PostgreSQL

# Advanced Postgres

Surendra Panpaliya

# Agenda

- Views and Materialized Views
- Creating and using views
- Benefits of materialized views
- Refreshing materialized views

# Agenda

01

Performance Tuning

02

Understanding query plans

03

Indexing strategies

04

Analyzing and optimizing queries

# Views and Materialized Views

**Surendra Panpaliya**

# Views

Virtual table

Representing

Result of

Stored query

# Views

Useful for

Simplifying complex queries,

Enforcing security

Creating a level of abstraction

Over raw data tables

# Creating a View

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

# Example

```
CREATE TABLE patients (
    patient_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    date_of_birth DATE,
    gender CHAR(1)
);
```

# Example

To create a view that shows only male patients

```
CREATE VIEW male_patients AS
SELECT patient_id, first_name, last_name, date_of_birth
FROM patients
WHERE gender = 'M';
```

# Using the View

```
SELECT * FROM male_patients;
```

# Updating Data Through a View

Views can also be updatable under certain conditions

```
UPDATE male_patients
SET first_name = 'John'
WHERE patient_id = 1;
```

# Advantages of Views

Simplifies complex queries.

Enhances security by restricting access

to specific rows or columns.

Provides a consistent, reusable interface.

# Materialized Views

Physical copy

of the result

of a query

# Materialized Views

Unlike regular views,

Materialized views

store data

Must be refreshed

when underlying

data changes.

# Creating a Materialized View

```
CREATE MATERIALIZED VIEW mat_view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition
WITH [ NO ] DATA;
```

# Example

Create a materialized view

that stores the result

of the query

for male patients

# Example

```
CREATE MATERIALIZED VIEW mat_male_patients AS
SELECT patient_id, first_name, last_name, date_of_birth
FROM patients
WHERE gender = 'M';
```

# Refreshing a Materialized View

Materialized views

must be refreshed

to get updated data

# Refreshing a Materialized View

```
REFRESH MATERIALIZED VIEW mat_male_patients;
```

# Using the Materialized View

```
SELECT * FROM mat_male_patients;
```

# Advantages of Materialized Views

Improves performance

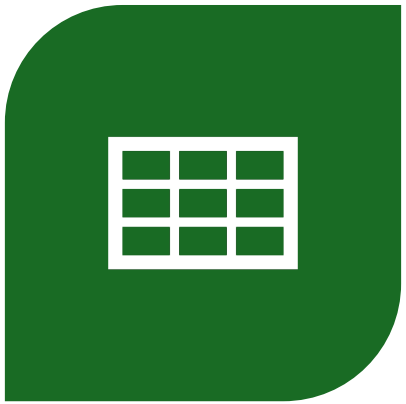
by storing query results.

Useful for read-heavy workloads

where data does not change frequently.

# Differences Between Views and Materialized Views

**STORAGE:**



VIEWS DO NOT STORE DATA;

MATERIALIZED VIEWS STORE THE QUERY RESULT.

# Differences Between Views and Materialized Views

**Performance:**

Views execute the underlying query

each time they are accessed;

materialized views provide

faster access

to precomputed results.

# Differences Between Views and Materialized Views

**REFRESH:**

VIEWS ALWAYS SHOW

UP-TO-DATE DATA;

MATERIALIZED VIEWS

NEED TO BE EXPLICITLY REFRESHED.

# Creating and using views

Surendra Panpaliya

# Creating and using views

Views can be used

to simplify complex queries

improve security

by restricting data access

# Creating and using views

Provide

a consistent

Reusable interface

for frequently

accessed queries

# Creating Tables in the Healthcare

```
CREATE TABLE patients (
    patient_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    date_of_birth DATE,
    gender CHAR(1)
);
```

# Creating Tables in the Healthcare

```
CREATE TABLE appointments (
    appointment_id SERIAL PRIMARY KEY,
    patient_id INT REFERENCES patients(patient_id),
    doctor_name VARCHAR(50),
    appointment_date TIMESTAMP,
    diagnosis VARCHAR(255)
);
```

# Creating Tables in the Healthcare

```
CREATE TABLE prescriptions (
    prescription_id SERIAL PRIMARY KEY,
    appointment_id INT REFERENCES
appointments(appointment_id),
    medication_name VARCHAR(50),
    dosage VARCHAR(50),
    instructions TEXT
);
```

