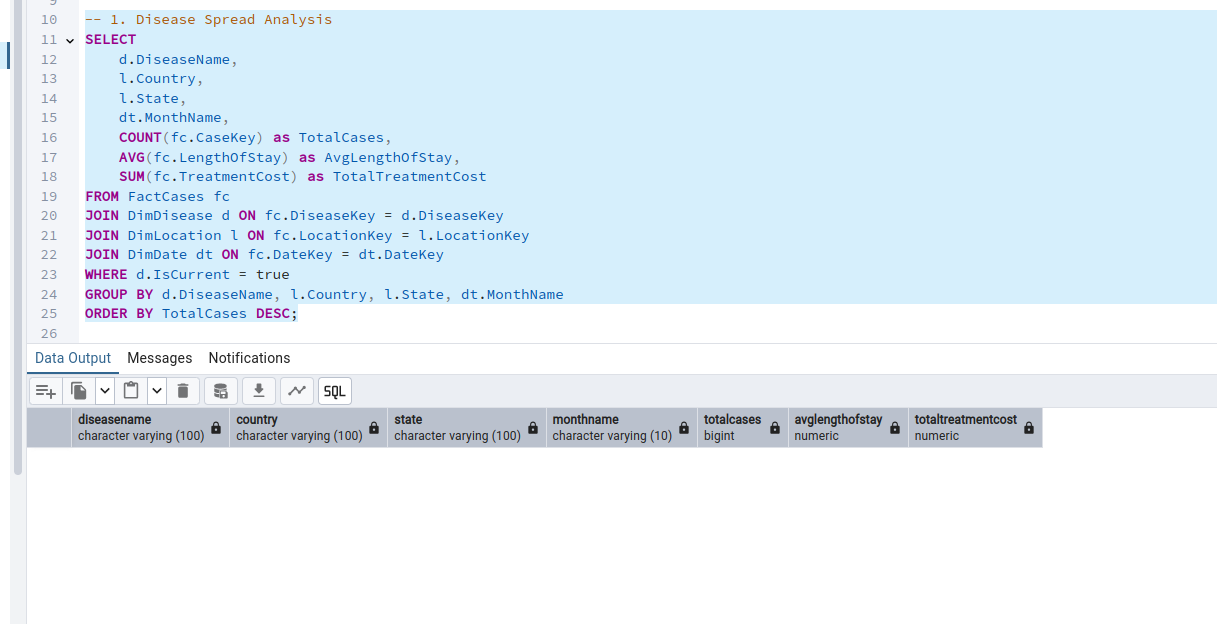


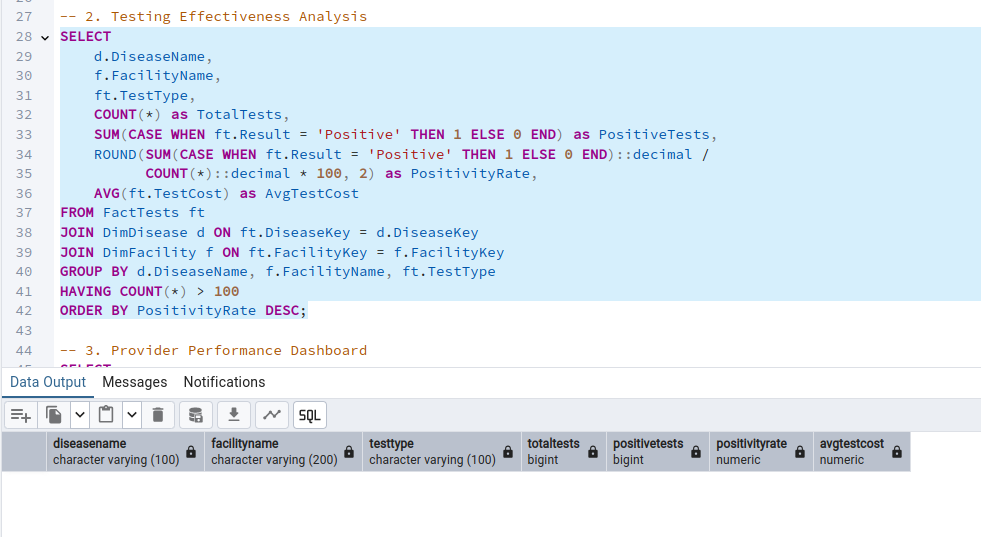
.....

creating the dimensional model (data warehouse) for analytical purposes

  
  
.....s

...

Some analytical queries that demonstrate the power of our dimensional model.   
  




...

