PostgreSQL

# Joins, Subqueries & CTEs

Surendra Panpaliya

# Agenda

- INNER, LEFT, RIGHT, FULL joins
- Subqueries in SELECT and FROM
- CTE (WITH) usage and optimization

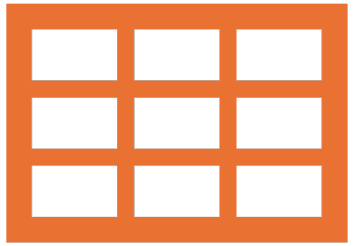
# Hands-On

Execute customer order reports using joins and CTEs

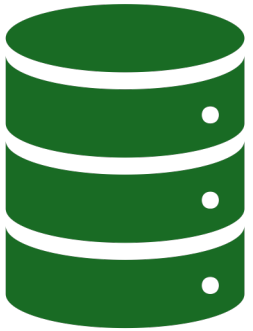
**Assignment:**

Write a multi-level CTE and compare it with subquery

# JOINs in PostgreSQL



Combine data from multiple tables



Retrieve related information in a single query

# What is a JOIN?

A **JOIN** combines rows from

two or more tables

**based on a related column**,

usually a **foreign key**.

# Why Learn Joins?

Essential for **relational database queries**

Enables **multi-table analysis** (customer + billing + plan)

Helps build **dashboards, APIs, reports, and alerts**

# Types of JOINs

| Join Type | Description |
| --- | --- |
| **INNER JOIN** | Matching rows in both tables |
| **LEFT JOIN** | All rows from left table + matching from right |
| **RIGHT JOIN** | All rows from right table + matching from left |
| **FULL OUTER JOIN** | All rows from both tables (matched/unmatched) |

# Example Scenario

| Table | Columns |
| --- | --- |
| **customer** | customer_id, name, city |
| **orders** | order_id, customer_id, product, amount |

# Create Sample Tables

```
CREATE TABLE customer (
    customer_id INT PRIMARY KEY,
    name VARCHAR(50),
    city VARCHAR(50)
);
```

# Create Sample Tables

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT REFERENCES customer(customer_id),
    product VARCHAR(50),
    amount NUMERIC(10,2)
);
```

# Insert Data

INSERT INTO customer VALUES (1, 'John', 'Pune'), (2, 'Alice', 'Mumbai'), (3, 'Bob', 'Delhi');

INSERT INTO orders VALUES (101, 1, 'Mobile Plan', 299.99), (102, 1, 'Broadband', 499.99), (103, 2, 'DTH', 199.99);

# INNER JOIN

```
SELECT c.name, o.product, o.amount
FROM customer c
INNER JOIN orders o ON c.customer_id = o.customer_id;
```

Result: Only customers with orders.

# LEFT JOIN

```
SELECT c.name, o.product, o.amount
FROM customer c
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

Result: All customers + orders if available.

Nulls where no order.

# RIGHT JOIN

```
SELECT c.name, o.product, o.amount
FROM customer c
RIGHT JOIN orders o ON c.customer_id = o.customer_id;
```

Result: All orders + customer info if available.

# FULL OUTER JOIN

```
SELECT c.name, o.product, o.amount
FROM customer c
FULL OUTER JOIN orders o ON c.customer_id = o.customer_id;
```

Result: All customers and all orders (matched and unmatched).

# What Are Subqueries?

A **SUBQUERY** IS A QUERY

**NESTED INSIDE**

**ANOTHER QUERY.**

# What is a Subquery?

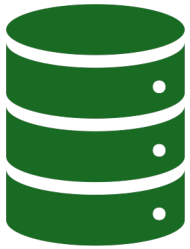
A **subquery** is a query

embedded **inside another query**

(in SELECT, FROM, or WHERE clause).