# 2. Creating Views for Simplified Data Access

**Example 1**

**View for Patients' Basic Information**

A view that provides basic information about patients

# View for Patients' Basic Information

```
CREATE VIEW patient_info AS
SELECT patient_id, first_name, last_name, date_of_birth, gender
FROM patients;
```

# 2. Creating Views for Simplified Data Access

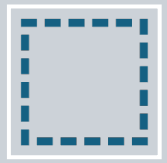
**Usage**

SELECT * FROM patient_info;

# 2. View for Upcoming Appointments

```
CREATE VIEW upcoming_appointments AS

SELECT a.appointment_id, p.first_name, p.last_name,

a.doctor_name, a.appointment_date, a.diagnosis

FROM appointments a

JOIN patients p ON a.patient_id = p.patient_id

WHERE a.appointment_date > CURRENT_TIMESTAMP;
```

# 2 View for Upcoming Appointments



**Usage**

SELECT * FROM upcoming_appointments;

# 3. View for Prescriptions by Patient

A view that lists

all prescriptions

for a specific patient

# Ex3. View for Prescriptions by Patient

```
CREATE VIEW patient_prescriptions AS

SELECT p.first_name, p.last_name, a.appointment_date,

pr.medication_name, pr.dosage, pr.instructions

FROM prescriptions pr

JOIN appointments a ON pr.appointment_id = a.appointment_id

JOIN patients p ON a.patient_id = p.patient_id;
```

# 3. View for Prescriptions by Patient

**Usage:**

```
SELECT * FROM patient_prescriptions WHERE
first_name = 'John' AND last_name = 'Doe';
```

# 3. Updating Data Through Views

Views can also be used

to update data under certain conditions.

Let's assume we have a view

that includes patients' basic information

# 3. Updating Data Through Views

```
CREATE VIEW editable_patient_info AS
SELECT patient_id, first_name, last_name, date_of_birth, gender
FROM patients
WITH CHECK OPTION;
```

# 3. Updating Data Through Views

| **Updating Data Using the View:** |
| --- |
| UPDATE editable_patient_info |
| SET first_name = 'Jonathan' |
| WHERE patient_id = 1; |

# 4. Securing Data with Views

Views can restrict access

to sensitive data.

For example

Hide patients’ full names in

a public-facing application

# 4. Securing Data with Views

```
CREATE VIEW anonymized_patient_info
AS SELECT patient_id,
CONCAT(SUBSTRING(first_name, 1, 1), '. ', last_name)
AS name, date_of_birth, gender
FROM patients;
```

# 4. Securing Data with Views

**Usage:**

SELECT * FROM anonymized_patient_info;

# Practical Examples

Surendra Panpaliya

# 1. View for All Female Patients

```
CREATE VIEW female_patients AS
SELECT patient_id, first_name, last_name, date_of_birth
FROM patients
WHERE gender = 'F';
```

# 2. View for Recent Diagnoses

```
CREATE VIEW recent_diagnoses AS

SELECT a.appointment_id, p.first_name, p.last_name,

a.doctor_name, a.diagnosis, a.appointment_date

FROM appointments a

JOIN patients p ON a.patient_id = p.patient_id

WHERE a.appointment_date > NOW() - INTERVAL '30 days';
```

# 3. View for Medication Instructions

```
CREATE VIEW medication_instructions AS

SELECT p.first_name, p.last_name, pr.medication_name,

pr.dosage, pr.instructions

FROM prescriptions pr

JOIN appointments a ON pr.appointment_id = a.appointment_id

JOIN patients p ON a.patient_id = p.patient_id;
```

# 4. View for Daily Appointments

```
CREATE VIEW daily_appointments AS

SELECT a.appointment_id, p.first_name, p.last_name,

a.doctor_name, a.appointment_date

FROM appointments a

JOIN patients p ON a.patient_id = p.patient_id

WHERE DATE(a.appointment_date) = CURRENT_DATE;
```

# Summary

VIEWS ARE A POWERFUL TOOL

FOR MANAGING

QUERYING DATA EFFICIENTLY.

# Summary

HELP SIMPLIFY COMPLEX QUERIES,

ENHANCE SECURITY

PROVIDE A LEVEL OF ABSTRACTION

THAT CAN MAKE THE DATABASE

MORE USER-FRIENDLY

# Summary

ENSURE THAT

SENSITIVE INFORMATION IS PROTECTED,

WHILE STILL MAKING NECESSARY DATA

EASILY ACCESSIBLE

TO AUTHORIZED USERS.