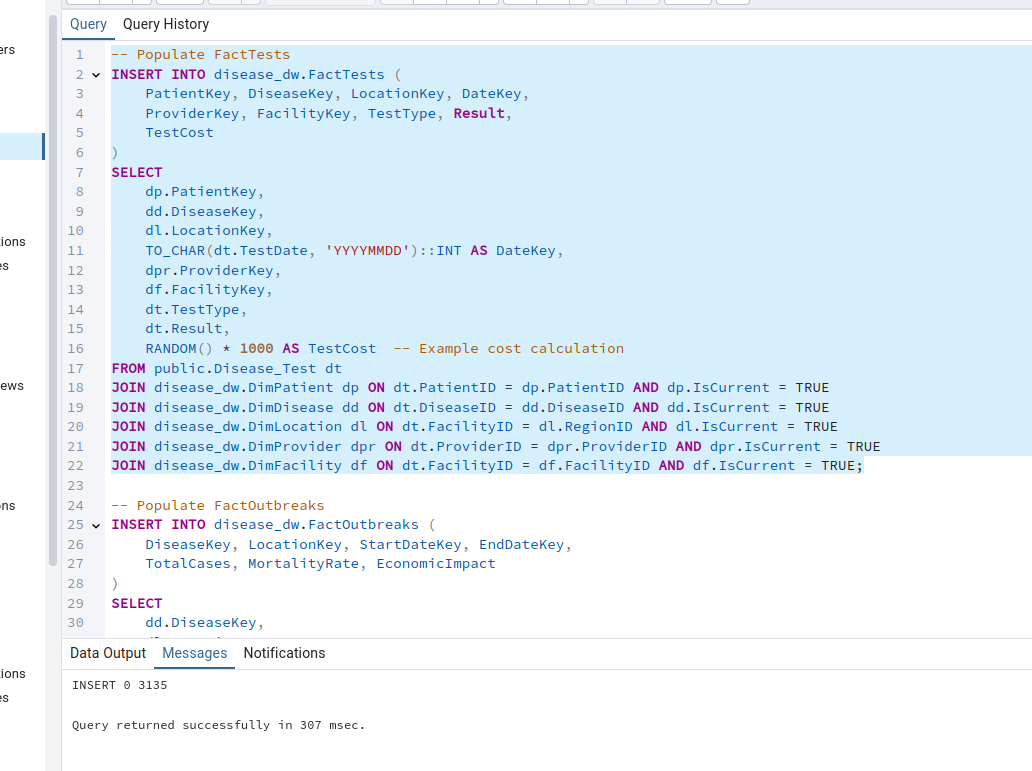
populate the dimension tables  


..

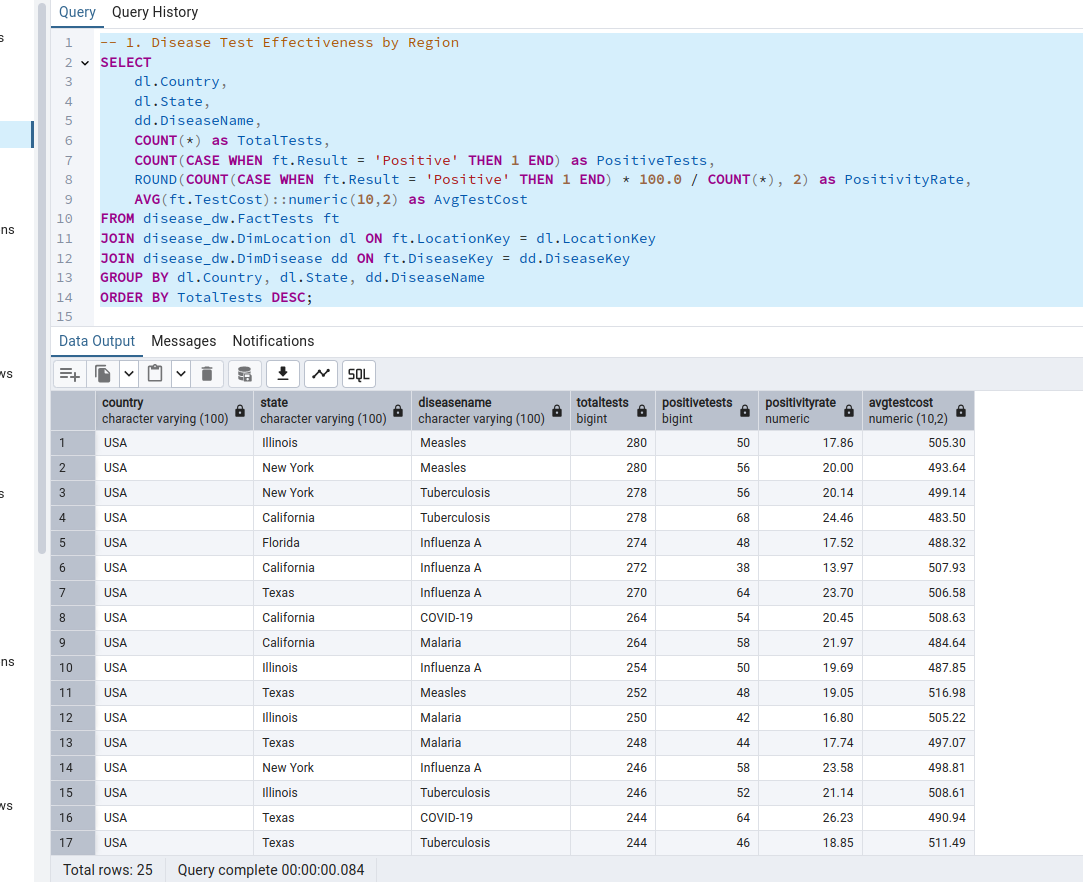


\..

