Subquery in WHERE Clause

To conceptulize Subquery in the WHERE Clause, & know when to use them effectively with operators aggregates & design logic, follow these Core notes.

- What is a Subquery in a WHERE Clause ?

- A Subquery in the WHERE clause is used to filter results based on a condition that involves another query.
- The Subquery returns values compared with outer query rows using Comparison or membership operators.

🧠 When to Use a Subquery in WHERE Clause

Use when:

- You need to compare a value from the outer query with **a computed result** (e.g., MAX, COUNT).
- You want to **filter using a dynamic list** from another table.
- Joins would be **less readable** or infeasible due to aggregation or nested logic.
- You want to check for **existence** of certain conditions (using `EXISTS`).

⚙️ Key Operators and When to Use Them

`IN`

- **Use when** comparing a column to a **list of values** returned by subquery.
- Subquery must return **one column, multiple rows**.
- Great for membership tests:

```sql
WHERE id IN (SELECT user_id FROM logins)
```

`NOT IN`

- Avoid if subquery can return **NULLs**—results become unpredictable.

`EXISTS`

- **Use when** you're testing for the **existence of rows**, not values.
- Subquery checks: “Is there at least one row that satisfies this condition?”
- Fast and efficient in many databases.
- Doesn't depend on what's SELECTed inside (usually use `SELECT *`) Beggining SQL .

`ANY` **or** `SOME`

- **Use when** comparing a value against **any** value in subquery result.
- Supports operators: `=`, `<`, `>`, etc.
- Example:

```sql
WHERE salary > ANY (SELECT salary FROM dept)
```

`ALL`

- **Use when** value must satisfy condition with **all** rows from subquery.
- Useful for range constraints:

```sql
WHERE score < ALL (SELECT score FROM exams)
```

📊 Using Aggregate Functions

- Use aggregate functions like `MAX`, `MIN`, `AVG`, `COUNT` in subqueries when you need to:
  - Compare against an **aggregate result** (e.g., “get films above average price”)
  - Ensure subquery returns **a single scalar value**
- Must be used where **a single value** is expected:

```sql
WHERE price = (SELECT MAX(price) FROM products)
```

🚫 When Not to Use Subqueries

Avoid subqueries in WHERE clause when:

- A **JOIN is clearer or faster** (especially in large datasets).
- The subquery returns **multiple columns** (unless using `EXISTS`).
- You need to reuse the subquery result in multiple places — better to use a CTE or derived table.

🧩 Tips to Decide What to Use

Scenario	Use This
Compare to a list	`IN`, `NOT IN`
Need true/false for existence	`EXISTS`, `NOT EXISTS`
Compare with a single value result	scalar subquery with `=`, `<`, etc.
Compare with every item in list	`ALL`
Compare with at least one in list	`ANY`, `SOME`
Join multiple related tables	JOINs (preferably over correlated subqueries)

Subquery in FROM Clause

• A Subquery in the FROM Clause is used to create a temp virtual table (called derived table) that can be queried by the outer/main query.

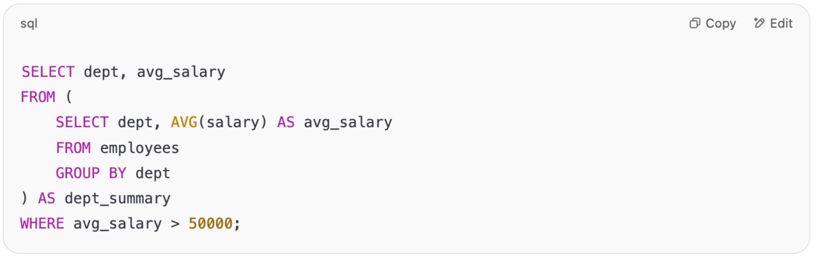
*Why use it?

• You use Subqueries in the FROM Clause when you want to perform a query on the result of another query - especially when that inner query needs to Summarize, group or Prepare the data in a Specific format first.

