Complementary SQL:

Set Operations & Subqueries

A Learning Resource for Basic and Intermediate Levels

Focus Topics:

- `UNION`, `UNION ALL`
- `INTERSECT`, `INTERSECT ALL`
- `EXCEPT`, `EXCEPT ALL`
- Scalar Subqueries
- Correlated Subqueries
- Subqueries in `FROM` Clause
- Subqueries in `SELECT` Clause

Compiled on: May 15, 2025

Contents

1 Set Operations: Combining Result Sets **2**
1.1 What Are Set Operations? (Meanings & Values) . . . 2
1.2 Relations: How Set Operations Play with Others . . . 3
1.2.1 `UNION` and `UNION ALL` . . . 4
1.2.2 `INTERSECT` and `INTERSECT ALL` . . . 6
1.2.3 `EXCEPT` and `EXCEPT ALL` . . . 8

2 Subqueries: Queries Within Queries **11**
2.1 What Are Subqueries? (Meanings & Values) . . . 11
2.2 Relations: How Subqueries Play with Others . . . 12
2.2.1 Scalar Subqueries . . . 13
2.2.2 Correlated Subqueries . . . 14
2.2.3 Subqueries in the `FROM` Clause (Derived Tables) . . . 16
2.2.4 Subqueries in the `SELECT` Clause . . . 18

Chapter 1

Set Operations: Combining Result Sets

Set operations in SQL are your toolkit for combining results from two or more `SELECT` statements into a single, unified result set. Think of them as the SQL way of performing mathematical set theory operations like union, intersection, and difference on your data tables. They're like a data smoothie maker – pour in different ingredients (queries), and get one combined drink (result set)!

1.1 What Are Set Operations? (Meanings & Values)

The Core Idea

Set operations in SQL are keywords used to combine the result sets of two or more `SELECT` statements. To be combined, the `SELECT` statements must have the same number of columns in their result sets, and the data types of corresponding columns must be compatible (e.g., you can combine a column of numbers with another column of numbers, but not directly with a column of dates without casting). The main set operators are:

- `UNION`: Combines results and removes duplicate rows.
- `UNION ALL`: Combines results and includes all rows, even duplicates. It's faster as it skips the duplicate check.
- `INTERSECT`: Returns only rows that are present in both result sets (removes duplicates).
- `INTERSECT ALL` (PostgreSQL 14+): Returns rows present in both result sets, including duplicates up to the minimum number of times they appear in each set.
- `EXCEPT` (or `MINUS` in some SQL dialects like Oracle): Returns rows from the first result set that are not present in the second result set (removes duplicates).
- `EXCEPT ALL` (PostgreSQL 14+): Returns rows from the first result set not in the second, accounting for duplicates.

Value/Output: The output of a set operation is a single result set. The column names for this result set are typically taken from the first `SELECT` statement in the operation. If your data sets could high-five, set operations would be the way they do it!

1.2 Relations: How Set Operations Play with Others

Connections to Prior SQL Concepts

Set operations don't live in a vacuum; they interact with many SQL concepts you've already learned:

- **`SELECT` Statements:** The fundamental building blocks. Each part of a set operation is a complete `SELECT` query, which can include `WHERE` clauses, joins, aggregate functions, etc.
- **Column Compatibility:** The `SELECT` lists of queries involved in a set operation must have the same number of columns, and corresponding columns must have compatible data types. If not, you'll need to use **Casters** (`CAST` or `::`) to make them compatible. For instance, you might cast an `INTEGER` to `VARCHAR` if combining it with a text column conceptually.
- **`WHERE` Clause:** Each individual `SELECT` query can have its own `WHERE` clause to filter rows *before* the set operation is applied. Like picking only red apples from one basket and only green apples from another, before you try to mix them.
- **`ORDER BY` Clause:** An `ORDER BY` clause can typically only be applied once, at the very end of the entire set operation, to sort the final combined result set. You usually can't order individual `SELECT`s within a `UNION` unless they are in subqueries or use `LIMIT`. Column names or aliases from the *first* `SELECT` statement are used for ordering.
- **`LIMIT` and `OFFSET`:** Similar to `ORDER BY`, `LIMIT` and `OFFSET` are typically applied to the final result of the set operation.
- **Aggregators (`GROUP BY`, `HAVING`, `SUM`, `AVG`, etc.):** You can use aggregate functions within the individual `SELECT` statements. For example, you could `UNION` a query that sums sales from 2022 with another query that sums sales from 2023.
- **`DISTINCT`:** `UNION`, `INTERSECT`, and `EXCEPT` implicitly perform a `DISTINCT` operation on their results. `UNION ALL` skips this. You can also use `DISTINCT` within the individual `SELECT` statements.
- **Arithmetic & Math Functions:** These can be used in the `SELECT` list of the queries being combined. E.g., `SELECT salary * 1.1 FROM Employees2023 UNION SELECT salary FROM Employees2022;`
- **Date Functions:** Useful for selecting date ranges or formatting dates consistently across queries before combining.
- **Null Space (`COALESCE`, `IS NULL`):** Can be used within individual queries to handle `NULL`s before combining. For set operations like `UNION`, `INTERSECT`, and `EXCEPT`, two `NULL` values are considered equal for the purpose of duplicate removal or comparison.
- **`CASE` Expressions:** Can be used to create derived columns with consistent meaning or type across different queries before they are combined.
- **Joins (`INNER`, `LEFT`, `RIGHT`, `FULL OUTER`, `CROSS`, `NATURAL`, `SELF`, `USING`):** Joins can be part of the individual `SELECT` statements that provide the result sets to be combined by the set operator.

- **String Functions (`CONCAT`, `SUBSTRING`, etc.):** Can be used in the `SELECT` lists to format or manipulate string data before combining. For example, combining lists of names from different tables, ensuring they have a consistent format.
- **Advanced `WHERE` Conditions (Subqueries with `IN`, `EXISTS`, `ANY`, `ALL`, `IS DISTINCT FROM`):** These can be part of the `WHERE` clause of individual `SELECT` statements involved in a set operation.

1.2.1 `UNION` and `UNION ALL`

`UNION` / `UNION ALL`: Combining Results

`UNION` combines the result sets of two or more `SELECT` statements and removes duplicate rows. `UNION ALL` also combines results but includes all rows, even duplicates. `UNION ALL` is generally faster because it skips the duplicate removal step.

Basic Syntax:

```
SELECT columnA, columnB FROM TableOne
WHERE conditionOne
UNION / UNION ALL
SELECT columnX, columnY FROM TableTwo
WHERE conditionTwo
ORDER BY columnA; -- Optional, orders the final result
```

Key Rules:

- The number of columns in all `SELECT` statements must be the same.
- The data types of corresponding columns must be compatible.
- Column names in the final result are taken from the first `SELECT` statement.

Think of `UNION` as making a fruit salad where each unique fruit appears once. `UNION ALL` is like just dumping all fruit bowls into one big bowl – duplicates and all!

What Are They? (Meanings & Values)

`UNION` & `UNION ALL`: Meaning & Value

`UNION`:

- **Meaning:** Merges the results of two or more queries into a single result set. It implicitly performs a `DISTINCT` operation, meaning any rows that are identical across the combined set will appear only once.
- **Value:** Useful for creating a consolidated, unique list from different sources. For example, getting a unique list of all customers from a ‘CurrentCustomers‘ table and an ‘ArchivedCustomers‘ table.

`UNION ALL`:

- **Meaning:** Merges the results of two or more queries, keeping all rows from each query, including any duplicates.
- **Value:** Useful when you need every record from all combined queries, and duplicates are meaningful or performance is critical (as it avoids the sort/hash for

deduplication). For example, combining transaction logs from different periods where each transaction event is important.

How to Use Them: Structures & Syntax

Example: Combining Active and Inactive Employees

Suppose we have 'ActiveEmployees' and 'FormerEmployees' tables, both with 'employeeId' and 'fullName'.

```sql
-- Get a unique list of all employee names
SELECT employeeId, fullName FROM ActiveEmployees
UNION
SELECT employeeId, fullName FROM FormerEmployees
ORDER BY fullName;

-- Get all records, including duplicates if an employee
-- somehow exists in both (e.g., re-hired with same ID)
-- or if we just want every entry.
SELECT employeeId, status FROM ActiveEmployees
UNION ALL
SELECT employeeId, 'Former' AS status FROM FormerEmployees
ORDER BY employeeId;
```