# Benefits of materialized views

**Surendra Panpaliya**

# Benefits of materialized views

Materialized views offer

several benefits in the healthcare domain,

especially in scenarios that

require frequent and

complex queries over large datasets.

# 1. Performance Improvement

**Faster Query Response**

Materialized views

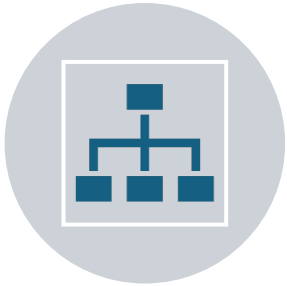
Store the result of a query physically.

Reduces the time needed

To fetch results

# 1. Performance Improvement

Example



Frequently accessed Reports,

such as patient visit summaries,

can be generated quickly.

# 1. Performance Improvement

```
CREATE MATERIALIZED VIEW patient_visit_summary AS
SELECT patient_id, COUNT(*) AS visit_count
FROM appointments
GROUP BY patient_id;
```

# Optimized Data Aggregation

Aggregated data,

such as daily patient visits or

medication usage statistics,

can be precomputed and stored,

improving query performance.

# Optimized Data Aggregation

```
CREATE MATERIALIZED VIEW daily_patient_visits AS
SELECT DATE(appointment_date) AS visit_date,
COUNT(*) AS visit_count
FROM appointments
GROUP BY DATE(appointment_date);
```

# 2. Reduced Load on Source Tables

**Offloading Queries**

Complex and resource-intensive queries

are offloaded from source tables,

reducing the load and

contention on those tables.

# 2. Reduced Load on Source Tables

```
CREATE MATERIALIZED VIEW medication_usage AS

SELECT medication_name, COUNT(*) AS usage_count

FROM prescriptions

GROUP BY medication_name;
```

# Improved Performance for Concurrent Users

IN ENVIRONMENTS WITH MANY CONCURRENT USERS,

MATERIALIZED VIEWS HELP IN

REDUCING THE PERFORMANCE IMPACT

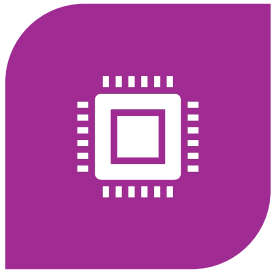
ON THE UNDERLYING TABLES.

# 3. Simplified Complex Queries

**PRECOMPUTED RESULTS**

QUERIES THAT INVOLVE COMPLEX JOINS AND

CALCULATIONS CAN BE PRECOMPUTED AND STORED,



MAKING IT EASIER FOR END-USERS TO RETRIEVE DATA.

# 3. Simplified Complex Queries

```
CREATE MATERIALIZED VIEW patient_medication_details AS

SELECT p.patient_id, p.first_name, p.last_name,

pr.medication_name, pr.dosage, pr.instructions

FROM patients p

JOIN appointments a ON p.patient_id = a.patient_id

JOIN prescriptions pr ON a.appointment_id = pr.appointment_id;
```

# Consistent Results

MATERIALIZED VIEWS PROVIDE

CONSISTENT AND REPEATABLE RESULTS

FOR COMPLEX QUERIES,

ENSURING DATA INTEGRITY AND ACCURACY.

# 4. Historical Data and Snapshots

**Data Archiving**

Materialized views can be used

to create snapshots of data

at specific points in time.

Useful for

Historical analysis and reporting

# 4. Historical Data and Snapshots

```
CREATE MATERIALIZED VIEW monthly_patient_visits AS

SELECT  DATE_TRUNC('month',  appointment_date)  AS  month,

COUNT(*) AS visit_count

FROM appointments

GROUP BY DATE_TRUNC('month', appointment_date);
```

# 4. Historical Data and Snapshots

**Trend Analysis**

Historical snapshots help

in analyzing trends over time,

such as patient admission rates or

medication usage patterns.

# 5. Improved Data Security and Access Control

**Restricting Access**

Sensitive data can be filtered and

stored in a materialized view,

restricting direct access

to the underlying tables

# 5. Improved Data Security and Access Control

```
CREATE MATERIALIZED VIEW limited_patient_info AS
SELECT patient_id, first_name, last_name, date_of_birth
FROM patients
WHERE gender = 'F';
```

# 5. Improved Data Security and Access Control

**Role-Based Access Control**

Materialized views can be used

to provide access to aggregated or

anonymized data,

ensuring compliance with

data protection regulations.