Ex 1: Using a Subquery to find department averages

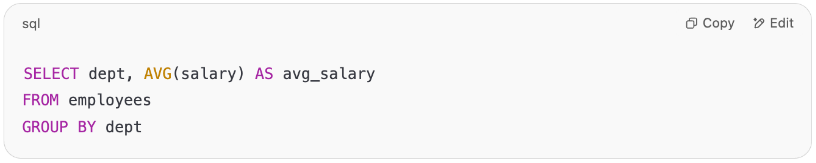
• Suppose you want to calculate the average salary for each department, & then only show departments where the average is more than $ 50,000

• You can do this in two steps using a Subquery in the FROM clause.



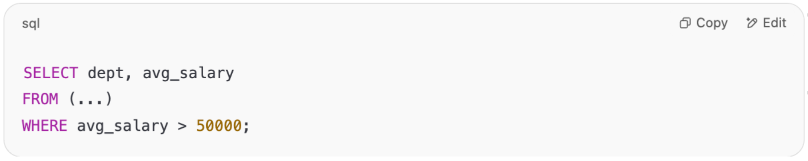
```sql
SELECT dept, avg_salary
FROM (
    SELECT dept, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY dept
) AS dept_summary
WHERE avg_salary > 50000;
```

What's happening: The Subquery groups employees by department & calculates average Salary.



```sql
SELECT dept, AVG(salary) AS avg_salary
FROM employees
GROUP BY dept
```

The outer-query selects only those dept where the average Salary is above the threshold



```sql
SELECT dept, avg_salary
FROM (...)
WHERE avg_salary > 50000;
```

- How it differs from WHERE Clause?

- Lets solve the same problem - find emp whose salary is above the overall average
- In WHERE Clause the subquery returns single value (average salary)
- The outer query filters employees with salaries above this value.

```sql
SELECT name, salary
FROM employees
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
);
```

🔁 Comparing the Two Approaches

Feature	`FROM` Subquery	`WHERE` Subquery
Purpose	Creates a temporary table for further queries	Filters data using a result from a subquery
Must Have Alias?	✅ Yes	❌ No
Return Type	Table (multiple rows/columns)	Single value or list (1 column only)
Can Be Correlated?	❌ No	✅ Yes
When to Use	When data needs pre-processing before querying	When conditionally filtering based on a value

Use a subquery in the `FROM` clause when:

- You want to do *more* with the subquery result — like joining or filtering on aggregated data.
- You need multiple columns or complex operations like `GROUP BY` or `RANK()`.

Use a subquery in the `WHERE` clause when:

- You only need a single value or list to filter your results.
- You're comparing one field against an aggregate or a set of values.

> Subqueries in the `FROM` clause let you "build" a new table within your query — just like views, but on the fly.

Subquery in SELECT Clause

- Just like you can use subqueries in FROM or WHERE, SQL also lets you use subqueries in the SELECT clause. These are useful when you want to add extra info to each row, such as a calculated value that comes from another table
- You can think of them as column-level lookups that return a single value for each row in the result.

Ex - Add each employee's dept name.

Let's say you have an `employees` table with `dept_id`, and a separate `departments` table with `id` and `dept_name`. You want to display each employee's name and department **name**, but without using a `JOIN`.

You can do this using a subquery in the `SELECT` clause:

```sql
SELECT
    name,
    (SELECT dept_name
     FROM departments
     WHERE departments.id = employees.dept_id) AS dept_name
FROM employees;
```

What's happening:

- For each row in `employees`, the subquery:

  ```sql
  SELECT dept_name
  FROM departments
  WHERE departments.id = employees.dept_id
  ```

  finds the corresponding department name.
- The result becomes a column in the output.

✅ When to Use Subqueries in `SELECT`

- When you want to **add a single value** for each row based on related data.
- When a `JOIN` is not needed or would complicate the logic.
- When working with **lookup values** or **aggregates per row**.

📌 Example: Show Each Employee's Salary and the Company Average

```sql
SELECT
    name,
    salary,
    (SELECT AVG(salary) FROM employees) AS company_avg
FROM employees;
```

- This adds the **overall average salary** to every row.
- The subquery is evaluated once and repeated in every result row.