You can chain multiple `UNION` or `UNION ALL` operations:

```sql
SELECT col1 FROM TableA
UNION ALL
SELECT col1 FROM TableB
UNION ALL
SELECT col1 FROM TableC;
```

Why Use Them? (Advantages)

`UNION` / `UNION ALL`: Advantages

- **Consolidation:** Combine data from disparate but structurally similar tables or queries into a single view. Like merging guest lists from different parties.
- **Simplified Queries:** Can break down complex data retrieval into simpler, individual queries that are then combined.
- **`UNION` for Uniqueness:** Automatically handles deduplication, saving you from writing complex logic to find distinct rows.
- **`UNION ALL` for Performance:** Significantly faster than `UNION` when duplicate removal is not needed, as it avoids the costly sort or hash operation. It's the express lane of data merging!

Watch Out! (Disadvantages)

UNION / UNION ALL: Disadvantages

⚠

- **Column Compatibility Strictness:** The number, order, and data types of columns in all `SELECT` statements must match or be compatible. An accidental mismatch can lead to errors or, worse, nonsensical results if implicit casting occurs unexpectedly. It's like trying to stack square pegs and round holes if you're not careful.
- **`UNION` Performance Overhead:** The deduplication process in `UNION` can be slow on very large datasets. If you know there are no duplicates or duplicates are acceptable, `UNION ALL` is your friend.
- **Generic Column Names:** The final result set takes column names from the first `SELECT`. If subsequent queries have more descriptive aliases, they are ignored for the final output column names. You might need to alias in the first query.
- **`ORDER BY` and `LIMIT` Scope:** These clauses generally apply to the entire combined result set, not individual parts, unless those parts are themselves subqueries with their own `ORDER BY` and `LIMIT` (and even then, the outer `ORDER BY` would prevail for the final order).

1.2.2 INTERSECT and INTERSECT ALL

INTERSECT / INTERSECT ALL: Finding Common Ground

`INTERSECT` returns only the rows that appear in *both* result sets of the `SELECT` statements, removing duplicates. `INTERSECT ALL` (PostgreSQL 14+) also returns common rows but considers duplicate counts.

Basic Syntax:

```
SELECT columnA, columnB FROM TableOne
INTERSECT / INTERSECT ALL
SELECT columnX, columnY FROM TableTwo
ORDER BY columnA; -- Optional
```

It's like finding friends who are on both your camping trip list and your cousin's.

What Are They? (Meanings & Values)

INTERSECT & INTERSECT ALL: Meaning & Value

`INTERSECT`:

- **Meaning:** Computes the set intersection of the rows returned by the involved queries. Only rows that are present in *all* query results are returned. Duplicates in the final result are eliminated.
- **Value:** Useful for finding common records between two datasets. For example, finding customers who made a purchase last year AND this year.

`INTERSECT ALL` (PostgreSQL 14+):

- **Meaning:** Similar to `INTERSECT`, but it respects multiplicity. If a row appears ‘m‘ times in the first result set and ‘n‘ times in the second, it will appear ‘min(m, n)‘ times in the final result.
- **Value:** Useful when the count of common occurrences matters. For example, if you’re analyzing common items in two different shopping carts and want to know how many of each common item there are, up to the minimum count in either cart.

Some database systems like SQL Server and Oracle have long supported `INTERSECT ALL`. PostgreSQL added it in version 14.

How to Use Them: Structures & Syntax

Example: Employees in Multiple Projects

Suppose you want to find employees who worked on ’ProjectAlpha’ AND ’ProjectBeta’.

```sql
SELECT employeeId FROM EmployeeProjects WHERE projectId = 'ProjectAlpha'
INTERSECT
SELECT employeeId FROM EmployeeProjects WHERE projectId = 'ProjectBeta';

-- With INTERSECT ALL, if an employee was listed twice for Alpha
-- and thrice for Beta (hypothetically), they'd appear twice.
-- (Assuming EmployeeProjects could have such duplicates for this example)
SELECT employeeId FROM ProjectAlphaMembers
INTERSECT ALL
SELECT employeeId FROM ProjectBetaMembers;
```

Why Use Them? (Advantages)

`INTERSECT` / `INTERSECT ALL`: Advantages

- **Clarity for Commonality:** Clearly expresses the intent to find common elements between sets, which can be more readable than complex joins or subqueries for the same purpose.
- **Automatic Deduplication (`INTERSECT`):** Simplifies finding unique common rows.
- **Handles `NULL`s Predictably:** For comparison purposes in `INTERSECT`, two `NULL` values are considered equal, unlike standard `WHERE` clause comparisons where `NULL = NULL` is false.
- **Duplicate Awareness (`INTERSECT ALL`):** Provides a way to find common rows while respecting their frequency, if needed.

