PostgreSQL

Advanced Postgres

Surendra Panpaliya

Agenda

- Backup and Recovery
- Importance of backups
- Backup strategies (pg_dump, pg_basebackup)
- Restoring data from backups

Agenda

- Advanced SQL Queries
- Joins (inner, outer, cross)
- Subqueries and common table expressions (CTEs)
- Window functions

Backup and Recovery

Surendra Panpaliya

Backup and Recovery

Backups are critical

for ensuring

data integrity

availability.

Backup and Recovery

Provide a safeguard

against data loss

due to Hardware failures

Software bugs

Human errors

Backup and Recovery

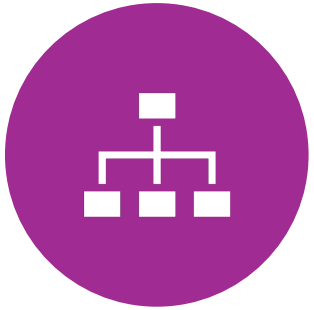
Regular backups	Allow to	Restore Database
To a Previous state	Minimize downtime	Data loss.

Importance of backups

DATA IS CRITICAL AND OFTEN SENSITIVE,

INCLUDING PATIENT RECORDS,

TREATMENT HISTORIES, DIAGNOSTIC RESULTS



ADMINISTRATIVE INFORMATION.

Patient Safety

ENSURING THE AVAILABILITY AND

INTEGRITY OF PATIENT RECORDS IS

CRUCIAL FOR PROVIDING CONTINUOUS AND

ACCURATE CARE.

Data Integrity and Compliance

HEALTHCARE PROVIDERS MUST COMPLY

WITH REGULATIONS SUCH AS HIPAA IN THE U.S.

WHICH MANDATE SECURE

RELIABLE HANDLING OF PATIENT INFORMATION.

Disaster Recovery

Backups provide a means	to restore data in the event of	Hardware failure
Cyber-attacks	Accidental deletions	Other unforeseen events

Operational Continuity

Ensuring

Administrative functions

Appointment scheduling

Billing

Continue smoothly

without data loss

Backup Examples

Surendra Panpaliya

Backup with pg_dump

Step 1: Backup Individual Tables

Backup the patients table

Backup with pg_dump

-- Backup the patients table

pg_dump -U postgres -d healthcare_db -t patients -F c -f

/path/to/backup/patients_table.backup

Backup the appointments table

pg_dump -U postgres -d healthcare_db -t appointments -F c -f

/path/to/backup/appointments_table.backup

Backup the appointments table

- By backing up
- Individual tables
- Ensure critical patient
- Appointment data is Secure
- Can be quickly restored if needed

Step 2: Backup the Entire Database

pg_dump -U postgres -d healthcare_db -F c -f

/path/to/backup/healthcare_db.backup

Step 2: Backup the Entire Database

A full database backup

captures all data, schemas

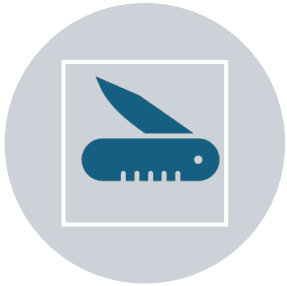
functions, allowing

a complete restoration

in case of a disaster

Backup with pg_basebackup

For full-cluster backups,



Useful for point-in-time recovery

Ensuring complete Data Integrity

Across multiple databases

Step 1: Create the Backup

pg_basebackup -U postgres -D

/path/to/backup/healthcare_cluster -Fp -Xs -P

Step 1: Create the Backup

Captures Entire cluster,

Including configuration files

Making it suitable for recovering

from severe failures

where entire instances

might need to be restored

Example Scenario: Disaster Recovery

Suppose there is
Some Attack
on the healthcare system
the data becomes
Inaccessible

