Subqueries & query rewriting

Denis Miginsky

The question of the day

What is the most expensive product for each company?

Subquery

SELECT query expects **tables** under some statements (FROM, JOIN, …)

SELECT query itself produces **table**-like structure (the application of any operator from the relational algebra is always producing a relation)

Thus, we can use SELECT-query instead of the actual table, cannot we?

Subquery example

SQL:

```
SELECT DISTINCT m.COMPANY
FROM MANUFACTURER m,
      (SELECT p.BILL_ID AS BILL
       FROM PRODUCT p, CATEGORY c
       WHERE p.WARE=c.WARE AND c.CLASS='Food') b
WHERE m.BILL_ID=b.BILL;
```

Q: What does this query do?

Q: Can it be rewritten without subqueries?

Pros and cons

Pros:

- Easier to design
- Query is more structured and better reflects the original question
- Some queries could not be written without subqueries

Cons:

- Worse performance in general

Tips:

- Fill free to use subqueries to create the initial query design
- Next, try to rewrite the query with as few subqueries as possible
- Try to keep the subqueries as simple as possible. Move all the complexity to the parent query.

Subqueries under WHERE: IN statement

SQL:

```
SELECT DISTINCT COMPANY
FROM MANUFACTURER
WHERE BILL_ID IN (SELECT BILL_ID
                  FROM PRODUCT
                  WHERE WARE IN (SELECT WARE
                                 FROM CATEGORY
                                 WHERE CLASS='Food'));
```

Limitations for IN statements

IN expects a “list” of values of the same type.

Thus, it expects that the subquery will return the result with the single column, that is compatible with the expression before IN by the type.

In general, you can use a tuple on the left and the corresponded projection (by the type) on the right.

Correlated subqueries: EXISTS

SQL:

```
SELECT DISTINCT p.WARE
FROM PRODUCT p
WHERE EXISTS (SELECT 1
              FROM MATERIAL mt1
              WHERE mt1.BILL_ID=p.BILL_ID  --dep. from outer q.
                AND mt1.WARE='Water');
```

Q: What does this query do?

Q: Can it be rewritten without subqueries?

NOT EXISTS

SQL:

```
SELECT DISTINCT p.WARE
FROM PRODUCT p
WHERE NOT EXISTS (SELECT 1
                  FROM MATERIAL mt
                  WHERE mt.BILL_ID=p.BILL_ID);
```

<the same questions here>

Notes on EXISTS

EXISTS checks if the subquery returns at least one row. The structure and values of the row do not matter (so it is the typical pattern to use constant-value projections with value 0 or 1).

Usually it can be rewritten with JOIN and DISTINCT in the projection.

NOT EXISTS is opposite to EXISTS.

Usually it can be rewritten with OUTER JOIN and the explicit IS NULL check, or with EXCEPT.

Lexical scope in subqueries

The subquery forms its own lexical scope that is nested to the scope of the parent query:

- The parent query can see only the result of the subquery, but not its local "variables".
- The **renaming** (AS) operation is essential for subqueries (unlike plain queries where the renaming is purely cosmetic)
- Correlated subqueries can appear in selection and projection sections (under SELECT, WHERE, HAVING, but not under FROM)
- The correlated subquery can see everything from the parent query that is available for the statement where it appears (consider HAVING/GROUP BY limitations for instance)

Question

For each product we need all the materials. There must be one row per product ware.

Looks familiar?

Can we rewrite this without GROUP BY?

What if we need the list of materials to be ordered?

Correct query

SQL:

```
SELECT p.WARE,
       (SELECT GROUP_CONCAT(WARE)
        FROM (SELECT DISTINCT mt1.WARE
              FROM MATERIAL mt1, PRODUCT p1
              WHERE mt1.BILL_ID=p1.BILL_ID
                AND p1.WARE=p.WARE
              ORDER BY mt1.WARE))
FROM (SELECT DISTINCT WARE FROM PRODUCT) p
ORDER BY p.WARE;
```

Is it an aggregation?

```
SELECT WARE, PRICE
FROM PRODUCT
ORDER BY PRICE DESC
LIMIT 1;
```

Q: What is the meaning of this query?

Q: Can we do the same for each company?