# Why Learn Subqueries?

Helpful for **modular querying**



Ideal for **filtering based on computed data**

Useful for **reusability & readability**

# Example: Subquery in WHERE

```
SELECT name, city
FROM telecom.customer
WHERE customer_id IN (
    SELECT customer_id
    FROM telecom.subscriptions
    WHERE plan_id = 5
);
```

**Use Case**: Find all customers who are on **plan ID 5**.

# Example: Scalar Subquery

```
SELECT name,
    (SELECT COUNT(*) FROM telecom.subscriptions s WHERE s.customer_id = c.customer_id) AS subscription_count
FROM telecom.customer c;
```

**Use Case**: Show how many subscriptions each customer has.

# What Are Subqueries?

- Allows subqueries in different parts of a query
- In the **SELECT** clause
- In the **FROM** clause
- In the **WHERE** clause (common)
- In **JOIN** conditions (advanced)

# Subquery in SELECT Clause

A subquery in the **SELECT** clause

returns a single value (scalar) **per row**.

# Syntax

```
SELECT
    column1,
    (SELECT ... WHERE condition matching column1) AS computed_column
FROM main_table;
```

# Example

Show **each customer's**

**name and number**

**of subscriptions:**

# Example

```
SELECT
    name,
    (SELECT COUNT(*)
     FROM telecom.subscriptions s
     WHERE s.customer_id = c.customer_id) AS subscription_count
FROM telecom.customers c;
```

# Subquery in FROM Clause

A subquery in
the **FROM** clause
acts like a temporary table
or inline view.

# Syntax

```
SELECT ...
FROM (
  SELECT ... FROM ... WHERE ...
) AS alias
```

# Example

Find **cities with more than 2 customers**:

```
SELECT city, city_count
FROM (
    SELECT city, COUNT(*) AS city_count
    FROM telecom.customers
    GROUP BY city
) AS customer_city
WHERE city_count > 2;
```

# Why Use Subqueries?

| Purpose | Why It Helps |
| --- | --- |
| **Break complex queries** | Easier to read and debug |
| **Perform per-row calculations** | Compute aggregates inline |
| **Filter summarized data** | Use GROUP BY + HAVING logic |
| **Reuse logic temporarily** | Acts like a virtual table |
| **Replace stored views** | Temporary result without creating a view |

# Real-World Use Cases for Amdocs

| Use Case | Where | How Subqueries Help |
| --- | --- | --- |
| **1. Subscription Count** | SELECT | Show # of plans per customer |
| **2. Top Cities** | FROM | Find cities with most users |
| **3. High-Value Customers** | FROM | Filter customers based on total balance or recent payments |

# Real-World Use Cases for Amdocs

| Use Case | Where | How Subqueries Help |
|---|---|---|
| **4. Billing Trend** | SELECT | Fetch latest payment or activity per customer |
| **5. Customer with No Subscriptions** | WHERE NOT IN | Subquery helps filter such customers |
| **6. Combine Usage + Payment** | FROM | Join summary subqueries for usage and payment metrics |

# Pro Tip

PostgreSQL encourages:

**CTEs** (WITH clause) for clarity when subqueries get complex

**Subqueries in SELECT** for row-wise calculations

**Subqueries in FROM** for inline summaries

# Summary

| Subquery Type | When to Use |
| --- | --- |
| **SELECT clause** | Per-row aggregate or lookup |
| **FROM clause** | Summary results, filtering aggregates |

# Subqueries in the WHERE Clause

Used to **filter results**
based on another
query’s output.

# Subqueries in the WHERE Clause

SELECT ...

FROM table1

WHERE column IN (SELECT column FROM table2 WHERE condition);

# Example 1: Customers who have a subscription

```
SELECT name
FROM telecom.customers
WHERE customer_id IN (
    SELECT customer_id
    FROM telecom.subscriptions
);
```

# Example 2: Customers without any subscription

```
SELECT name
FROM telecom.customers
WHERE customer_id NOT IN (
   SELECT customer_id
   FROM telecom.subscriptions
);
```

# Example 3: Customer with maximum balance

```
SELECT name, balance
FROM telecom.customers
WHERE balance = (
    SELECT MAX(balance)
    FROM telecom.customers
);
```

# Use Cases at Amdocs

| Use Case | Why Use Subquery |
| --- | --- |
| **Find customers with activity** | Filter from activity table |
| **List subscribers of specific plan** | Use plan_id filter |
| **Customers with highest billing** | Aggregate filter using MAX() |

# Subqueries in JOIN Conditions (Advanced)

A **subquery in a JOIN condition**
allows you to dynamically join
on a derived or aggregated result.