Backup strategies

Provides several methods

for backing up data,

Suitable for

different Scenarios

Requirements

pg_dump

A UTILITY

FOR PERFORMING

LOGICAL

BACKUPS

pg_dump

DUMPS A DATABASE

INTO A SCRIPT OR

ARCHIVE FILE

CONTAINING SQL COMMANDS

TO RECREATE THE DATABASE

Use Cases

SUITABLE FOR

SMALLER DATABASES

LOGICAL BACKUPS

Syntax

pg_dump dbname > backupfile.sql

Example:

pg_dump mydatabase > mydatabase_backup.sql

pg_basebackup

A UTILITY

FOR PERFORMING

PHYSICAL

BACKUPS

pg_basebackup

CREATES A

BINARY COPY

OF THE DATABASE

CLUSTER'S FILES

pg_basebackup

Useful for

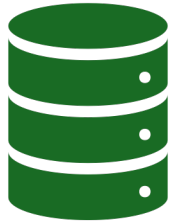
creating base backups

for streaming replication or

continuous archiving

Use Cases

Suitable for



larger databases

physical backups

Use Cases

You need

to restore

the entire

database cluster

Syntax

pg_basebackup -D /path/to/backupdir -Fp -Xs -P

Example:

pg_basebackup -D /var/lib/postgresql/data/backup -Fp -Xs -P

Continuous Archiving and Point-in-Time Recovery (PITR):

USES

PG_BASEBACKUP

FOR BASE BACKUPS

Continuous Archiving and Point-in-Time Recovery (PITR):

WITH CONTINUOUS

WAL (WRITE-AHEAD LOGGING) ARCHIVING

ENABLE RECOVERY

ANY POINT IN TIME

Use Cases

Suitable for environments

Requiring high availability

Minimal data loss.

Configuration

Enable WAL archiving in postgresql.conf:

archive_mode = on

archive_command = 'cp %p /path/to/archive/%f'

Restoring data from backups

Surendra Panpaliya

Restoring from pg_dump

Restore a Database

psql dbname < backupfile.sql

Restore a Database

```
psql mydatabase < mydatabase_backup.sql
```

Restoring from pg_basebackup

Steps:

1. Stop the PostgreSQL server.
2. Clear the existing data directory.
3. Restore the backup files.
4. Start the PostgreSQL server.

Restoring from pg_basebackup:

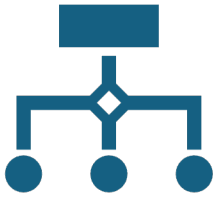
Example:

```
pg_ctl stop -D /var/lib/postgresql/data
rm -rf /var/lib/postgresql/data/*
cp -r /path/to/backup/* /var/lib/postgresql/data/
pg_ctl start -D /var/lib/postgresql/data
```

Point-in-Time Recovery (PITR)

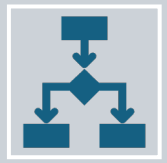
Restore the base backup

Restore the archived WAL files



Start the PostgreSQL server in recovery mode.

Example Configuration



Create a recovery command in recovery.conf

restore_command = 'cp /path/to/archive/%f %p’

Example Configuration

Start the server in recovery mode

```
pg_ctl start -D /var/lib/postgresql/data -m recovery
```

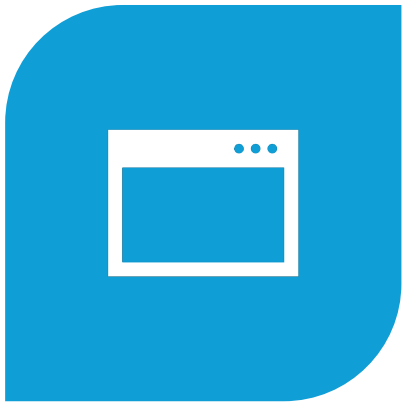
Best Practices

Surendra Panpaliya

Regular Backups

SCHEDULE REGULAR BACKUPS

BASED ON THE CRITICALITY OF THE DATA AND



THE ACCEPTABLE DATA LOSS WINDOW.

Automate Backups

USE CRON JOBS

SCHEDULING TOOLS

TO AUTOMATE

BACKUP PROCESSES

Verify Backups

REGULARLY TEST

BACKUPS

TO ENSURE

RESTORE

Secure Backups

Store backups in

Secure location

with restricted access

use encryption

if necessary

Keep Multiple Copies

STORE MULTIPLE COPIES OF BACKUPS IN

DIFFERENT PHYSICAL OR CLOUD LOCATIONS

TO PROTECT AGAINST DATA LOSS.