⚠️ Things to Watch Out For

- The subquery **must return exactly one value per row** — either a scalar or a single-column result.
- If it returns multiple rows or columns, SQL will raise an error.
- Overuse can affect performance, especially if the subquery depends on the outer row (correlated subquery).

Conceptual Understanding of Subqueries.

🟩 1. FROM Clause Subquery (Temporary Workspace)

Conceptually:
Imagine you're preparing ingredients on a separate table before cooking your meal. This table is temporary —you use it just for preparation, and then you move ingredients into the main recipe.

In SQL terms, the subquery creates this temporary workspace (called a **derived table**) which you use to:

- Summarize (average, count, sum)
- Rank or reorder data
- Clean or reshape your data before working with it further.

Example Scenario:
Calculate the average salary per department first, then filter only the departments that pay well (average salary above a threshold).

🟨 2. WHERE Clause Subquery (Filter or Condition Check)

Conceptually:
Now, imagine a checklist or a condition you must meet to enter a building. A subquery in the `WHERE` clause is like this checklist—it returns values (or a single value) that you use to check against each row of your main data.

In SQL terms, this subquery is often used to filter:

- Checking if a value exists in another dataset (IN, EXISTS)
- Comparing values against a single value or set of values.

Example Scenario:
Find employees whose salary is above the company average.

🟦 3. SELECT Clause Subquery (Quick Lookup or Calculator)

Conceptually:
Picture asking for someone's name, and immediately referencing your notes for their phone number. This quick lookup is similar to a subquery in the `SELECT` clause—it adds just one extra piece of information per row.

In SQL terms, a `SELECT` subquery is:

- A quick calculation or a reference (lookup) from another table.
- Evaluated separately for each row of your main query.

Example Scenario:
Fetch each employee's department name alongside their information, without explicitly joining the tables.

🚦 Easy Analogies Summary:

Clause	Everyday Analogy	SQL Task
`FROM`	Temporary prep table	Aggregate, rank, reshape data
`WHERE`	Checklist or filter	Condition-based filtering
`SELECT`	Quick lookup/calculation	Add a calculated/related value

Bottom line:

- **FROM subqueries** help you organize and reshape your data first.
- **WHERE subqueries** help you filter your data based on specific conditions.
- **SELECT subqueries** help you quickly add related or computed information per row.

Practical Analogy for Subqueries

Imagine you're running a small coffee shop. You have a notebook to manage your business. Here's how subqueries in each clause work practically, using that notebook:

☕ 1. FROM Subquery (Your Prep Counter)

You decide to bake cookies. Before serving customers, you first mix and bake the dough in your kitchen on a separate table. Once done, you place these freshly baked cookies at the front counter.

In SQL terms:

- The kitchen table where you prepare (mix, bake, cool) your dough is like a subquery in the **FROM** clause.
- The final cookies (results) are passed to your counter (the outer query).

This step helps you **organize or summarize** your data before using it further.

✅ 2. WHERE Subquery (Entrance Checklist)

You have a rule for your cafe: customers can enter the special lounge only if their names appear on a VIP list. Every time someone arrives, you quickly check this VIP list.

In SQL terms:

- The VIP list you refer to is like a subquery in the **WHERE** clause.
- You filter who can enter based on a condition (their name being on your VIP list).

This step helps you **filter or restrict** rows based on a specific condition.

📒 3. SELECT Subquery (Quick Reference)

When each customer orders coffee, you quickly look up their usual preference (cream, sugar, etc.) from your notebook and write it directly on their order slip.

In SQL terms:

- This quick preference check is like a subquery in the **SELECT** clause.
- You add a small piece of additional information directly to each customer's order.

This step helps you **add related or calculated information** quickly per row.

📝 Analogy Summary

SQL Clause	Coffee Shop Analogy	SQL Action
FROM	Kitchen prep table (baking cookies first)	Data preparation/summarize
WHERE	Checking VIP list at the entrance	Filtering based on a list
SELECT	Quick notebook lookup (customer preferences)	Quick lookup per row

This analogy provides a practical way to think about subqueries in SQL:

- `FROM` — **Prepare** data first.
- `WHERE` — **Filter** rows based on a checklist or criteria.
- `SELECT` — Quickly **look up or compute** extra details per row.