Watch Out! (Disadvantages)

INTERSECT / INTERSECT ALL: Disadvantages

⚠

- **Performance:** Like `UNION`, `INTERSECT` involves processing both datasets and then finding commonalities, which can be resource-intensive, often involving sorting or hashing both inputs.
- **Column Compatibility:** Same strict rules as `UNION` regarding column count and data types.
- **Limited to Common Columns:** You can only find intersections based on the columns selected in the queries. If you need other columns from the original tables for the final output, you might need to join the result of the `INTERSECT` operation back to the original tables.
- **`INTERSECT ALL` Availability:** Not all SQL databases support `INTERSECT ALL`, or it might be a newer feature (like in PostgreSQL). Always check your DBMS version.

1.2.3 EXCEPT and EXCEPT ALL

EXCEPT / EXCEPT ALL: Finding Differences

`EXCEPT` (known as `MINUS` in Oracle) returns distinct rows from the first `SELECT` statement that are *not* found in the second `SELECT` statement. `EXCEPT ALL` (PostgreSQL 14+) does this while considering duplicate counts.

Basic Syntax:

```
SELECT columnA, columnB FROM TableOne
EXCEPT / EXCEPT ALL
SELECT columnX, columnY FROM TableTwo
ORDER BY columnA; -- Optional
```

It's like making a list of people invited to your party but *excluding* those who said they can't come.

What Are They? (Meanings & Values)

EXCEPT & EXCEPT ALL: Meaning & Value

`EXCEPT`:

- **Meaning:** Computes the set difference. It returns all distinct rows that are in the result of the first query but not in the result of the second query.
- **Value:** Useful for finding records unique to one dataset compared to another. For example, finding products available in Warehouse A but not in Warehouse B.

`EXCEPT ALL` (PostgreSQL 14+):

- **Meaning:** Similar to `EXCEPT`, but it respects multiplicity. If a row appears 'm' times in the first set and 'n' times in the second, it will appear 'max(0, m - n)' times in the result.

- **Value:** Useful when the count of differences matters. E.g., if you have 5 apples and give away 2 (those 2 are in the second set), `EXCEPT ALL` tells you you have 3 apples left.

Oracle SQL uses `MINUS` instead of `EXCEPT`, and `MINUS ALL` is not standard or widely available. PostgreSQL added `EXCEPT ALL` in version 14.

How to Use Them: Structures & Syntax

Example: Products Not Sold This Month

Suppose you want to find products in your 'Products' table that have no sales records in 'MonthlySales'.

```sql
SELECT productId FROM Products
EXCEPT
SELECT productId FROM MonthlySales;

-- With EXCEPT ALL, if Products listed productId 101 three times
-- (e.g. different batches but same ID for this example)
-- and MonthlySales listed it once, the result would show productId 101 twice.
SELECT productId FROM AllProductStock
EXCEPT ALL
SELECT productId FROM ProductsSold;
```

Why Use Them? (Advantages)

`EXCEPT` / `EXCEPT ALL`: Advantages

- **Clear for Differences:** Provides a very readable way to find rows present in one set but not another, often clearer than 'NOT IN' or 'NOT EXISTS' subqueries, especially for complex set definitions.
- **Automatic Deduplication (`EXCEPT`):** Simplifies finding unique differing rows.
- **Handles `NULL`s Predictably:** Like `INTERSECT`, two `NULL`s are considered equal for comparison. This avoids the common pitfalls of `NOT IN` with `NULL`s.
- **Duplicate Awareness (`EXCEPT ALL`):** Allows for set difference while respecting row counts.