Agenda

- Advanced SQL Queries
- Joins (inner, outer, cross)
- Subqueries and Common Table Expressions (CTEs)
- Window functions

Joins (inner, outer, cross)

patients:

Stores patient information

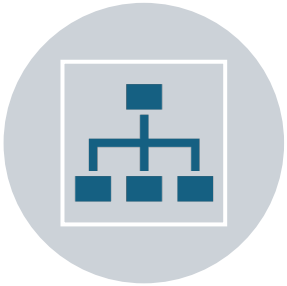
doctors:

Stores doctor information

Joins (inner, outer, cross)

appointments:

Stores information about patient appointments with doctors.



departments:

Stores information about hospital departments.

Create patients table

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

Create doctors table

```
CREATE TABLE doctors (
    doctor_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    department_id INTEGER
);
```

Create departments table

```
CREATE TABLE departments (
    department_id SERIAL PRIMARY KEY,
    department_name VARCHAR(50)
);
```

Create appointments table

```
CREATE TABLE appointments (
    appointment_id SERIAL PRIMARY KEY,
    patient_id INTEGER REFERENCES patients(patient_id),
    doctor_id INTEGER REFERENCES doctors(doctor_id),
    appointment_date DATE,
    status VARCHAR(20)
);
```

Inserting Sample Data

```
-- Insert data into departments
INSERT INTO departments (department_name) VALUES
('Cardiology'), ('Neurology'), ('Pediatrics'), ('Oncology');
```

-- Insert data into doctors

INSERT INTO doctors (first_name, last_name, department_id) VALUES ('John', 'Doe', 1), ('Jane', 'Smith', 2), ('Alice', 'Johnson', 3), ('Bob', 'Brown', 4);

Insert data into patients

```
INSERT INTO patients (first_name, last_name, date_of_birth, gender)
VALUES
('Michael', 'Jackson', '1960-08-29', 'M'),
('Elvis', 'Presley', '1935-01-08', 'M'),
('Marilyn', 'Monroe', '1926-06-01', 'F');
```

-- Insert data into appointments

INSERT INTO appointments (patient_id, doctor_id, appointment_date, status) VALUES

(1, 1, '2024-07-01', 'Completed'),

(1, 2, '2024-07-02', 'Scheduled'),

(2, 1, '2024-07-03', 'Cancelled'),

(3, 3, '2024-07-04', 'Completed');

Join Queries

INNER JOIN	Fetch information	about appointments
along with	patient and	doctor details

INNER JOIN

```
SELECT
    p.first_name AS patient_first_name,
    p.last_name AS patient_last_name,
    d.first_name AS doctor_first_name,
    d.last_name AS doctor_last_name,
    a.appointment_date,
    a.status
```

INNER JOIN

```
FROM
    appointments a
INNER JOIN patients p ON a.patient_id = p.patient_id
INNER JOIN doctors d ON a.doctor_id = d.doctor_id;
```

2. LEFT OUTER JOIN

Fetch all patients

appointments

include patients

with no appointments

2. LEFT OUTER JOIN

```
SELECT
    p.first_name AS patient_first_name,
    p.last_name AS patient_last_name,
    a.appointment_date,
    a.status
```

2. LEFT OUTER JOIN

```
FROM
    patients p
LEFT JOIN appointments a ON p.patient_id = a.patient_id;
```

3. RIGHT OUTER JOIN

Fetch all appointments

Corresponding patient information

Including appointments

Without patient details

3. RIGHT OUTER JOIN

```
SELECT
    p.first_name AS patient_first_name,
    p.last_name AS patient_last_name,
    a.appointment_date,
    a.status
```

3. RIGHT OUTER JOIN

```
FROM
    patients p
RIGHT JOIN appointments a ON p.patient_id = a.patient_id;
```

4. FULL OUTER JOIN

Fetch all patients

their appointments,

including those

with no matching records

on either side.