# 6. Reduced Network Traffic

**Local Storage of Data**

Materialized views store data locally,

reducing the need to fetch data

from remote databases repeatedly.

# 6. Reduced Network Traffic

**Efficient Data Sharing**

Data can be shared efficiently

between different departments or

applications without impacting

the performance of the source tables.

# 7. Data Consistency and Availability

**Ensuring Data Consistency**

Materialized views provide

a consistent state of data,

which is especially useful

in environments where

data consistency is critical.

# 7. Data Consistency and Availability

**HIGH AVAILABILITY**

MATERIALIZED VIEWS CAN BE REFRESHED

AT REGULAR INTERVALS,

ENSURING THAT DATA IS UP-TO-DATE AND

AVAILABLE FOR REPORTING AND ANALYSIS.

# Examples in Healthcare Domain

Surendra Panpaliya

# Example 1: Daily Patient Summary

A materialized view

to summarize daily patient visits,

reducing the need

to query the appointments table frequently.

# Example 1: Daily Patient Summary

```
CREATE MATERIALIZED VIEW daily_patient_summary AS

SELECT patient_id, DATE(appointment_date) AS visit_date,

COUNT(*) AS visit_count

FROM appointments

GROUP BY patient_id, DATE(appointment_date);
```

# Example 2: Monthly Medication Usage

A materialized view

to track monthly usage of medications,

helping in inventory management and

procurement planning.

# Example 2: Monthly Medication Usage

```
CREATE MATERIALIZED VIEW monthly_medication_usage AS

SELECT medication_name, DATE_TRUNC('month',

prescription_date) AS month, COUNT(*) AS usage_count

FROM prescriptions

GROUP BY medication_name, DATE_TRUNC('month',

prescription_date);
```

# Example 3: High-Risk Patients

A materialized view

to identify high-risk patients

based on certain criteria,

such as multiple visits

in a short period.

# Example 3: High-Risk Patients

```
CREATE MATERIALIZED VIEW high_risk_patients AS

SELECT patient_id, COUNT(*) AS visit_count

FROM appointments

WHERE appointment_date > NOW() - INTERVAL '30 days'

GROUP BY patient_id

HAVING COUNT(*) > 5;
```

# Conclusion

Materialized views provide

numerous benefits in the healthcare domain

by improving query performance,

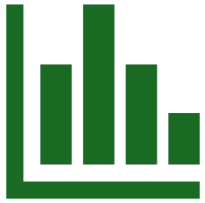
reducing load on source tables,

simplifying complex queries,

enhancing data security.

# Conclusion

Are a valuable tool



for efficient data management

analysis in healthcare databases.

# Refreshing materialized views

Surendra Panpaliya

# Refreshing materialized views

Crucial in a healthcare domain

to ensure that the data is

up-to-date for accurate

reporting and analysis.

# Refreshing materialized views

Materialized views can be refreshed

Manually or Automatically

at specified intervals,

depending on the requirements.

# Types of Refresh Methods

**Complete Refresh**

This rebuilds

the entire materialized view

from scratch.

# Types of Refresh Methods

**Incremental Refresh**

Known as "fast refresh,"

Updates only the changes since

the last refresh,

which is more efficient

than a complete refresh.

# Types of Refresh Methods

| |
|---|
| **Manual Refresh** |
| You can manually refresh a materialized view |
| using the REFRESH MATERIALIZED VIEW command. |
| **Example 1: Complete Refresh** |
| REFRESH MATERIALIZED VIEW daily_patient_summary; |

# Example 2: Incremental Refresh

To enable incremental refresh,

Create the materialized view

with the WITH DATA and

WITH NO DATA clauses,

ensure that the underlying tables

have LOGGED updates.

# Example 2: Incremental Refresh

```
CREATE MATERIALIZED VIEW daily_patient_summary AS

SELECT patient_id, DATE(appointment_date) AS visit_date,

COUNT(*)  AS visit_count

FROM appointments

GROUP BY patient_id, DATE(appointment_date)

WITH NO DATA;
```

# Example 2: Incremental Refresh

Later, you can do an incremental refresh:

```
REFRESH MATERIALIZED VIEW daily_patient_summary WITH
DATA;
```

# Automatic Refresh

Set up Automatic refresh

using pg_cron extension or

Similar job scheduler

to refresh the materialized view

at regular intervals.