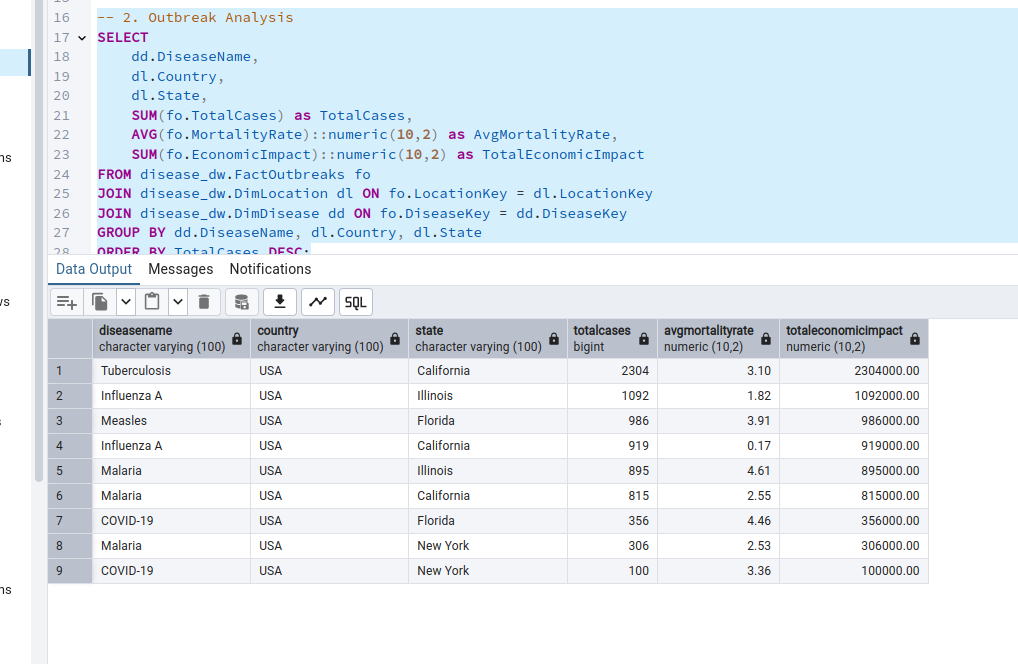
populate the fact tables  
  

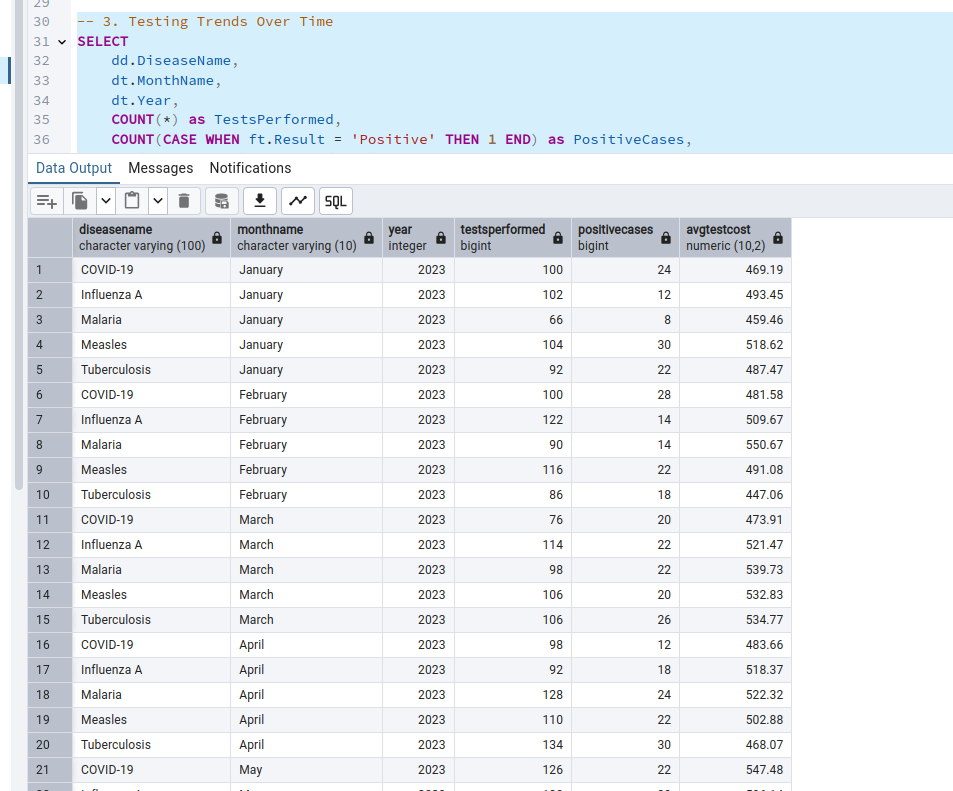



..  
  
..  
a series of analytical queries to test our data warehouse and reveal meaningful insights:

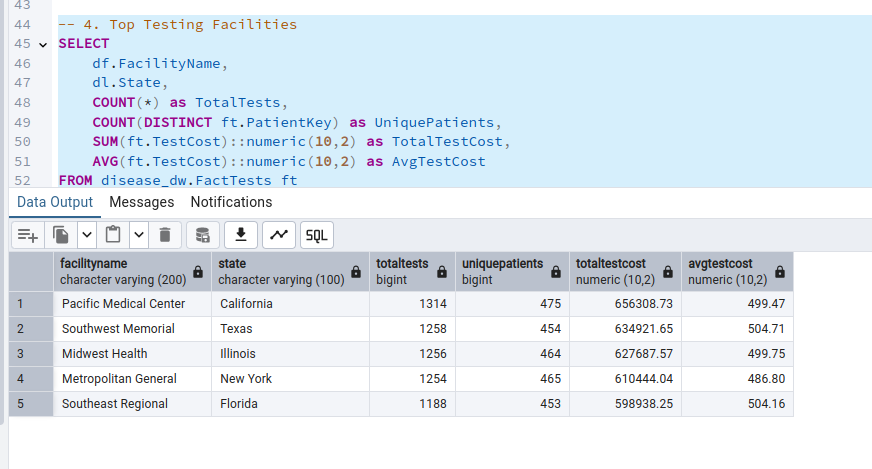
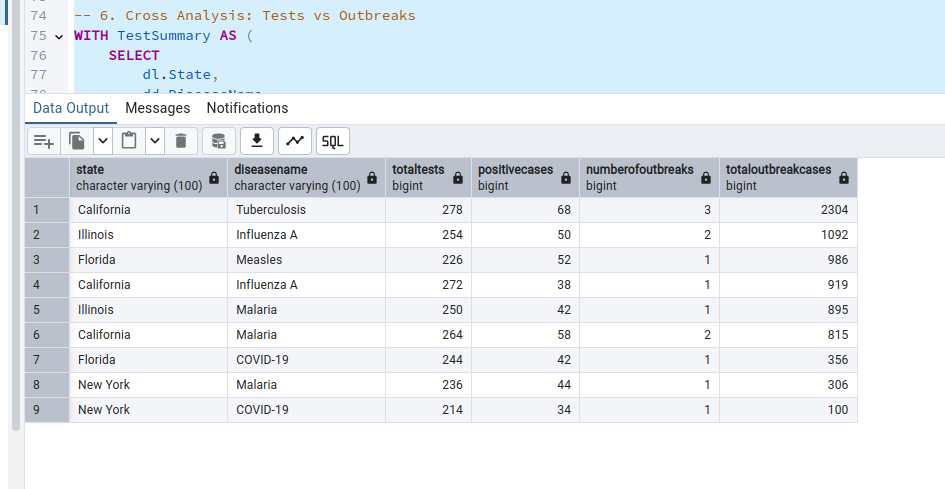
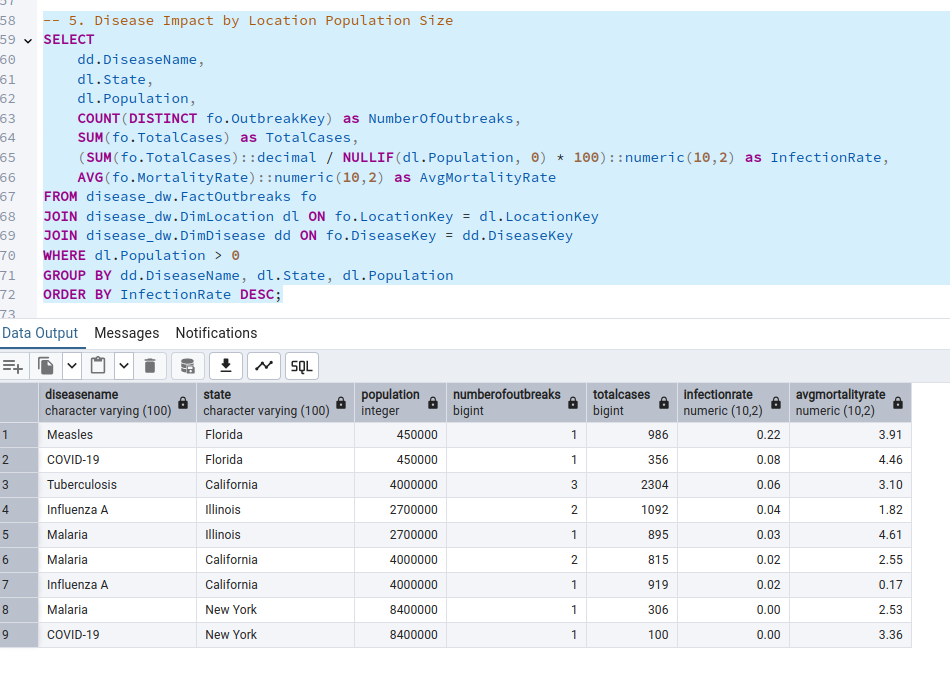


