Database

SQL

Nested Queries

Splitting Query

Copy

Permanent

Copy of Table

We can make a copy of a subquery in a new **table**.

```
CREATE TABLE singapore_customer AS
  SELECT *
  FROM customers c
  WHERE c.country = 'Singapore';
```

```
SELECT cs.last_name, d.name
FROM singapore_customer cs, downloads d
WHERE cs.customerid = d.customerid;
```

Splitting Query

Copy

Temporary

Copy of Table

We can make a copy of a subquery in a **temporary table**.

```
CREATE TEMPORARY TABLE singapore_customer AS
  SELECT *
  FROM customers c
  WHERE c.country = 'Singapore';
```

Note

A temporary table only exists for the duration of the database session.

Splitting Query

Copy

Unchanged

No Change in Copy

The copies do **not** change when the **base table** *(in the example, the table* ***customers****)* change *(i.e.,* ***INSERT****,* ***DELETE****,* ***UPDATE****, etc)*.

Note

It is **rarely** a good idea. Most of the time, the same query can be **rewritten** as a simple query.

```
SELECT c.last_name, d.name
FROM customers c, downloads d
WHERE c.country = 'Singapore'
  AND c.customerid = d.customerid;
```

Splitting Query

Query

View

Visible Change

It could be a good idea since `VIEW` change when the **base table** change. Unfortunately, views are **unmaterialized** and **materialized views** need to be refreshed.

recomputed
fresh

```
CREATE VIEW singapore_customer AS
  SELECT *
  FROM customers c
  WHERE c.country = 'Singapore';
```

Note

Most of the time, the same query can be **rewritten** as a simple query.

Splitting Query

WITH cte1 AS (...),
cte2 AS (...)

Query

CTE

A Single Query

Common table expression *(CTE)* is a copy of a subquery in a temporary table that only exists for the query.

outer query

```
WITH singapore_customer AS
  ( SELECT *
    FROM customers c
    WHERE c.country = 'Singapore' )
SELECT cs.last_name, d.name
FROM singapore_customer cs, downloads d
WHERE cs.customerid = d.customerid;
```

CTE => computed first (1)

inner query (2)

*Most of the time, the same query can be **rewritten** as a simple query.

Splitting Query

Query

FROM

required up to psql 15

A Single Query

We can use **subquery** in the FROM clause. As a good practice, you should rename the result using AS operator.

SFW1

```
SELECT cs.last_name, d.name
FROM ( SELECT *
       FROM customers c
       WHERE c.country = 'Singapore' ) AS cs, downloads d
WHERE cs.customerid = d.customerid;
```

SFW2

*Most of the time, the same query can be rewritten as a simple query.

Splitting Query

count

value

Query

SELECT

Scalar Subquery

We can use **subquery** in the SELECT clause, but it must return **only one column and one row** *(also known as **scalar subquery**)*.

up to

returns one value

```
SELECT (
  SELECT COUNT(*) FROM customers c
  WHERE c.country = 'Singapore' );
```

```
SELECT COUNT(*)
FROM customers c
WHERE c.country = 'Singapore';
```

*Most of the time, the same query can be rewritten without nesting.

Splitting Query

Remark

Readable and Maintainable

Copies, temporary tables, views, common table expressions*, and nested queries have legitimate and appropriate usage.

It is --*however*-- recommended to seek simpler solutions first or, at least, to be able to justify their usage.

These may not always yield **readable** or **efficient** queries.

*We do not discuss recursive queries here.

Nesting Query

7:12

WHERE

X IN Set

IN

Computing Tuples

We can use subquery in `WHERE` clause to compute the tuples for `IN` clause.

```
SELECT d.name
FROM downloads d
WHERE d.customerid IN (
    SELECT c.customerid
    FROM customers c
    WHERE c.country = 'Singapore'
);
```

compute

exist → T

not → F

*Most of the time, the same query can be rewritten as a simple query.

