SQL essentials

Mastering database queries: a comprehensive guide to SQL

Learning session 04
Subqueries and joins

Instructor
Péter Fülöp
peteristvan.fulop@hcl.com

What exactly is a subquery?

- A **subquery** is a **query nested inside another query**.
- The **data returned by a subquery** (inner query) will be used in the main(outer) query (e.g as a condition to further restrict the data that is retrieved)
- Subqueries **can be used** in various parts of a SQL statement, including the SELECT, FROM, and WHERE clauses.
- While there are many situations where **subqueries can be rewritten using joins** (and might even be more efficient when done so), there are specific scenarios where subqueries are more appropriate or even necessary.

Using subqeries in the WHERE clause

Retrieve all employees whose country of residence matches that of the employee with ID 10.

SQL

```sql
SELECT id, first_name, last_name, residence_country_id
FROM employees
WHERE residence_country_id = (SELECT residence_country_id
                              FROM employees
                              WHERE id = 10)
      AND id <> 10;
```

OUTPUT

	id	first_name	last_name	residence_country_id
1	14	Katalin	Szabo	USA
2	17	Tamas	Molnar	USA
3	20	Noemi	Biro	USA

7 rows

Using subqeries in the WHERE clause

Retrieve the ID, last_name, first_name, and salary of employees whose salary matches one of the three highest salaries among junior employees. Arrange the results alphabetically.

First, let's identify the **top three salaries** of employees in the Junior band.

SQL

```sql
SELECT DISTINCT salary
FROM employees
WHERE band_id IN (SELECT id from bands WHERE lower(name) LIKE 'junior%')
ORDER BY salary DESC
LIMIT 3;
```

OUTPUT

	salary
1	4000
2	3800
3	3500

3 rows

Replacing the subquery with join

Subqueries and joins are both techniques used in SQL to combine data from multiple tables or, in some cases, from the same table.

In many cases, joins are more efficient than subqueries because **database systems are optimized for join operations**. However, this isn't a hard and fast rule; the performance can vary based on the specific query, database design, and the database system in use.

SQL

```sql
SELECT DISTINCT e.salary
FROM employees e
INNER JOIN bands b ON (e.band_id = b.id) AND (lower(b.name) LIKE 'junior%')
ORDER BY salary DESC
LIMIT 3
```

Writing the outer query with the subquery in the WHERE clause

Having constructed the subquery to retrieve the top three salaries for Junior positions, we can now move forward with **building the main/outer query**.

SQL

```
SELECT id, first_name, last_name, salary
FROM employees
WHERE salary IN (SELECT DISTINCT e.salary
                 FROM employees e
                          INNER JOIN bands b ON (e.band_id = b.id)
                     AND (lower(b.name) LIKE 'junior%')
                 ORDER BY salary DESC
                 LIMIT 3)
ORDER BY last_name, first_name;
```

Is it possible to construct a different query than the one above without incorporating a subquery in the WHERE clause?

What is a derived table

- **A derived table** is a temporary result set that can be used within a SQL statement, such as a SELECT, INSERT, UPDATE, or DELETE statement.
- Derived tables **are often used in conjunction with subqueries in the FROM clause** of a SQL statement.
- **A derived table is essentially a subquery** that produces a table result. It is not stored as a persistent object in the database, but rather, it exists just for the duration of the query execution.
- In essence, derived tables allow you to create intermediate steps or transformations on your data, which can be especially useful for complex queries.

Example of using derived tables

Having constructed the subquery to retrieve the top three salaries for Junior positions, we can now move forward with **building the main/outer query** using derived tables this time.

SQL

```
SELECT e.id, e.first_name, e.last_name, e.salary
FROM employees e,
     (SELECT DISTINCT e.salary
      FROM employees e
               INNER JOIN bands b ON (e.band_id = b.id)
          AND (lower(b.name) LIKE 'junior%')
      ORDER BY salary DESC
      LIMIT 3) top_salaries
WHERE e.salary = top_salaries.salary
ORDER BY e.last_name, e.first_name;
```

Example of using derived tables

Having constructed the subquery to retrieve the top three salaries for Junior positions, we can now move forward with **building the main/outer query**.

SQL

```
SELECT e.id, e.first_name, e.last_name, e.salary
FROM employees e
         INNER JOIN (SELECT DISTINCT e.salary
                     FROM employees e
                              INNER JOIN bands b ON (e.band_id = b.id)
                         AND (lower(b.name) LIKE 'junior%')
                     ORDER BY salary DESC
                     LIMIT 3) top_salaries
         ON e.salary = top_salaries.salary
ORDER BY last_name, first_name;
```

Pros and cons of using derived table

PROS

Modularity: Derived tables can break down complex queries into more manageable and logical parts. This can make the SQL more readable and easier to understand.

Encapsulation: Derived tables can encapsulate specific transformations or calculations, ensuring they are done before further processing in the main query.

Reuse: In some cases, using a derived table can prevent the need to repeat the same subquery logic multiple times within a query.

Compatibility: Derived tables are supported in almost all relational database systems, making them a portable solution if you ever need to migrate your SQL code.

CONS

Performance: Not all query optimizers handle derived tables efficiently. In some cases, joins or common table expressions (CTEs) with the WITH clause might be more performant.

Readability: While derived tables can enhance readability by breaking down complex queries, excessive nesting of derived tables can make a SQL statement harder to read and understand.

Debugging: If there's an issue with the logic inside a derived table, it might be more challenging to debug than a straightforward join or a CTE, especially if multiple levels of derived tables are used.

Lack of Persistence: Derived tables are temporary and only exist for the duration of the query execution. If you find yourself using the same derived table logic frequently, it might be more efficient to create a view or a temporary table.