..

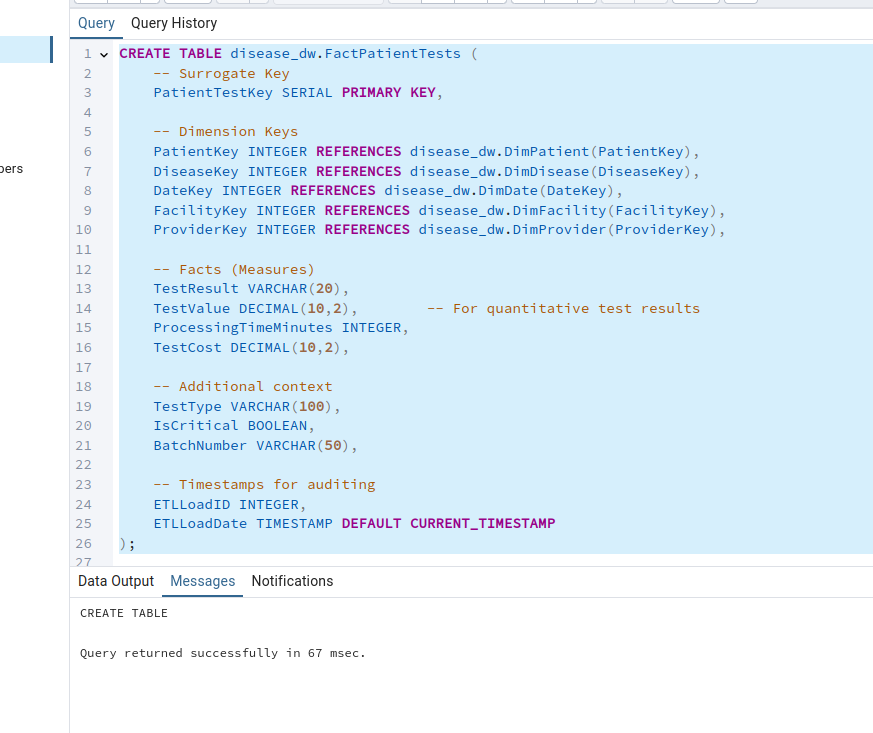
  
  
..



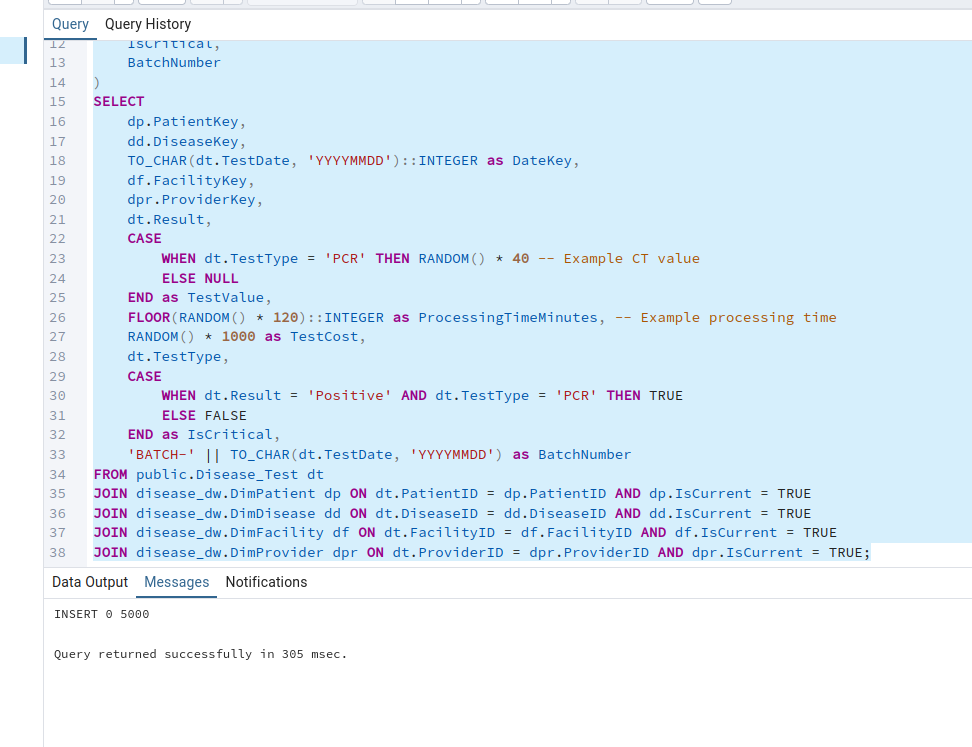
..