Nesting Query

WHERE

ANY

Equals to Any

The following two queries are the same. Never use comparison to a subquery without specifying the quantifier ALL or ANY.

```
SELECT d.name
FROM downloads d
WHERE d.customerid IN (
    SELECT c.customerid
    FROM customers c
    WHERE c.country = 'Singapore'
);
```

=

```
SELECT d.name
FROM downloads d
WHERE d.customerid = ANY (
    SELECT c.customerid
    FROM customers c
    WHERE c.country = 'Singapore'
);
```

Nesting Query

WHERE

ALL

Outer Join, Except, Aggregate

ALL adds expressive power similar to that of `OUTER JOIN`, `EXCEPT`, and aggregate functions. The query below finds the **most expensive** games.

```
SELECT g1.name, g1.version, g1.price
FROM games g1
WHERE g1.price >= ALL (
    SELECT g2.price
    FROM games g2
);
```

*Change ALL to ANY and we print all games!

Nesting Query

WHERE

ALL

ALL to the Rescue

The following queries do not work *(`GROUP BY` limitation, syntax error)* but could be rewritten as a nested query using ALL subquery.

Error

```
SELECT g.name, g.version, g.price
FROM games g WHERE g.price = MAX(g.price)
```

```
SELECT g1.name, g1.version, g1.price
FROM games g1
WHERE g1.price = MAX(
  SELECT g2.price FROM games g2
);
```

OK

```
SELECT g1.name, g1.version, g1.price
FROM games g1
WHERE g1.price = ALL(
  SELECT MAX(g2.price)
  FROM games g2
);
```

Nesting Query

WHERE

EXISTS

Empty or Not Empty

EXISTS evaluates to **true** if the subquery has **some result**. It evaluates to **false** if the subquery has **no result**.

```
SELECT d.name
FROM downloads d
WHERE EXISTS (
  SELECT c.customerid
  FROM customers c
  WHERE d.customerid = c.customerid
    AND c.country = 'Singapore'
);
```

Note

The subquery is **correlated** to the query. The column `d.customerid` of the `customer` table of the outer query appears in the `WHERE` clause of the inner query.

We call such subquery as **correlated subquery**.

Nesting Query

Correlation

Subquery

Correlated Subquery

All subqueries can be correlated. The query below finds the names, versions, and prices of the games that are **most expensive** among the games of the same name.

```
SELECT g1.name, g1.version, g1.price
FROM games g1
WHERE g1.price >= ALL (
    SELECT g2.price
    FROM games g2
    WHERE g1.name = g2.name
);
```

Nesting Query

Correlation

Scoping

Nested Scoping

You can always use **column from an outer table** in an inner query but not the other way around. This is similar to **lexical scoping**.

```
SELECT c.customerid, d.name
FROM downloads d
WHERE d.customerid IN (
    SELECT c.customerid
    FROM customers c
    WHERE c.country = 'Singapore'
);
```

Nesting Query

Correlation

Scalar

Correlated SELECT

We can use subquery in SELECT clause, it still needs to be a **scalar subquery**, but it can be correlated.

```
SELECT (
    SELECT c.last_name
    FROM customers c
    WHERE c.country = 'Singapore'
      AND d.customerid = c.customerid
  ), d.name
FROM downloads d;
```

*Most of the time, the same query can be rewritten as a simple query.

Nesting Query

Negated Queries

Nested + Negation

!=

Where It Matters

Nested queries are powerful when combined with **negation**.

not equal to

```
SELECT c.customerid
FROM customers c
WHERE c.customerid NOT IN (
  SELECT d.customerid
  FROM downloads d
);
```

=

```
SELECT c.customerid
FROM customers c
WHERE c.customerid <> ALL (
  SELECT d.customerid
  FROM downloads d
);
```

≡

```
SELECT c.customerid
FROM customers c
WHERE NOT EXISTS (
  SELECT d.customerid
  FROM downloads d
  WHERE c.customerid =
        d.customerid
);
```