4. FULL OUTER JOIN

```
SELECT
    p.first_name AS patient_first_name,
    p.last_name AS patient_last_name,
    a.appointment_date,
    a.status
```

4. FULL OUTER JOIN

```
FROM
    patients p
FULL OUTER JOIN appointments a ON p.patient_id = a.patient_id;
```

5. CROSS JOIN

Fetch every possible

combination of

patients and doctors

5. CROSS JOIN

Rarely useful

in practice

Can be used

for analysis

5. CROSS JOIN

```
SELECT
    p.first_name AS patient_first_name,
    p.last_name AS patient_last_name,
    d.first_name AS doctor_first_name,
    d.last_name AS doctor_last_name
```

5. CROSS JOIN

```
FROM
   patients p
CROSS JOIN doctors d;
```

Summary

- Queries showcase
- How to combine data
- from multiple tables
- to retrieve
- comprehensive information

Summary

Essential for

Effective

Database management

Reporting

in the healthcare domain.

Subqueries and common table expressions (CTEs)

Can be used

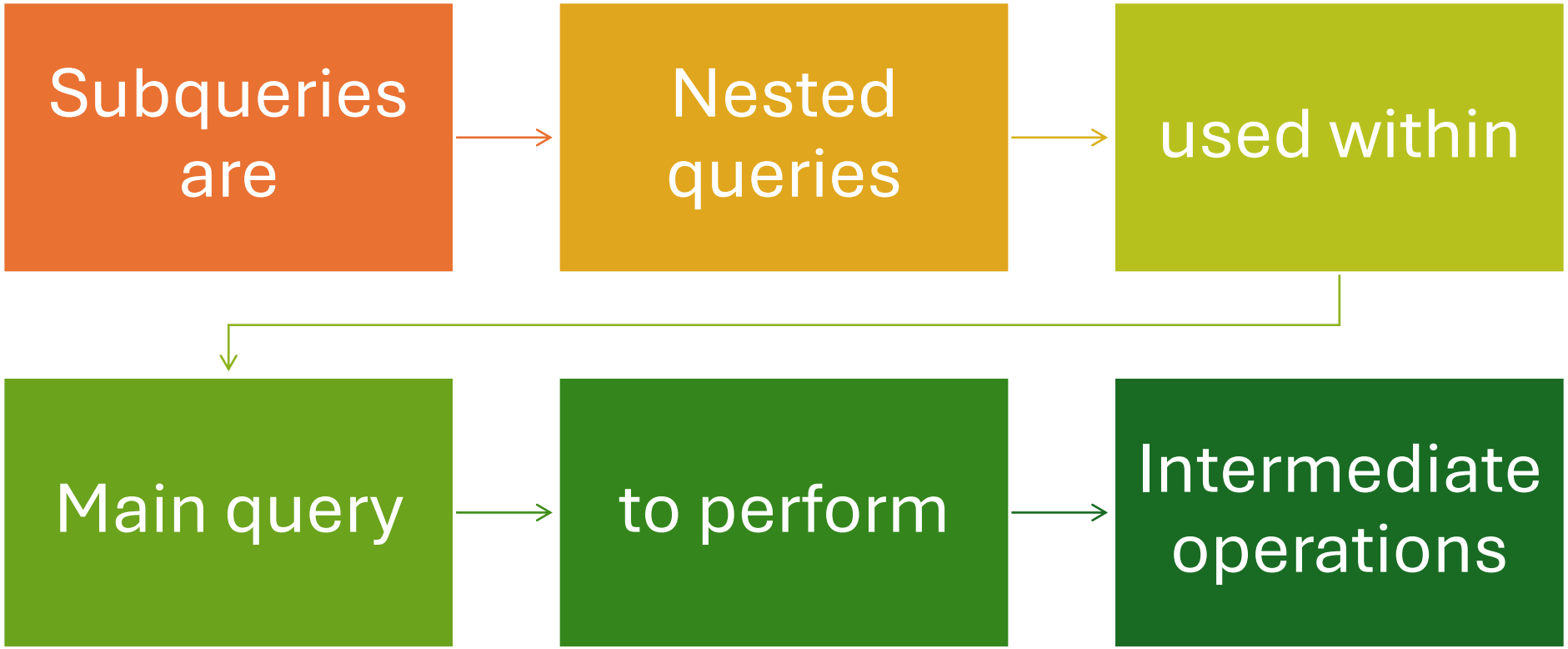
to simplify

complex queries

make them

more readable

Subqueries


Subqueries are
Nested queries
used within
Main query
to perform
Intermediate operations

Example 1: Subquery in SELECT

Fetch

patient names

along with

number of appointments

they have.