Watch Out! (Disadvantages)

`EXCEPT` / `EXCEPT ALL`: Disadvantages

- **Performance:** Can be resource-intensive, similar to `UNION` and `INTERSECT`, as it needs to process both datasets.
- **Column Compatibility:** The same strict rules for column count and data types

apply.

- **Order Matters:** Unlike `UNION` and `INTERSECT` where ‘A UNION B‘ is the same as ‘B UNION A‘ (for distinct rows), ‘A EXCEPT B‘ is different from ‘B EXCEPT A‘. The order defines which set is the minuend and which is the subtrahend. "Order matters, or your answer scatters!"
- **`EXCEPT ALL` Availability:** Not all SQL databases support `EXCEPT ALL`. Check your DBMS. The keyword `MINUS` is used in Oracle instead of `EXCEPT`.

Chapter 2

Subqueries: Queries Within Queries

Subqueries, often called inner queries or nested queries, are queries embedded within another SQL query (the outer query). They are powerful tools for performing complex data retrieval and manipulation by breaking down problems into smaller, manageable parts. A subquery can return a single value (scalar), a single column of multiple rows, or multiple columns of multiple rows (a table). It's like asking a question to get an answer that helps you ask a better, more specific main question!

2.1 What Are Subqueries? (Meanings & Values)

The Core Idea: Queries Hiding in Queries

A subquery, also known as an inner query or nested query, is a `SELECT` statement embedded inside another SQL statement (the outer query). It's like a question within a question! The subquery executes first, and its result is then used by the outer query.
Types of Values/Outputs from Subqueries:

- **Scalar Subquery:** Returns a single value (one row, one column). Imagine asking, "What's the highest salary?" – the answer is one number.
- **Column Subquery:** Returns a single column of multiple rows. Like asking, "List all employee IDs in the Sales department."
- **Row Subquery:** Returns a single row with multiple columns (less common as a direct output type to be used by specific operators, but a subquery in `FROM` returning one row is an example).
- **Table Subquery:** Returns multiple rows and multiple columns (a virtual table). This is typical for subqueries in the `FROM` clause.

Subqueries can appear in various parts of an SQL statement:

- In the `SELECT` list.
- In the `FROM` clause (as a derived table).
- In the `WHERE` clause (very common, often with `IN`, `EXISTS`, `ANY`, `ALL`, or comparison operators).
- In the `HAVING` clause.
- Within `UPDATE`, `DELETE`, `INSERT` statements.

They help break down complex problems. Instead of one giant leap, you take smaller, calculated steps.

2.2 Relations: How Subqueries Play with Others

Subqueries and Their SQL Friends

Subqueries are versatile and interact with almost every part of SQL that you've learned:

- **Outer Query:** The subquery's result is consumed by its parent, the outer query. The relationship can be simple (subquery runs once) or correlated (subquery runs for each row of the outer query).
- **`WHERE` Clause Operators:**
  - **Comparison Operators (`=`, `<`, `>`, `<=`, `>=`, `<>`):** Used with scalar subqueries. E.g., 'WHERE salary = (SELECT MAX(salary) FROM Employees)'.
  - **`IN` / `NOT IN`:** Used with column subqueries. E.g., 'WHERE departmentId IN (SELECT departmentId FROM Departments WHERE location = 'NewYork')'. This is an "Advanced WHERE Condition" concept you've seen.
  - **`EXISTS` / `NOT EXISTS`:** Checks if the subquery returns any rows. E.g., 'WHERE EXISTS (SELECT 1 FROM Orders WHERE Orders.customerId = Customers.customerId)'. Also an "Advanced WHERE Condition".
  - **`ANY` / `SOME`, `ALL`:** Used with comparison operators and column subqueries. E.g., 'WHERE salary > ANY (SELECT salary FROM Interns)'. These are also "Advanced WHERE Conditions".
  - **`IS DISTINCT FROM` / `IS NOT DISTINCT FROM`:** Can be used if comparing the result of a scalar subquery, especially if NULLs are involved.
- **`SELECT` Clause:** A scalar subquery can compute a value for each row of the outer query.
- **`FROM` Clause:** A subquery here acts as a temporary table (derived table) that the outer query can select from, join with, etc. Its columns must have names (use aliases if they are expressions).
- **Aggregators (`GROUP BY`, `HAVING`, `SUM`, `AVG`, etc.):** Subqueries often use aggregate functions to produce summary values (e.g., average salary, total count) that the outer query then uses for comparison or display. The outer query can also aggregate results from a derived table subquery.
- **Joins (`INNER`, `LEFT`, etc.):** Subqueries in the `FROM` clause can be joined with other tables or other derived tables. Correlated subqueries often implicitly "join" data with the outer query through their correlation condition.
- **Casters (`CAST`, `::`):** May be needed within a subquery or when comparing its result to ensure data type compatibility.
- **Null Space (`COALESCE`, `IS NULL`):** Can be crucial within subqueries, especially those used in `WHERE` conditions, to handle `NULL`s appropriately.
- **`CASE` Expressions:** Can be used inside subqueries to derive columns or values.