Answer to the question of the day

```
SELECT DISTINCT m.COMPANY,
        (SELECT p1.WARE
         FROM PRODUCT p1, MANUFACTURER m1
         WHERE p1.BILL_ID=m1.BILL_ID
           AND m1.COMPANY=m.COMPANY
         ORDER BY p1.PRICE DESC
         LIMIT 1) AS EXP_WARE,
        (SELECT p1.PRICE
         FROM PRODUCT p1, MANUFACTURER m1
         WHERE p1.BILL_ID=m1.BILL_ID
           AND m1.COMPANY=m.COMPANY
         ORDER BY p1.PRICE DESC
         LIMIT 1) AS PRICE
FROM MANUFACTURER m;
```

Problem: a little bit bulky, there are two almost identical queries

Problem: the performance is not perfect

More efficient variants

```
SELECT DISTINCT m.COMPANY,
       … AS EXP_WARE,
       … AS PRICE
FROM (SELECT DISTINCT COMPANY FROM MANUFACTURER) m;
```

```
SELECT m.COMPANY,
       … AS EXP_WARE,
       … AS PRICE
FROM (SELECT DISTINCT COMPANY FROM MANUFACTURER) m;
```

```
SELECT m.COMPANY,
       … AS EXP_WARE,
       … AS PRICE
FROM MANUFACTURER m
GROUP BY m.COMPANY;
```

Subqueries as values

```
SELECT DISTINCT m.COMPANY
FROM MANUFACTURER m
WHERE (SELECT COUNT(BILL_ID) FROM MANUFACTURER m1
       WHERE m1.COMPANY=m.COMPANY) > 2;
```

Problem: the performance!

Other variants

```
SELECT m.COMPANY
FROM (SELECT DISTINCT COMPANY FROM MANUFACTURER) m
WHERE (SELECT COUNT(BILL_ID) FROM MANUFACTURER m1
       WHERE m1.COMPANY=m.COMPANY) > 2;
```

```
SELECT m.COMPANY,
       (SELECT COUNT(BILL_ID) FROM MANUFACTURER m1
        WHERE m1.COMPANY=m.COMPANY) AS CNT
FROM (SELECT DISTINCT COMPANY FROM MANUFACTURER) m
WHERE CNT > 2;
```

```
SELECT DISTINCT m.COMPANY          --the fastest one
FROM MANUFACTURER m
GROUP BY m.COMPANY
HAVING COUNT(BILL_ID)>2;
```

COUNT vs EXISTS: performance

```
SELECT DISTINCT m.COMPANY
FROM MANUFACTURER m
WHERE EXISTS (SELECT 1 FROM MANUFACTURER m1, MATERIAL mt1
              WHERE mt1.BILL_ID=m1.BILL_ID
                AND m.COMPANY=m1.COMPANY);
```

```
SELECT DISTINCT m.COMPANY
FROM MANUFACTURER m
WHERE (SELECT COUNT() FROM MANUFACTURER m1, MATERIAL mt1
              WHERE mt1.BILL_ID=m1.BILL_ID
                AND m.COMPANY=m1.COMPANY)>0;
```

Q: Which one is better?

The best variant: no COUNT/EXISTS

```
SELECT DISTINCT m.COMPANY
FROM MANUFACTURER m
JOIN MATERIAL mt
ON mt.BILL_ID=m.BILL_ID;
```

COUNT vs NOT EXISTS: performance

```
SELECT DISTINCT m.COMPANY
FROM MANUFACTURER m
WHERE NOT EXISTS (SELECT 1 FROM MANUFACTURER m1, MATERIAL mt1
                  WHERE mt1.BILL_ID=m1.BILL_ID
                    AND m.COMPANY=m1.COMPANY);
```

```
SELECT DISTINCT m.COMPANY
FROM MANUFACTURER m
WHERE (SELECT COUNT() FROM MANUFACTURER m1, MATERIAL mt1
       WHERE mt1.BILL_ID=m1.BILL_ID
         AND m.COMPANY=m1.COMPANY)=0;
```

Q: Which one is better?

The best variant

```
SELECT DISTINCT m.COMPANY
FROM MANUFACTURER m
LEFT JOIN MATERIAL mt
ON mt.BILL_ID=m.BILL_ID
WHERE mt.BILL_ID IS NULL;
```

Q: Is it correct? Is it equivalent to the previous ones?

Pros and cons

	Performance	Flexibility
COUNT	**The worst**	**The best**
EXISTS	**Bad**	**Good**
JOIN	**The best**	**The Worst**

Notes on performance and flexibility

- Being closer to the formal model (relational algebra) means better performance: plain query vs nested queries
- Better flexibility means worse performance: EXISTS/COUNT vs JOIN
- Highly specialized/higher level operations provide more information to the query planner, and it can make a better plan: EXISTS vs COUNT, equijoin vs θ-join

Typical situations there subqueries are essential

- The aggregation should be used, and more complex manipulations with groups are required than standard aggregators can provide:
  - Ordering
  - Sampling
  - …
- Set operations should be used with subsequent manipulations with results:
  - Aggregation
  - Join
  - …
- The appropriate level of query flexibility is necessary: one needs to change conditions in a parent query and subqueries relatively independently.
- …