# Automatic Refresh

**Example: Using pg_cron**

First, install the pg_cron extension

(if not already installed)

CREATE EXTENSION pg_cron;

# Automatic Refresh

Schedule a job to refresh the materialized view

```
SELECT cron.schedule('0 0 * * *', 'REFRESH
MATERIALIZED VIEW daily_patient_summary’);
```

Schedules the materialized view to refresh daily at midnight.

# Examples in Healthcare Domain

Surendra Panpaliya

# Example 1: Daily Patient Summary

```
-- A materialized view to summarize daily patient visits, refreshed daily
CREATE MATERIALIZED VIEW daily_patient_summary
AS SELECT patient_id, DATE(appointment_date)
AS visit_date, COUNT(*) AS visit_count
FROM appointments
GROUP BY patient_id, DATE(appointment_date);
```

# Example 1: Daily Patient Summary

```
-- Schedule a daily refresh

SELECT cron.schedule('0 0 * * *', 'REFRESH

MATERIALIZED VIEW daily_patient_summary');
```

# Example 2: Monthly Medication Usage

A materialized view

to track monthly usage of medications,

refreshed monthly

# Example 2: Monthly Medication Usage

```
CREATE MATERIALIZED VIEW monthly_medication_usage
AS SELECT medication_name, DATE_TRUNC('month',
prescription_date) AS month, COUNT(*) AS usage_count
FROM prescriptions
GROUP BY medication_name, DATE_TRUNC('month',
prescription_date);
```

# Example 2: Monthly Medication Usage

```
-- Schedule a monthly refresh
```

```
SELECT cron.schedule('0 0 1 * *', 'REFRESH MATERIALIZED
VIEW monthly_medication_usage');
```

# Example 3: High-Risk Patients

A materialized view

to identify high-risk patients

based on certain criteria,

refreshed hourly

# Example 3: High-Risk Patients

```
CREATE MATERIALIZED VIEW high_risk_patients AS
SELECT patient_id, COUNT(*) AS visit_count
FROM appointments
WHERE appointment_date > NOW() - INTERVAL '30 days'
GROUP BY patient_id
HAVING COUNT(*) > 5;
```

# Example 3: High-Risk Patients

-- Schedule an hourly refresh

SELECT cron.schedule('0 * * * *’,

'REFRESH MATERIALIZED VIEW high_risk_patients');

# Summary

Refreshing materialized views

in a healthcare domain is

essential for maintaining

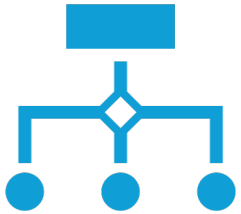
up-to-date and

accurate data.

# Summary

Depending on the requirements,

materialized views can be refreshed



manually or automatically.

# Summary

Automating the refresh process

ensures timely updates

without manual intervention

# Summary

Improving the efficiency and

reliability of

data management in

healthcare databases.

# Agenda

1. **Performance Tuning**
2. Understanding query plans
3. Indexing strategies
4. Analyzing and optimizing queries

# Performance Tuning

Surendra Panpaliya

# Performance Tuning

| | | |
|---|---|---|
| Involves | Various strategies | Techniques to optimize |
| The database system | for faster and | more efficient operation |

# Key Areas and Techniques

Surendra Panpaliya

# 1. Indexing

Indexes are critical for improving

the performance of queries.

However, they should be used

judiciously because

they can also impact

write performance.

# Types of Indexes

**B-TREE INDEXES**

DEFAULT INDEX TYPE,

GOOD FOR MOST QUERIES.

# Types of Indexes

**Hash Indexes**

Useful for

equality

comparisons.

# Types of Indexes

**GIN/GIST INDEXES**

USEFUL FOR FULL-TEXT SEARCH,

JSONB DATA TYPES, AND

OTHER SPECIALIZED QUERIES.

# Types of Indexes

**BRIN Indexes**

Useful for very large tables

where columns have

a natural ordering.

# Creating an Index

```
CREATE INDEX idx_patient_last_name ON patients (last_name);
```

# 2. Query Optimization

Optimize SQL queries

for better performance.

Use EXPLAIN command

to understand

the query execution plan.

# 2. Query Optimization Example

```
EXPLAIN ANALYZE SELECT * FROM patients WHERE
last_name = 'Smith';
```

# 3. Configuration Tuning

Adjust

PostgreSQL configuration

settings in postgresql.conf

for better performance.