Example 1: Subquery in SELECT

```
SELECT
    p.first_name,
    p.last_name,
    (SELECT COUNT(*)
     FROM appointments a
     WHERE a.patient_id = p.patient_id) AS appointment_count
FROM
    patients p;
```

Example 2: Subquery in WHERE

Fetch appointments

of patients

who have

more than

one appointment

Example 2: Subquery in WHERE

```
SELECT
    a.appointment_id,
    a.patient_id,
    a.doctor_id,
    a.appointment_date,
    a.status
```

Example 2: Subquery in WHERE

```
FROM
   appointments a
WHERE
   a.patient_id IN (
      SELECT
         patient_id
      FROM
         appointments
```

Example 2: Subquery in WHERE

```
GROUP BY
        patient_id
    HAVING
        COUNT(*) > 1
  );
```

Common Table Expressions (CTEs)

Provide a way

To break down

Complex queries

By defining

Temporary result sets

Common Table Expressions (CTEs)

- Can be
- Referenced
- within
- Main query

Example 1: Simple CTE

Fetch patient names

their total number of

appointments

using a CTE

Example 1: Simple CTE

```
WITH patient_appointments AS (
    SELECT
        patient_id,
        COUNT(*) AS appointment_count
    FROM
        appointments
    GROUP BY
        patient_id
)
```

Example 1: Simple CTE

```
SELECT
    p.first_name,
    p.last_name,
    pa.appointment_count
FROM
    patients p
JOIN
    patient_appointments pa ON p.patient_id = pa.patient_id;
```

Example 2: Recursive CTE

Find all patients

who have referred

other patients

Example 2: Recursive CTE

CREATE

THE REFERRALS TABLE

TO STORE

REFERRAL

RELATIONSHIPS

Example 2: Recursive CTE

```
CREATE TABLE referrals (
    referrer_id INTEGER REFERENCES patients(patient_id),
    referred_id INTEGER REFERENCES patients(patient_id)
);
```

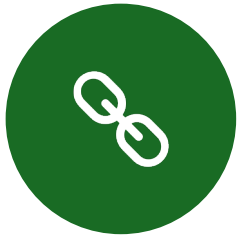
Example 2: Recursive CTE

```
-- Insert sample data into referrals table

INSERT INTO referrals (referrer_id, referred_id) VALUES
(1, 2), (2, 3), (3, 4);
```

Example 2: Recursive CTE

USE A RECURSIVE CTE



TO FIND THE CHAIN OF

REFERRALS

STARTING FROM

A SPECIFIC PATIENT.

Example 2: Recursive CTE

```
WITH RECURSIVE referral_chain AS (
    SELECT
        referrer_id,
        referred_id,
        1 AS level
    FROM
        referrals
```

Example 2: Recursive CTE

```
WHERE
      referrer_id = 1
   UNION
   SELECT
      r.referrer_id,
      r.referred_id,
      rc.level + 1
   FROM
      referrals r
   INNER JOIN
      referral_chain rc ON r.referrer_id = rc.referred_id
)
```

Example 2: Recursive CTE

```
SELECT
    referrer_id,
    referred_id,
    level
FROM
    referral_chain;
```

Example 3: CTE for Joining Multiple Tables

Fetch patient names

Doctor names

Appointment details

Using CTEs

To simplify the query

Example 3: CTE for Joining Multiple Tables

```
WITH patient_details AS (
    SELECT
        patient_id,
        first_name AS patient_first_name,
        last_name AS patient_last_name
    FROM
        patients
),
```

Example 3: CTE for Joining Multiple Tables

```
doctor_details AS (
    SELECT
        doctor_id,
        first_name AS doctor_first_name,
        last_name AS doctor_last_name
    FROM
        doctors
)
```

Example 3: CTE for Joining Multiple Tables

```
SELECT
    p.patient_first_name,
    p.patient_last_name,
    d.doctor_first_name,
    d.doctor_last_name,
    a.appointment_date,
    a.status
```

Example 3: CTE for Joining Multiple Tables

```
FROM
   appointments a
JOIN
   patient_details p ON a.patient_id = p.patient_id
JOIN
   doctor_details d ON a.doctor_id = d.doctor_id;
```

Summary

- Subqueries
- Useful for
- performing
- intermediate operations
- within a main query

Summary

CTEs

Simplify

complex queries

by defining

temporary

result sets

Summary

- Can be
- referenced
- within
- Main query

Summary

Can be

recursive

to handle

hierarchical data

Window functions

Provide a way

to perform calculations

across a set of table rows

that are related

to the current row.