Using the WITH clause

CTEs are a part of the SQL standard, so their behavior is more consistent across different database systems compared to some proprietary implementations of derived tables.

SQL

```sql
WITH top_salaries AS (
    SELECT DISTINCT e.salary
    FROM employees e
    INNER JOIN bands b ON e.band_id = b.id
    WHERE lower(b.name) LIKE 'junior%'
    ORDER BY e.salary DESC
    LIMIT 3
)

SELECT e.id, e.first_name, e.last_name, e.salary
FROM employees e
JOIN top_salaries ON e.salary = top_salaries.salary
ORDER BY e.last_name, e.first_name;
```

Correlated vs Non-correlated subqueries

NON-CORRELATED SUBQUERY

In a regular (non-correlated) subquery the outer query relies on the values returnd by the inner query. **The subquery (inner query) does not rely upon the outer query**, the subquery is noncorrelated with the outer query.

Execution / Evaluation: The inner query is executed first and PostgreSQL works its way from the innermost query to the outermost query.

CORRELATED SUBQUERY

A subquery that uses values from the outer/main query. In a correlated subquery **the inner query relies on the values from the outer query**.

Execution / Evaluation: A correlated subquery is evalutated for each row processed by the main/outer query.

Example of using correlated subqueries

Retrieve the id, first_name, and last_name of employees who are currently assigned to one or more active projects.

Let's proceed incrementally. To start, let's display the project assignments for active projects for the employee with an ID of 1.

SQL

```sql
SELECT *
FROM projects p
         INNER JOIN project_assignments pa ON pa.project_id = p.id
WHERE COALESCE(p.start_date, current_date) <= current_date
  AND p.cancellation_date IS NULL
  AND p.completion_date IS NULL
  AND pa.employee_id = 1;
```

OUTPUT

..	name	employee_id	p.start_date
2	BIOasis: Business Intelligence Optimization & Analytics	1	2023-03-13
7	PivotPulse: Dynamic BI Query Engine	1	2021-01-15

2 rows

Example of using correlated subqueries

Having constructed the query to identify active project assignments for a specified employee, we can now move forward. We'll **craft the primary query** and incorporate the one we previously developed as a correlated subquery.

SQL

```sql
SELECT id, first_name, last_name
FROM employees emp
WHERE  (SELECT COUNT(*)
        FROM projects p
                 INNER JOIN project_assignments pa ON pa.project_id = p.id
        WHERE COALESCE(p.start_date, current_date) <= current_date
          AND p.cancellation_date IS NULL
          AND p.completion_date IS NULL
          AND pa.employee_id = emp.id) >0;
```

OUTPUT

	id	first_name	last_name
1	11	Anna	Kovacs
2	12	Balazs	Toth
3	1	Gabor	Nagy

10 rows

Using EXISTS to determine if a subquery returns any results

The **EXISTS** keyword in SQL is used to determine if a subquery returns any results. It's a boolean operator that returns true if the subquery returns one or more rows, and false if the subquery returns no rows.

SQL

```sql
SELECT id, first_name, last_name
FROM employees emp
WHERE EXISTS (SELECT p.id
        FROM projects p
                 INNER JOIN project_assignments pa ON pa.project_id = p.id
        WHERE COALESCE(p.start_date, current_date) <= current_date
          AND p.cancellation_date IS NULL
          AND p.completion_date IS NULL
          AND pa.employee_id = emp.id);
```

OUTPUT

	id	first_name	last_name
1	11	Anna	Kovacs
2	12	Balazs	Toth
3	1	Gabor	Nagy

10 rows

Employing correlated subqueries within the SELECT clause

Fetch the `id`, `first_name`, and `last_name`, along with the count of active projects, for employees who are assigned to one or more ongoing projects.

SQL

```sql
SELECT *
FROM (SELECT id,
             first_name,
             last_name,
             (SELECT COUNT(*)
              FROM projects p
                       INNER JOIN project_assignments pa ON pa.project_id = p.id
              WHERE COALESCE(p.start_date, current_date) <= current_date
                AND p.cancellation_date IS NULL
                AND p.completion_date IS NULL
                AND pa.employee_id = emp.id) as nr_active_projects
      FROM employees emp) AS tmp
WHERE tmp.nr_active_projects > 0;
```

Once we delve into the GROUP BY clause and the aggregate functions, we can craft a more efficient solution.

Example of using multi-column subqueries

Fetch a list of employees for whom there is another employee with the same team lead and band, both of which must not be null.

SQL

```
SELECT emp.id, emp.first_name, emp.last_name,
       tld.id, tld.first_name, tld.last_name,
       b.name as band
FROM employees emp
         INNER JOIN employees tld on emp.lead_employee_id = tld.id
         INNER JOIN bands b on emp.band_id = b.id
WHERE
  (emp.band_id, emp.lead_employee_id) IN (SELECT e.band_id, e.lead_employee_id
                                          FROM employees e
                                          WHERE e.band_id = emp.band_id
                                          AND e.lead_employee_id = emp.lead_employee_id
                                          AND e.id <> emp.id);
```

Rewriting multi-column subqueries to joins

Fetch a list of employees for whom there is another employee with the same team lead and band, both of which must not be null.

SQL

```sql
SELECT DISTINCT emp.id, emp.first_name,
       emp.last_name, tld.id, tld.first_name,
       tld.last_name, b.name as band
FROM employees emp
         INNER JOIN employees tld on emp.lead_employee_id = tld.id
         INNER JOIN bands b on emp.band_id = b.id
         INNER JOIN employees another_employee ON
            emp.id <> another_employee.id
            AND another_employee.band_id = emp.band_id
            AND another_employee.lead_employee_id = emp.lead_employee_id;
-- Could you explain the reason for utilizing DISTINCT in our query?
```