# Important Settings

**shared_buffers**

Amount of memory

the database server

uses for

shared memory buffers.

# Important Settings

**work_mem**

Amount of memory used

for internal sort operations and

hash tables.

# Important Settings

**Maintenance_work_mem:**

Memory allocated

for maintenance

tasks such as VACUUM.

# Important Settings

**effective_cache_size:**

An estimate of how much memory

is available for disk caching

by the operating system and

within the database itself.

# Important Settings

**max_connections:**

The maximum number

of concurrent connections

to the database.

# Example Configuration

shared_buffers = 4GB

work_mem = 64MB

maintenance_work_mem = 512MB

effective_cache_size = 12GB

max_connections = 100

# 4. Vacuuming and Analyzing

Regularly vacuum and

analyze the database

to maintain performance and

recover disk space.

# VACUUM Command

VACUUM FULL;

# ANALYZE Command

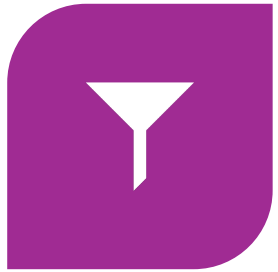
```
ANALYZE;
```

# 5. Partitioning

PARTITION LARGE TABLES

TO IMPROVE QUERY PERFORMANCE AND MANAGEABILITY.

POSTGRESQL SUPPORTS RANGE,



LIST, AND HASH PARTITIONING.

# Creating a Partitioned Table

```
CREATE TABLE patient_visits (
    visit_id SERIAL PRIMARY KEY,
    patient_id INT,
    visit_date DATE,
    details TEXT
) PARTITION BY RANGE (visit_date);
```

# 5. Partitioning

CREATE TABLE

patient_visits_2023

PARTITION OF patient_visits

FOR VALUES

FROM ('2023-01-01') TO ('2023-12-31');

# Connection Pooling

Use connection pooling

to manage

database connections

efficiently.

# Connection Pooling

Tools like PgBouncer

can help

reduce the overhead

of establishing connections.

# Connection Pooling

**Example PgBouncer Configuration:**

```
[databases]
mydb = host=127.0.0.1 port=5432 dbname=mydb

[pgbouncer]
listen_addr = 127.0.0.1
listen_port = 6432
```

# Connection Pooling

auth_type = md5

auth_file = /etc/pgbouncer/userlist.txt

pool_mode = session

max_client_conn = 100

default_pool_size = 20

# 7. Caching

USE CACHING

TO STORE FREQUENTLY ACCESSED DATA

IN MEMORY,

REDUCING THE NEED

FOR REPEATED DATABASE QUERIES.

# Using Redis for Caching

```
import redis

r = redis.Redis(host='localhost', port=6379, db=0)

# Set a value in the cache
r.set('patient_1234', 'John Doe')

# Get a value from the cache
patient_name = r.get('patient_1234')
```

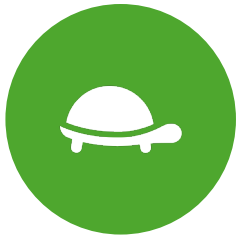
# 8. Monitoring and Logging

MONITOR

DATABASE PERFORMANCE

SET UP LOGGING

TO IDENTIFY BOTTLENECKS AND



SLOW QUERIES.

# 8. Monitoring and Logging

**Enabling Logging**

log_statement = 'all'

log_min_duration_statement = 500

# Using Monitoring Tools

**pg_stat_statements:**

Provides execution statistics

for all SQL statements executed.

CREATE EXTENSION pg_stat_statements;

# Using Monitoring Tools

**pgAdmin:**

A web-based PostgreSQL

management tool.

**Prometheus and Grafana:**

For advanced monitoring and visualization.

# 9. Hardware and System Tuning

ENSURE THAT

THE UNDERLYING HARDWARE AND

OPERATING SYSTEM ARE

OPTIMIZED FOR

DATABASE PERFORMANCE.

# 9. Hardware and System Tuning

**Key Considerations:**

**Disk I/O:**

Use fast disks (e.g., SSDs)

for better performance

# 9. Hardware and System Tuning

**Key Considerations:**

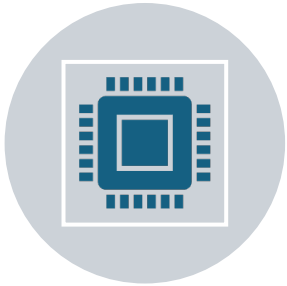
**Memory:**

Ensure enough RAM is available

for the database and the operating system.