Window functions

Useful for performing

Aggregations

Calculations

Over subsets of data

Window Functions Examples

Example 1: ROW_NUMBER()

Assign a unique row number

to each appointment

for a specific patient

Example 1: ROW_NUMBER()

```
SELECT
    patient_id,
    doctor_id,
    appointment_date,
    status,
    ROW_NUMBER() OVER (PARTITION BY patient_id ORDER BY appointment_date) AS row_num
FROM
    appointments;
```

Example 2: RANK()

Rank doctors

based on the number of

appointments they have.

Ties will receive

the same rank.

Example 2: RANK()

```
SELECT
   doctor_id,
   COUNT(*) AS appointment_count,
   RANK() OVER (ORDER BY COUNT(*) DESC) AS rank
FROM
   appointments
GROUP BY
   doctor_id;
```

Example 3: DENSE_RANK()

Similar to RANK()

But without gaps

in ranking numbers.

Example 3: DENSE_RANK()

```
SELECT
    doctor_id,
    COUNT(*) AS appointment_count,
    DENSE_RANK() OVER (ORDER BY COUNT(*) DESC) AS
dense_rank
FROM
    appointments
GROUP BY
    doctor_id;
```

Example 4: NTILE()

Divide the patients

into four quartiles

based on

their number

of appointments

Example 4: NTILE()

```
SELECT
    patient_id,
    COUNT(*) AS appointment_count,
    NTILE(4) OVER (ORDER BY COUNT(*) DESC) AS quartile
FROM
    appointments
GROUP BY
    patient_id;
```

Example 5: LAG()

Fetch

previous appointment details

for each patient

Example 5: LAG()

```
SELECT
    patient_id,
    appointment_date,
    LAG(appointment_date) OVER (PARTITION BY patient_id ORDER BY appointment_date) AS previous_appointment_date
FROM
    appointments;
```

Example 6: LEAD()

FETCH

NEXT APPOINTMENT DETAILS

FOR EACH PATIENT.

Example 6: LEAD()

```
SELECT
    patient_id,
    appointment_date,
    LEAD(appointment_date) OVER (PARTITION BY patient_id ORDER BY appointment_date) AS next_appointment_date
FROM
    appointments;
```

Example 7: SUM()

Calculate

the running

total of appointments

for each patient

over time

Example 7: SUM()

```
SELECT
    patient_id,
    appointment_date,
    SUM(1) OVER (PARTITION BY patient_id ORDER BY
appointment_date) AS running_total
FROM
    appointments;
```

Example 8: AVG()

Calculate

the average number of appointments

per patient

over a rolling window

of the last three appointments.

Example 8: AVG()

```
SELECT
    patient_id,
    appointment_date,
    AVG(1) OVER (PARTITION BY patient_id ORDER BY
appointment_date ROWS BETWEEN 2 PRECEDING AND CURRENT
ROW) AS rolling_avg
FROM
    appointments;
```

Summary

Window functions

allow for

powerful

flexible

data analysis

Summary

Can be used

To Assign

Row numbers or

Ranks within

Partitions of data

Summary

Calculate

Running totals,

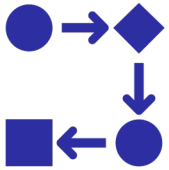
Moving averages

Other cumulative

Statistics

Summary

Fetch previous or



next values in a sequence,

useful for time-series data

like appointment histories.

Summary

Learned

How to use window functions

To perform advanced

Data analysis

Reporting in

a healthcare database

Surendra Panpaliya

Founder and CEO

GKTCS Innovations

https://www.gktcs.com