  
  
..  
  
  
  
  
  
>...  


|  
  
First, let's create our core fact table with this grain:

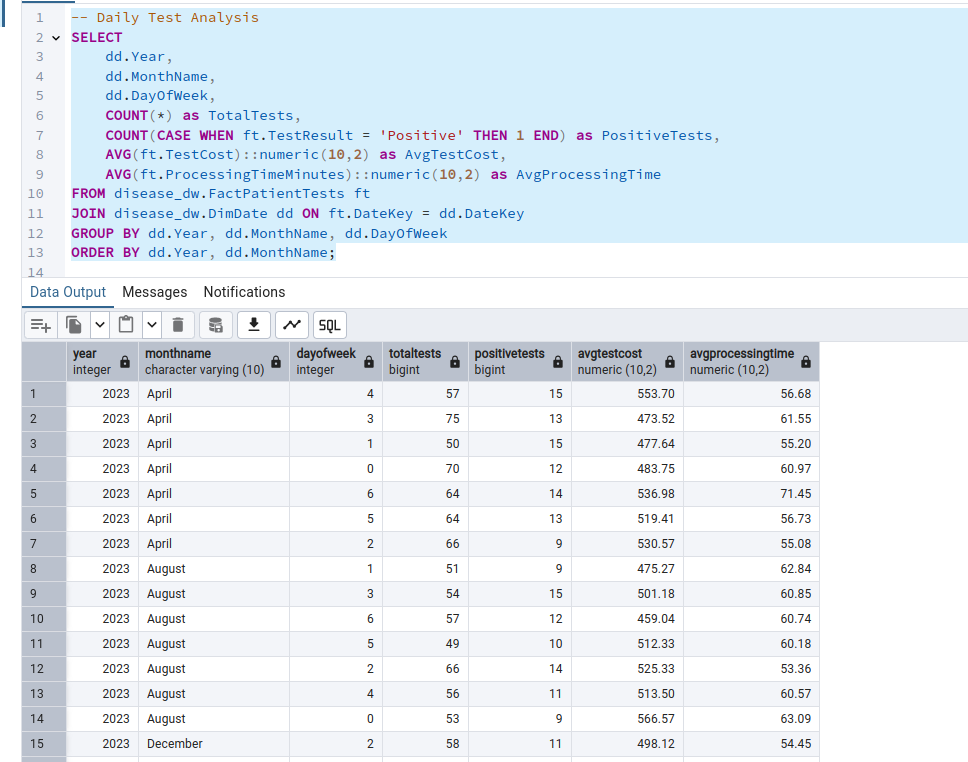
…

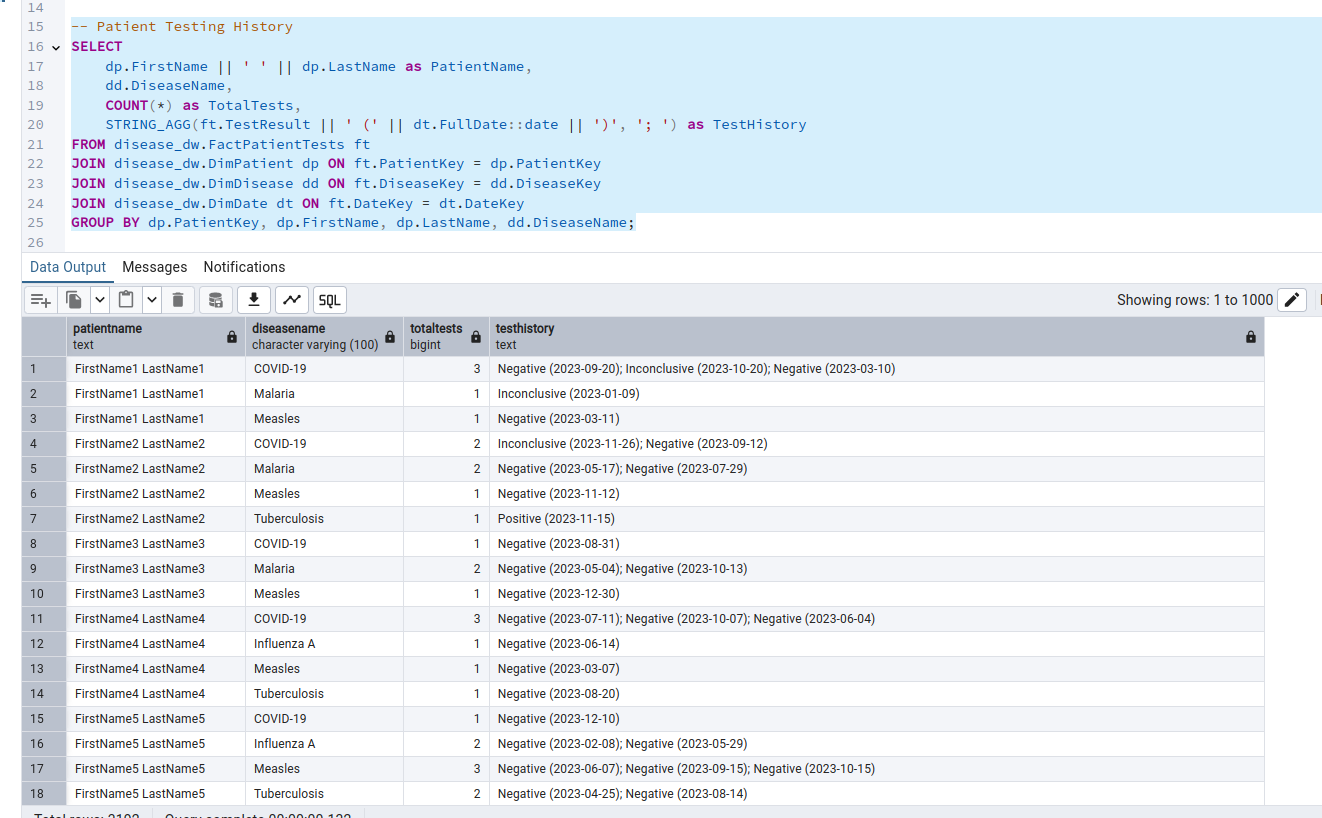
ETL Process to populate this fact table:

…

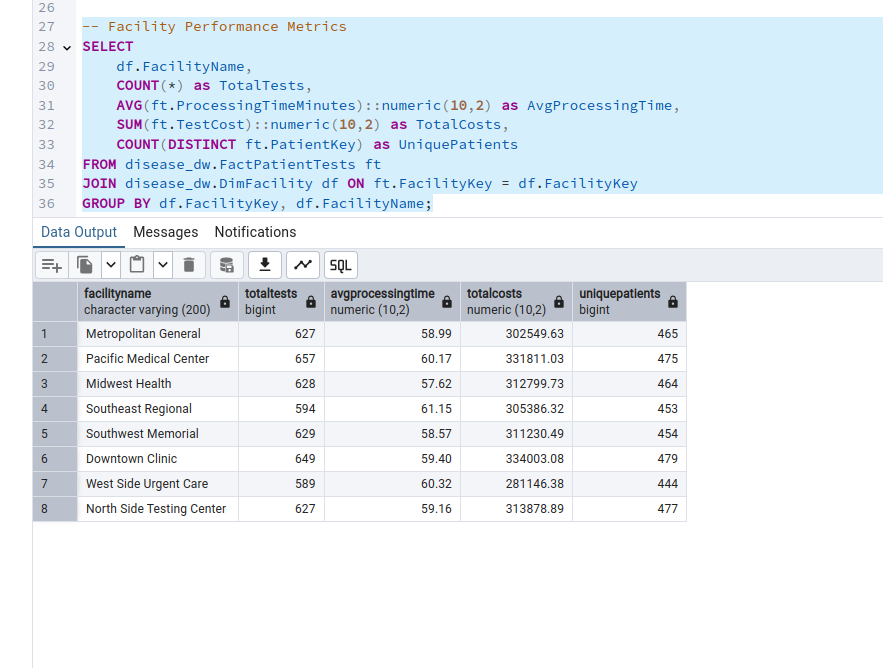
**Example analytical queries using this grain:**

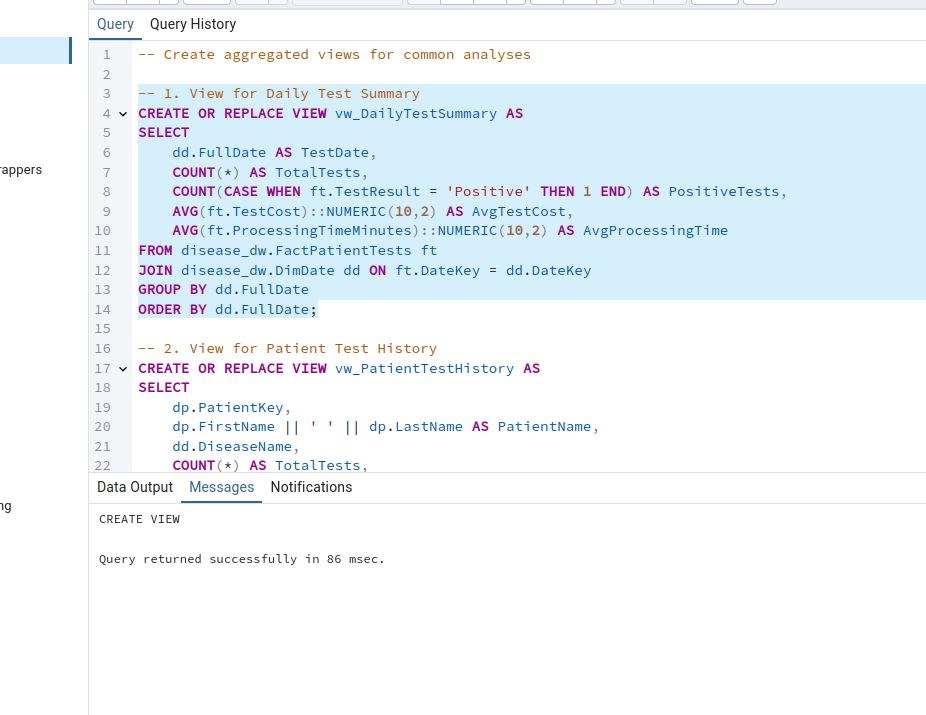
**Daily tes analyis**

 **Patient test history**

****

facility perfomance metrics

views for common analyses, including daily test summaries, patient test history, facility performance metrics, disease trend analysis, and provider test activity.

..

insightful queries using the views you created:

1.

some insightful queries using the views you created:

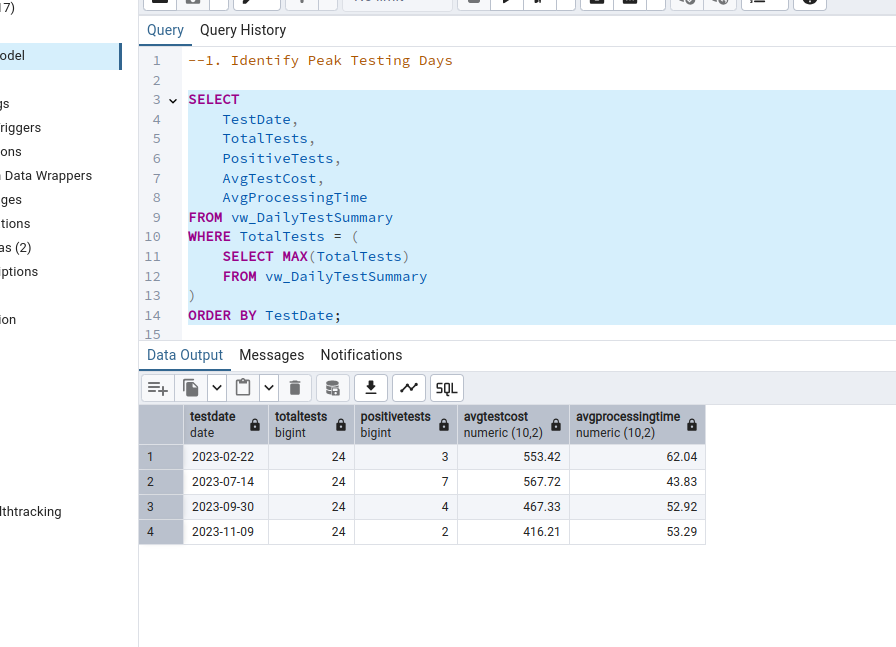
1. Identify Peak Testing Days

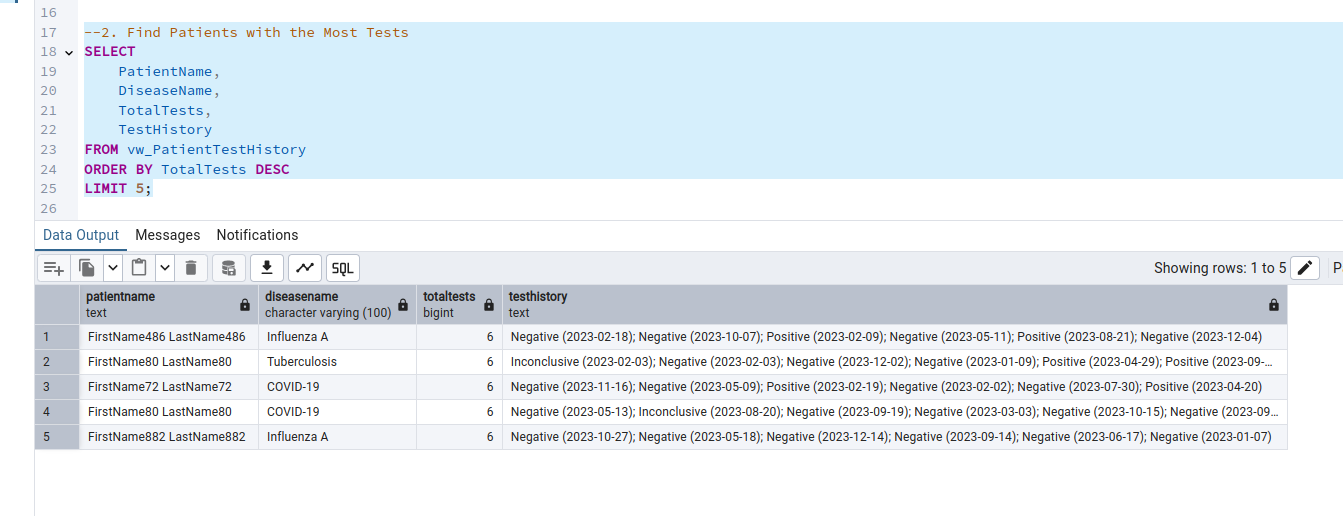
2. 2. Find Patients with the Most Tests

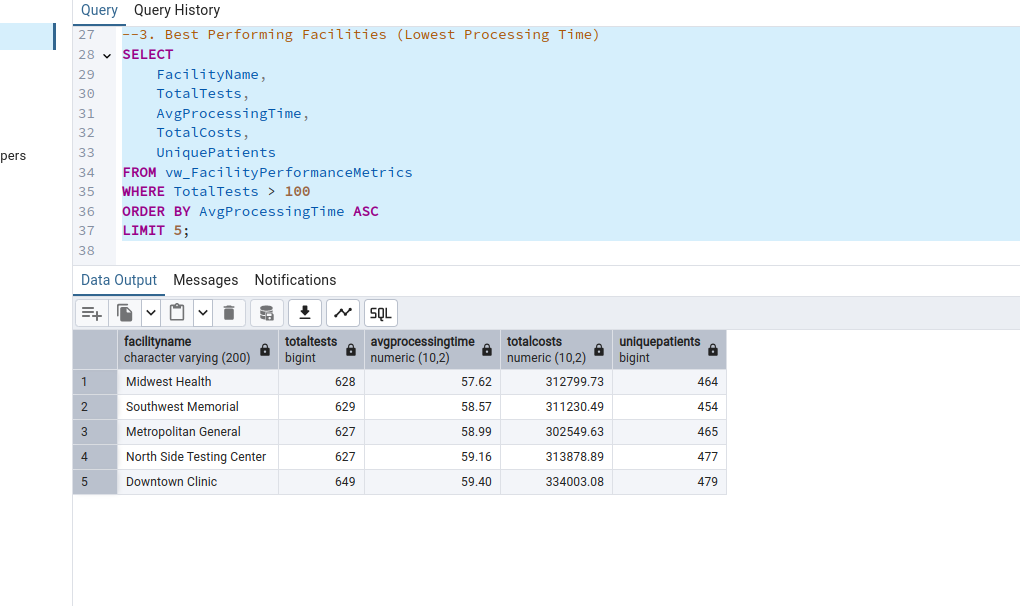
3. 3. Best Performing Facilities (Lowest Processing Time)

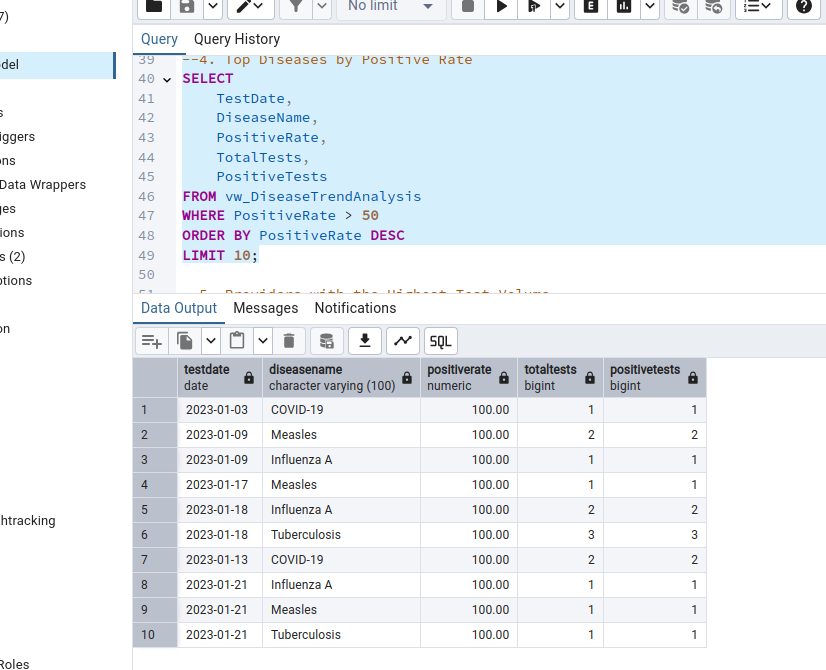
4. 4. Top Diseases by Positive Rate

5. Providers with the Highest Test Volume

..