# Syntax

```
SELECT ...
FROM table1 t1
JOIN (
    SELECT ... FROM table2 WHERE ...
) AS sub ON t1.column = sub.column;
```

# Example 1: Join with customers who have recent subscriptions

```
SELECT c.name, s.start_date
FROM telecom.customers c
JOIN (
    SELECT customer_id, MAX(start_date) AS start_date
    FROM telecom.subscriptions
    GROUP BY customer_id
) AS s ON c.customer_id = s.customer_id;
```

# Example 2: Join with plan details of customers who paid more than 500

```
SELECT c.name, p.plan_name
FROM telecom.customers c
JOIN telecom.subscriptions s ON c.customer_id = s.customer_id
JOIN (
   SELECT plan_id
   FROM telecom.subscriptions
   WHERE customer_id IN (
      SELECT customer_id
      FROM telecom.customers
      WHERE balance > 500
   )
) AS high_value ON s.plan_id = high_value.plan_id
JOIN telecom.plans p ON p.plan_id = s.plan_id;
```

# Use Cases at Amdocs

| Use Case | Benefit |
| --- | --- |
| **Recent recharge or subscription** | Filter only latest usage |
| **Dynamic filter by payment or usage trend** | Join only relevant customers |
| **Advanced reporting with temporary filters** | Combine summaries with live tables |

# Summary Table

| Subquery Location | When to Use | Common In |
| --- | --- | --- |
| **WHERE clause** | Filter rows using another query | Filtering IDs, max values, missing data |
| **JOIN condition** | Join with aggregated or derived results | Latest subscription, high-value users |

# Tip for Oracle Developers

- PostgreSQL subqueries work like Oracle,
- but **encourage more use of CTEs (WITH)** for readability.
- Subqueries in JOINs are powerful for
- **reporting, summarization, and analytics**.

# Subqueries

Subquery in SELECT

Get customer name + total amount spent:

# Subquery in SELECT

```
SELECT name,
    (SELECT SUM(amount) FROM orders o WHERE
o.customer_id = c.customer_id) AS total_spent
FROM customer c;
```

# Subquery in FROM

```
SELECT customer_id, total
FROM (
    SELECT customer_id, SUM(amount) AS total
    FROM orders
    GROUP BY customer_id
) AS order_totals;
```

# Summary

| Query Pattern | Use For |
| --- | --- |
| **Subquery in WHERE** | Filter using dynamic results |
| **Subquery in SELECT** | Embed calculated values |
| **Subquery in FROM** | Build complex comparisons |
| **Correlated Subquery** | Row-wise comparisons |
| **NOT IN Subquery** | Churn analysis |
| **HAVING + Subquery** | Ranking + Aggregations |

# CTE (Common Table Expression)

**Why Use CTE?**

Improve **readability**

Simplify **complex joins or subqueries**

Reuse derived tables in the same query

# Syntax:

```
WITH cte_name AS (
   SELECT ...
)
SELECT * FROM cte_name;
```

# Example: CTE for Total Spending

```
WITH customer_spending AS (
    SELECT customer_id, SUM(amount) AS total_spent
    FROM orders
    GROUP BY customer_id
)
SELECT c.name, COALESCE(cs.total_spent, 0) AS total_spent
FROM customer c
LEFT JOIN customer_spending cs ON c.customer_id =
cs.customer_id;
```

# CTE vs Subquery: When to Use

| Use Case | Recommended Approach |
| --- | --- |
| **Simple scalar values** | Subquery in SELECT |
| **Aggregation reuse / recursive logic** | CTE |
| **Complex filters and multiple joins** | CTE (WITH clause) |
| **One-time derived data** | Subquery in FROM |

# Recursive CTE Example (Bonus)

If you need hierarchical queries (like Oracle CONNECT BY):

```
WITH RECURSIVE numbers AS (
    SELECT 1 AS n
    UNION ALL
    SELECT n + 1 FROM numbers WHERE n < 5
)
SELECT * FROM numbers;
```