- **String Functions, Date Functions, Math Functions:** All these can be used within the subquery's `SELECT` list or `WHERE` clause to process data before it's passed to the outer query.
- **Set Operations (`UNION`, `INTERSECT`, `EXCEPT`):** A subquery can itself be composed of set operations. For instance, a subquery in the `FROM` clause could be '(SELECT id FROM TableA UNION SELECT id FROM TableB) AS CombinedIds'.

Subqueries are like SQL's secret agents – they go deep, get information, and report back!

2.2.1 Scalar Subqueries

Scalar Subquery: A Single Value Answer

A scalar subquery is a subquery that returns exactly one column and at most one row (i.e., a single value). If it returns more than one row, or no rows when one is expected (e.g., in an assignment), it can cause an error.
Common Usage: In `WHERE` clauses for comparison, in `SELECT` lists, or anywhere a single value expression is expected.

```
-- In WHERE clause
SELECT productName, price
FROM Products
WHERE price > (SELECT AVG(price) FROM Products);

-- In SELECT list (often correlated, see later)
SELECT employeeName, salary,
       (SELECT AVG(salary) FROM Employees) AS companyAverageSalary
FROM Employees;
```

If a scalar subquery might return no rows, `COALESCE` can provide a default: 'COALESCE((SELECT MAX(score) FROM Attempts), 0)'.

What Is It? (Meaning & Value)

Scalar Subquery: Meaning & Value

- **Meaning:** It's a query designed to produce a single data value. This value is then used by the outer query, typically for comparison or as a calculated field.
- **Value:** Allows dynamic computation of comparison values or individual data points. For example, finding all employees earning above the company average salary, where the average is calculated on-the-fly. It keeps your numbers fresh, not stale like last week's bread!

How to Use It: Structures & Syntax

Example: Product Priced Above Average

```
SELECT productName, unitPrice
FROM Products
WHERE unitPrice = (
    SELECT MAX(unitPrice)
    FROM Products
```

```
    WHERE category = 'Electronics'
); -- Finds the most expensive electronic product.
```

The subquery '(SELECT MAX(unitPrice) FROM Products WHERE category = 'Electronics')' must return one value.

Why Use It? (Advantages)

Scalar Subquery: Advantages

- **Dynamic Comparisons:** Values used in conditions don't need to be hardcoded. They can be derived from the current state of the data.
- **Readability (sometimes):** Can make the outer query's logic clearer by encapsulating the calculation of a specific value.
- **Flexibility:** Can be used in `SELECT`, `WHERE`, `HAVING`, and even `ORDER BY` (though less common there).

Watch Out! (Disadvantages)

Scalar Subquery: Disadvantages

- **Single Row, Single Column Strictness:** If the subquery returns more than one row, you'll get an error. "One value it must ferry, or your query will be sorry!"
- **Performance (if correlated and complex):** If a scalar subquery in the `SELECT` list is correlated and complex, it might be executed for every row of the outer query, potentially impacting performance.
- **No Rows Returned:** If a scalar subquery returns no rows, its result is `NULL`. This can be surprising in comparisons (e.g., 'salary > NULL' is unknown). Use `COALESCE` if a default is needed.