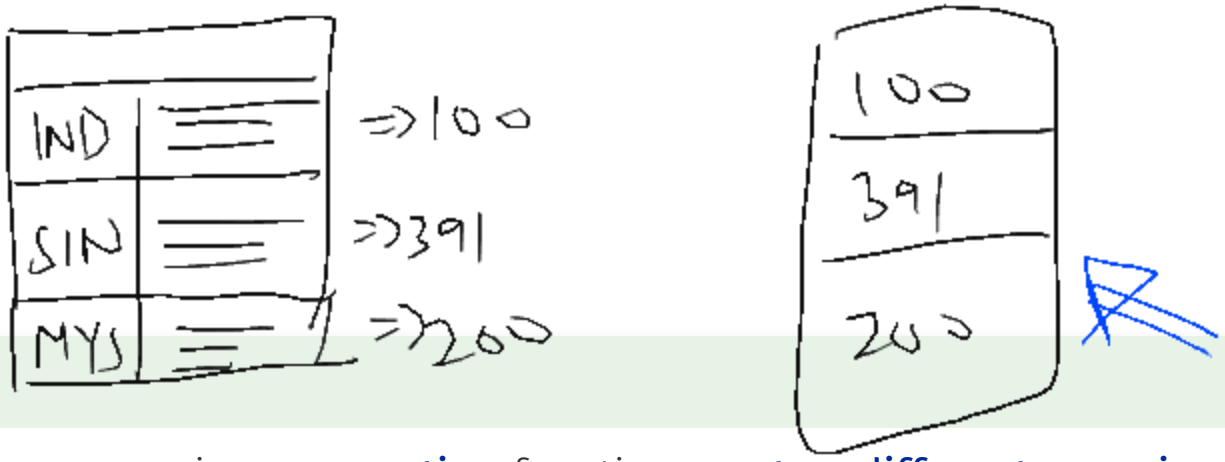
Note

The three queries above find the 22 customers who never downloaded a game.

Nesting Query

Nested Having

Nested + HAVING



Where It Matters

Nested queries may be necessary if we are using **aggregation** functions on **two different groupings**.

```
SELECT c1.country
FROM customers c1
GROUP BY c1.country
HAVING COUNT(*) >= ALL (
    SELECT COUNT(*)
    FROM customers c2
    GROUP BY c2.country
);
```

100, 391, 200

Note

What is the query on the left?

The query on the left finds the countries with the largest number of customers.

100 { 100 >= 100 AND
100 >= 391 AND
100 >= 200

Conclusion

Solving

Question

Who are our best customers in each country *(i.e., those who spend the most money among all the customers in their country)*?

```
SELECT c.customerid, c.country, SUM(g.price) AS total  -- find total spent by a customer id
FROM customers c, downloads d, games g                 --   need these 3 relations
WHERE c.customerid = d.customerid                      --   to connect c and d
  AND g.name = d.name AND g.version = d.version        --   to connect g and d
GROUP BY c.customerid, c.country                       --   needed to compute sum
HAVING SUM(g.price) >= ALL (                           -- such that the total spent by customer
 SELECT SUM(g2.price) AS total                         --   is greater than all other customer
 FROM customers c2, downloads d2, games g2
 WHERE c2.customerid = d2.customerid
   AND g2.name = d2.name AND g2.version = d2.version
   AND c2.country = c.country                          -- from the same country
 GROUP BY c2.customerid
);
```

Conclusion

Reading

$\neg\neg\forall v \in V\ (\exists d \in D\ has(c,d,v))$

$\neg(\exists v \in V\ \neg(\exists d \in D\ has(c,d,v)))$

Question

What does this query find? *(We need to understand the theoretical foundation of SQL to master reading and writing such queries)*

```
SELECT c.first_name, c.last_name
FROM customers c
WHERE NOT EXISTS (
    SELECT *
    FROM games g
    WHERE g.name = 'Aerified'
      AND NOT EXISTS (
        SELECT *
        FROM downloads d
        WHERE d.customerid = c.customerid
          AND d.name = g.name
          AND d.version = g.version ));
```

Answer

Find all customer such that there is **NO** version of Aerified that the customer has **NOT** downloaded.

Equivalently

Find all customer that has downloaded **ALL** version of Aerified.

```
postgres=# exit
Press any key to continue . . .
```