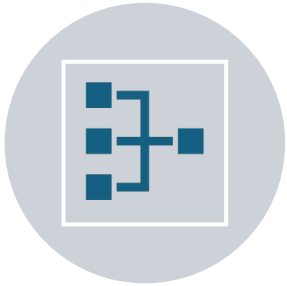
# 9. Hardware and System Tuning

**Key Considerations:**



**CPU:**

More CPU cores can help handle



more parallel processes.

# 9. Hardware and System Tuning

**Key Considerations:**

**Network:**

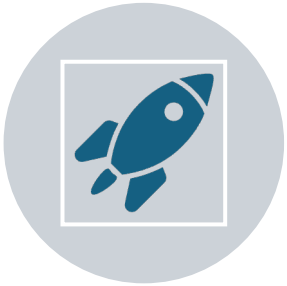
Optimize network settings

if the database is accessed over a network.

# 10. Use Appropriate Data Types

Choose the most appropriate

data types for your columns



to save space and

improve performance.

# 10. Use Appropriate Data Types

**Example:**

Use INTEGER instead of BIGINT

if the values will fit within the range of INTEGER.

Use TEXT or VARCHAR

for variable-length strings.

# Conclusion

Performance tuning in

PostgreSQL

is a multi-faceted

process

# Conclusion

- Involves
- optimizing queries,
- indexing strategies
- Configuration settings,
- Hardware resources

# Conclusion

Regular monitoring,

Maintenance, and adjustments based

on workload and performance metrics

are essential to ensure

optimal database performance.

# Understanding query plans

**Surendra Panpaliya**

# Understanding query plans

Essential for

optimizing database performance,

particularly in a healthcare domain

where timely and accurate data

retrieval is critical

# Understanding query plans

PostgreSQL provides tools

EXPLAIN and

EXPLAIN ANALYZE

to understand how queries

are executed.

# Why Query Plans are Important?

Query plans show

how PostgreSQL executes

a query.

# Why Query Plans are Important?

- The order of operations (scans, joins, sorts, etc.)
- Estimated and actual costs
- Estimated and actual row counts
- Use of indexes
- Memory usage

# Why Query Plans are Important?

Can identify

inefficiencies

Make informed decisions

to optimize queries.

# Using EXPLAIN

Provides

query execution plan

without running the query.

# EXPLAIN ANALYZE

Runs the query and

provides the actual execution plan,

including run-time statistics.

# Example Scenarios

Retrieving Patient Information

```
SELECT * FROM patients WHERE
last_name = 'Jackson';
```

# Execution Plan

```
EXPLAIN ANALYZE
```

```
SELECT * FROM patients WHERE
last_name = 'Jackson';
```

# Sample Output

"Seq Scan on patients  (cost=0.00..13.62 rows=1 width=252) (actual time=0.016..0.019 rows=1 loops=1)"

"  Filter: ((last_name)::text = 'Jackson'::text)"

"  Rows Removed by Filter: 2"

"Planning Time: 0.061 ms"

"Execution Time: 0.036 ms"

# Sample Output

A sequential scan is used

because there is no index

on the last_name column.

Adding an index on last_name

can improve performance.

# Sample Output

```
CREATE INDEX idx_patients_last_name ON patients
(last_name);
```

```
EXPLAIN ANALYZE SELECT * FROM patients WHERE last_name
= 'Jackson';
```

# Sample Output After Indexing

"Seq Scan on patients  (cost=0.00..1.04 rows=1 width=252) (actual time=0.010..0.011 rows=1 loops=1)"

"  Filter: ((last_name)::text = 'Jackson'::text)"

"  Rows Removed by Filter: 2"

"Planning Time: 0.316 ms"

"Execution Time: 0.026 ms"

# Indexing strategies

Surendra Panpaliya

# Indexing strategies

ESSENTIAL TO

ENSURE EFFICIENT DATA RETRIEVAL,

ESPECIALLY GIVEN

THE LARGE VOLUMES

OF DATA.