2.2.2 Correlated Subqueries

Correlated Subquery: The Dependent Inner Query

A correlated subquery is an inner query that depends on the outer query for its values. It uses columns from the outer query's tables in its own `WHERE` clause (or other parts). This means the subquery is re-evaluated for each row processed by the outer query.
Common Usage: In `WHERE` (often with `EXISTS`), `SELECT` list.

```
-- Find employees who have made at least one sale
SELECT E.firstName, E.lastName
FROM Employees E
WHERE EXISTS (
    SELECT 1
    FROM Sales S
    WHERE S.employeeId = E.employeeId -- Correlation!
);
```

```sql

-- Show each department and the number of employees in it
SELECT D.departmentName,
       (SELECT COUNT(*)
        FROM Employees E
        WHERE E.departmentId = D.departmentId) AS numberOfEmployees
FROM Departments D;
```

It's like the inner query constantly asking the outer query, "For your current row, what's the value of X?"

What Is It? (Meaning & Value)

Correlated Subquery: Meaning & Value

- **Meaning:** A subquery that references columns from the table(s) in the outer query. It cannot be run independently of the outer query because it needs data from each outer row to complete its own execution.
- **Value:** Enables row-by-row processing logic. Powerful for checks like "does a matching record exist in another table for this specific row?" or "calculate a specific aggregate for this row's related group?". For instance, finding all departments that have at least one employee, or calculating the average order value for each customer.

How to Use It: Structures & Syntax

Example: Employees Earning More Than Their Department Average

```sql
SELECT E1.firstName, E1.lastName, E1.salary, D.departmentName
FROM Employees E1
JOIN Departments D ON E1.departmentId = D.departmentId
WHERE E1.salary > (
    SELECT AVG(E2.salary)
    FROM Employees E2
    WHERE E2.departmentId = E1.departmentId -- Correlation on departmentId
);
```

Here, the subquery calculates the average salary for the *current* employee's (‘E1‘) department.

Why Use It? (Advantages)

Correlated Subquery: Advantages

- **Row-Specific Logic:** Perfect for conditions or calculations that depend on the values of the current row being processed by the outer query.
- **Expressiveness:** Can sometimes express complex conditions more naturally than intricate joins, especially with `EXISTS`.
- **`EXISTS` Efficiency:** Correlated subqueries with `EXISTS` can be very efficient as the database can stop evaluating the subquery as soon as the first matching row is found. It doesn't need to find all matches.

Watch Out! (Disadvantages)

Correlated Subquery: Disadvantages

- **Performance Pitfalls:** Since a correlated subquery is (conceptually) executed for each row of the outer query, it can be slow if not optimized well by the database or if the subquery itself is complex. "For each outer row, a new scurry; use with care, or performance worries!"
- **Harder to Read (sometimes):** The nested logic and dependency on the outer query can sometimes make them harder to understand than an equivalent join.
- **Optimization Challenges:** Database optimizers might have a harder time finding the most efficient execution plan compared to some join-based alternatives. However, modern optimizers are quite good.

2.2.3 Subqueries in the `FROM` Clause (Derived Tables)

Subquery in `FROM`: The Virtual Table

A subquery in the `FROM` clause creates a temporary, virtual table (often called a derived table or inline view). The outer query then operates on this derived table as if it were a regular table. The derived table *must* be given an alias. Any calculated columns within the derived table should also be aliased.

Syntax:

```
SELECT dt.columnA, dt.someValue
FROM (
    SELECT T.columnA, COUNT(*) AS someValue
    FROM SomeTable T
    GROUP BY T.columnA
) AS dt -- Alias for the derived table is mandatory!
WHERE dt.someValue > 10;
```

It's like building a custom Lego model (the subquery) and then using that model as a piece in a larger Lego creation (the outer query).

What Is It? (Meaning & Value)

Subquery in FROM: Meaning & Value

- **Meaning:** The result set of the subquery is treated as a table by the outer query. This allows you to pre-process data (e.g., aggregate, filter, pivot) before the main query logic is applied.
- **Value:** Fantastic for breaking complex queries into logical steps. You can perform aggregations or window functions in a subquery and then join those results to other tables, or filter on those aggregated values in the outer query. It's a cornerstone of modular query design before Common Table Expressions (CTEs) became widely used (and CTEs are often syntactic sugar over derived tables).