Output: 1, 2, 3, 4, 5

# Summary Table

| Feature | Oracle SQL/PLSQL | PostgreSQL |
|---|---|---|
| Joins | Same syntax | Same syntax |
| Subqueries | Same | Same |
| CTE | WITH clause | Same |
| Recursive Queries | CONNECT BY | WITH RECURSIVE |

# Key Takeaways

PostgreSQL **joins** and **subqueries** work similarly to Oracle SQL

**CTEs** improve readability and maintainability

**Recursive CTEs** replace Oracle’s CONNECT BY

# Hands-On

Execute customer order reports using joins and CTEs

# Scenario

- You work with **Customer and Order data**.
- Generate **Customer Order Reports** using:

| Feature | Practice |
| --- | --- |
| INNER JOIN | ✅ |
| LEFT JOIN | ✅ |
| CTE (WITH clause) | ✅ |

# Assignment

## Write a Multi-Level CTE and Compare It with Subquery

# Objective

| Skill | Practice |
| --- | --- |
| Use multi-level CTEs (chained WITH clauses) | ✅ |
| Rewrite the same logic using subqueries | ✅ |
| Understand readability and optimization differences | ✅ |

# Scenario: Customer-Order Analysis

| Table | Columns |
| --- | --- |
| **customer** | customer_id, name, city |
| **orders** | order_id, customer_id, product, amount |

# Compare CTE vs Subquery

| Aspect | Multi-Level CTE | Subquery |
|---|---|---|
| **Readability** | Easy to break into steps | Harder to read for complex logic |
| **Debugging** | Can test each CTE independently | Not modular |
| **Performance** | Similar for simple queries | Similar |
| **Use Case** | Best for multi-step pipelines or recursion | Good for single-use subqueries |

# 🔍 Why Use CTEs?

| Reason | Benefits |
| --- | --- |
| ✅ **Readability** | Break complex queries into logical blocks |
| ✅ **Reusability** | Reference CTEs multiple times |
| ✅ **Modularity** | Chain multiple CTEs (step-by-step logic) |
| ✅ **Recursive logic** | Solve hierarchical/tree problems |
| ✅ **Replace views** | Temporary without cluttering DB schema |

# CTE vs Subquery

| Feature | Subquery | CTE |
| --- | --- | --- |
| **Readability** | Lower | Higher |
| **Naming** | Not reusable | Named result |
| **Multiple Usage** | Repeated computation | Reusable within the query |
| **Recursive Support** | ❌ | ✅ |

# ⚙️ CTE Optimization Tips

| Tip | Reason |
| --- | --- |
| **Use indexes on columns used in JOIN or WHERE** | Boosts speed |
| **Prefer CTE over subqueries when reused multiple times** | Improves clarity |
| **Avoid recursive CTEs unless necessary** | Slower on large datasets |
| **Use WITH ... only for modular queries** | Not needed for simple SELECTs |

# Summary

| Feature | Purpose | Use Case |
|---|---|---|
| CTE (WITH) | Modularize SQL | Group filters + joins |
| Aggregation CTE | Clean up complex summaries | Total data/call usage |
| Filtering CTE | Highlight high usage or churn | WHERE on CTE output |
| Recursive CTE | Generate time windows or hierarchies | Monthly billing periods |

# Comparison: CTE vs Subquery

| Aspect | Multi-Level CTE | Nested Subquery |
|---|---|---|
| **Readability** | ✅ Very clear and modular | ❌ Becomes complex for multiple metrics |
| **Reusability** | ✅ You can reuse CTEs inside the query | ❌ Subqueries repeat logic for each column |
| **Performance** | ✅ Better for large joins with aggregation | ❌ Re-evaluates subqueries per row (unless optimized by planner) |
| **Maintainability** | ✅ Easier to debug and scale | ❌ Harder to modify if logic grows |
| **Use Case Fit** | Best for reporting, complex joins | Good for simple value lookups |

# Agenda

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

# Agenda

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

# Joins (inner, outer, cross)

patients:

Stores patient information

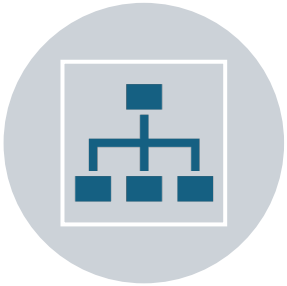
doctors:

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

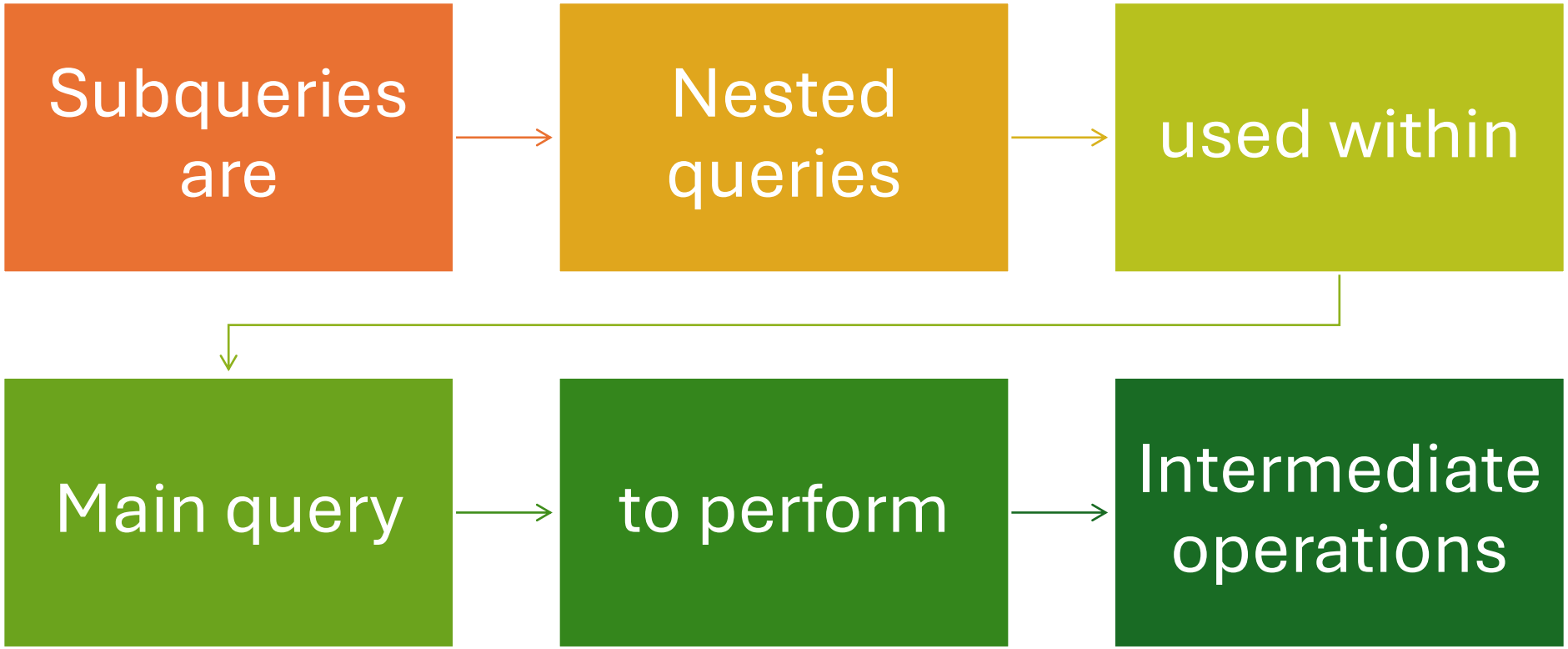
to simplify

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)
);
```

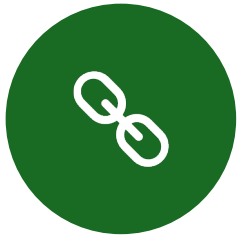
# Example 2: Recursive CTE

```
-- 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


**Surendra Panpaliya**

**Founder and CEO**

**GKTCS Innovations**

https://www.gktcs.com