# Indexing strategies

Proper indexing can

significantly improve

query performance

reduce response times

optimize overall

database performance

# Analyzing and optimizing queries

Surendra Panpaliya

# Analyzing and optimizing queries

Crucial to ensure

efficient and timely access

to patient data,

which is vital

for healthcare providers

# Steps for Query Analysis and Optimization

Identify Slow Queries

Analyze Query Plans

Optimize Indexes

Refactor Queries

Monitor Performance

Regular Maintenance

# 1. Identify Slow Queries

Use PostgreSQL's

pg_stat_statements extension

to track the performance of all queries:

# 1. Identify Slow Queries

Use PostgreSQL’s

pg_stat_statements

extension to track

the performance of

all queries

# 1. Identify Slow Queries

CREATE EXTENSION pg_stat_statements;

Retrieve the most time-consuming queries:

SELECT query, total_time, calls, mean_time

FROM pg_stat_statements

ORDER BY total_time DESC

LIMIT 10;

# 2. Analyze Query Plans

Use the EXPLAIN

EXPLAIN ANALYZE commands

to understand

how PostgreSQL executes a query.

# 2. Analyze Query Plans

**Example Query:**

```
SELECT * FROM patients WHERE last_name = 'Smith’;
```

**Execution Plan:**

```
EXPLAIN ANALYZE SELECT * FROM patients WHERE last_name = 'Smith';
```

# 3. Optimize Indexes

Indexes can significantly

speed up query performance.

Ensure that appropriate

indexes are in place.

# 3. Optimize Indexes

**Example Index:**

CREATE INDEX idx_patients_last_name

ON patients (last_name);

# 3. Optimize Indexes

Re-run the query and

check the new execution plan

```
EXPLAIN ANALYZE SELECT * FROM patients WHERE last_name = 'Smith';
```

# 4. Refactor Queries

Refactor complex queries

for better performance.

Break down complex queries

into smaller, manageable parts

using Common Table Expressions (CTEs)

or temporary tables.

# Example with CTE:

```
WITH RecentVisits AS (
    SELECT patient_id, visit_date
    FROM visits
    WHERE visit_date > '2023-01-01'
)
```

# Example with CTE:

```
SELECT p.patient_id, p.first_name, p.last_name, rv.visit_date
FROM patients p
JOIN RecentVisits rv ON p.patient_id = rv.patient_id;
```

# 5. Monitor Performance

Continuously monitor

query performance

using tools like pg_stat_statements and

built-in PostgreSQL views.

# 5. Monitor Performance

**Example Monitoring Query:**

```
SELECT * FROM pg_stat_activity WHERE state = 'active';
```

# 6. Regular Maintenance

Perform regular maintenance tasks

such as vacuuming, analyzing, and

reindexing to keep the database

performance optimal.

# 6. Regular Maintenance

**Vacuum and Analyze:**

VACUUM ANALYZE;

**Reindexing:**

REINDEX TABLE patients;

# Example Scenarios

**Scenario 1:**

Retrieving Patient Records by Last Name

**Initial Query:**

```
SELECT * FROM patients WHERE last_name = 'Smith';
```

# Example Scenarios

**Optimization:**

Add an index on last_name:

CREATE INDEX idx_patients_last_name ON patients (last_name);

# Example Scenarios

**Revised Execution Plan:**

EXPLAIN ANALYZE SELECT * FROM patients WHERE last_name = 'Smith’;

Expect an index scan instead of a sequential scan, reducing execution time.

# Scenario 2: Joining Patient Visits with Doctors

```
SELECT p.patient_id, p.first_name, p.last_name,
v.visit_date, d.doctor_name
FROM patients p
JOIN visits v ON p.patient_id = v.patient_id
JOIN doctors d ON v.doctor_id = d.doctor_id
WHERE v.visit_date > '2023-01-01';
```

# Scenario 2: Joining Patient Visits with Doctors

**Optimization**

Ensure indexes on visits.patient_id and doctors.doctor_id:

CREATE INDEX idx_visits_patient_id ON visits (patient_id);

CREATE INDEX idx_doctors_doctor_id ON doctors (doctor_id);

# Revised Execution Plan

```
EXPLAIN ANALYZE SELECT p.patient_id,

p.first_name, p.last_name, v.visit_date, d.doctor_name

FROM patients p

JOIN visits v ON p.patient_id = v.patient_id

JOIN doctors d ON v.doctor_id = d.doctor_id

WHERE v.visit_date > '2023-01-01';
```

# Revised Execution Plan

Check for nested loop joins and

index scans,

indicating better performance.


**Surendra Panpaliya**

**Founder and CEO**

**GKTCS Innovations**

https://www.gktcs.com