How to Use It: Structures & Syntax

Example: Departments with Average Salary

List departments along with their average employee salary, derived from a subquery.

```sql
SELECT D.departmentName, DeptSalaries.avgSalary
FROM Departments D
JOIN (
    SELECT departmentId, AVG(salary) AS avgSalary
    FROM Employees
    GROUP BY departmentId
) AS DeptSalaries ON D.departmentId = DeptSalaries.departmentId
WHERE DeptSalaries.avgSalary > 60000;
```

The subquery calculates average salaries per department, and the outer query joins this to get department names.

Why Use It? (Advantages)

Subquery in FROM: Advantages

- **Modularity & Readability:** Breaks down complex logic into more understandable intermediate steps.
- **Pre-Aggregation/Transformation:** Allows aggregation or calculation of values that can then be used in joins or further filtering by the outer query. Essential when you need to filter on an aggregate result without repeating the aggregation logic.
- **Working with Summarized Data:** Enables joining summarized data (e.g., average salary per department) back to detail tables.
- **Simplifies Complex Joins:** Can sometimes simplify what would otherwise be very complex join conditions or self-joins.

Watch Out! (Disadvantages)

Subquery in FROM: Disadvantages

⚠

- **Alias Requirement:** Forgetting to alias the derived table or its calculated columns is a common syntax error. "Give your derived kid a name, or SQL will play a blaming game!"
- **Nesting Complexity:** Deeply nested derived tables can become hard to read and debug, much like Russian dolls of queries. (CTEs often improve readability here).
- **Potential for Inefficiency (older DBMS):** Historically, some older database optimizers struggled with complex derived tables, though modern ones are much better. The derived table might be materialized (written to a temporary space), which could add overhead.

2.2.4 Subqueries in the SELECT Clause

Subquery in SELECT: A Column from a Query

A subquery in the `SELECT` clause is used to retrieve a single value that will become a column in the result set of the outer query. This type of subquery must be a scalar subquery (return one row, one column). It is often, but not always, correlated.
Syntax:

```
SELECT
    E.employeeName,
    E.salary,
    (SELECT D.departmentName
     FROM Departments D
     WHERE D.departmentId = E.departmentId) AS departmentName, -- Correlated scalar
    (SELECT AVG(S.salary) FROM Employees S) AS companyAvgSalary -- Non-correlated scalar
FROM Employees E;
```

Each selected subquery adds another column to your results, like picking an extra topping for your data pizza.

What Is It? (Meaning & Value)

Subquery in SELECT: Meaning & Value

- **Meaning:** For each row produced by the outer query, the subquery in the `SELECT` list is executed, and its single-value result is included as a column in that row.
- **Value:** Useful for fetching related information or calculating specific metrics for each row of the main query without complex joins, especially when the related information is a single aggregate or lookup. For example, showing each employee and the name of their department, or each product and the date of its last sale.

How to Use It: Structures & Syntax

Example: Employee with Their Department's Project Count

```
SELECT
    E.firstName,
    E.lastName,
    (SELECT COUNT(P.projectId)
     FROM Projects P
     WHERE P.departmentId = E.departmentId) AS departmentProjectCount
FROM Employees E;
```

For each employee, the subquery counts projects associated with that employee's department.

Why Use It? (Advantages)

Subquery in SELECT: Advantages

- **Row-Specific Detail:** Can retrieve specific, related details or calculations for each row of the outer query in a separate column.
- **Alternative to Joins (sometimes):** Can sometimes replace a LEFT JOIN when you only need one specific piece of information from the related table and want to avoid potential row duplication that joins might cause if the relationship is one-to-many.
- **Readability for Simple Lookups:** For straightforward lookups (like getting a department name from an ID), it can be quite readable.

Watch Out! (Disadvantages)

Subquery in SELECT: Disadvantages

- **Performance Cost:** If the subquery is correlated and executed for every row of a large outer result set, it can lead to significant performance degradation. This is a classic N+1 query problem if not optimized well. "A select subquery, row by row it might flurry; if many rows, performance will worry!"
- **Scalar Requirement:** Must return a single value (one row, one column). Returning multiple rows will cause an error.
- **Limited to One Value:** You can only retrieve one column's value per subquery. If you need multiple columns from a related entity, a JOIN is usually better.
- **Readability with Complexity:** If the subquery logic itself is complex, having it in the SELECT list can make the overall query harder to read.