…

…



Transitioning the relational structure of the Patient-Test-Date data warehouse to a NoSQL

MongoDB (Document-Based NoSQL Database)

How this disease tracking system would be structured differently in NoSQL versus the current relational model.

In a Relational Database (Current Structure):

- Data is normalized across multiple tables with strict relationships (foreign keys)

- Each entity (patient, disease, facility, etc.) has its own table

- Relationships are maintained through IDs (e.g., patientid, facilityid)

- Data integrity is enforced through constraints

- Complex queries are handled through JOINs

In a NoSQL Database (Document-oriented like MongoDB):

1. Patient Document:

```json

{

"\_id": ObjectId(),

"firstName": "string",

"lastName": "string",

"dateOfBirth": Date,

"gender": "string",

"bloodType": "string",

"contactInfo": {

"phone": "string",

"email": "string"

},

"medicalHistory": "text",

"region": {

"id": ObjectId(),

"name": "string",

"country": "string",

"state": "string",

"city": "string"

},

"cases": [

{

"diseaseId": ObjectId(),

"diseaseName": "string",

"diagnosisDate": Date,

"severity": "string",

"symptoms": ["string"],

"treatment": "text",

"outcome": "string",

"dischargeDate": Date,

"facility": {

"id": ObjectId(),

"name": "string",

"type": "string"

},

"provider": {

"id": ObjectId(),

"name": "string",

"specialty": "string"

}

}

],

"tests": [

{

"testId": ObjectId(),

"diseaseId": ObjectId(),

"testDate": Date,

"testType": "string",

"result": "string",

"facility": {

"id": ObjectId(),

"name": "string"

}

}

]

}

```

Key Differences:

1. Data Embedding vs. Normalization:

- NoSQL: Embeds related data within documents (e.g., patient contains cases and tests)

- Relational: Separates data into normalized tables with foreign key relationships

2. Schema Flexibility:

- NoSQL: Flexible schema allows for varying document structures

- Relational: Rigid schema with predefined columns and data types

3. Query Performance:

- NoSQL: Better for read-heavy operations as all related data is in one document

- Relational: Requires JOINs which can impact performance for complex queries

4. Data Redundancy:

- NoSQL: Accepts data redundancy for better query performance (e.g., facility name stored in multiple documents)

- Relational: Minimizes redundancy through normalization

5. Transaction Support:

- NoSQL: Limited transaction support across multiple documents

- Relational: Strong ACID compliance and transaction support

6. Scalability:

- NoSQL: Better horizontal scalability for large datasets

- Relational: Traditionally better vertical scalability

The choice between these models would depend on:

1. Read vs. Write patterns

2. Data consistency requirements

3. Scale of the application

4. Query complexity needs

5. Data relationships complexity

For this disease tracking system, the relational model might be better due to:

- Complex relationships between entities

- Need for data consistency

- Regular reporting requirements

- Complex querying needs across multiple entities

- Importance of data integrity in medical records

However, if the system needed to scale massively or handle varied disease tracking requirements across different regions, a NoSQL approach might be worth considering.

**comprehensive AWS architecture for this disease tracking system that follows AWS best practices and the Lambda Architecture pattern for both batch and real-time data processing.**

**...\**

**the key components and how they work together:**

1. Data Processing Architecture (Lambda Architecture):

A. Batch Layer:

* S3 buckets store raw data files
* AWS Glue for ETL jobs
* Amazon EMR for large-scale data processing
* Batch processing results stored in Aurora

B. Speed Layer:

* Amazon Kinesis Data Streams for real-time data ingestion
* Lambda functions for stream processing
* DynamoDB for real-time data storage
* Amazon ElastiCache for caching frequently accessed data

1. Application Layer:

* ECS/EKS clusters running containerized applications
* Auto-scaling groups across multiple AZs
* Application Load Balancer for traffic distribution
* Lambda functions for serverless processing

1. Database Layer:

* Amazon Aurora for relational data (Multi-AZ)
* DynamoDB for NoSQL requirements
* Amazon ElastiCache for Redis (caching layer)

1. Security Measures:

A. Network Security:

* VPC with public and private subnets
* Network ACLs and Security Groups
* AWS WAF for web application firewall
* AWS Shield for DDoS protection

B. Data Security:

* AWS KMS for encryption key management
* Data encrypted at rest and in transit
* AWS Secrets Manager for credentials
* IAM roles and policies

C. Compliance:

* AWS CloudTrail for audit logging
* AWS Config for compliance monitoring
* Amazon GuardDuty for threat detection

1. Resilience Features:

A. High Availability:

* Multi-AZ deployments
* Auto-scaling groups
* Cross-region replication for critical data
* Route 53 health checks and DNS failover

B. Disaster Recovery:

* Regular snapshots to S3
* Cross-region backup strategy
* Aurora global databases
* DynamoDB global tables

1. Performance Optimization:

A. Database:

* Aurora read replicas
* DynamoDB DAX
* ElastiCache for caching
* Auto-scaling policies

B. Application:

* CloudFront CDN for static content
* Auto-scaling based on metrics
* Lambda concurrent execution limits
* Kinesis throughput optimization

1. Monitoring and Maintenance:

A. Monitoring:

* CloudWatch metrics and alarms ==
* X-Ray for distributed tracing
* CloudWatch Logs for centralized logging
* AWS Health Dashboard

B. Operational Tools:

* Systems Manager for management
* CloudFormation for infrastructure as code
* AWS Backup for automated backups
* AWS Organizations for multi-account management

...

.....