
Agenda

- Subqueries
- Subqueries and IN clause
- Subqueries in FROM clause
- ALL and ANY
- Correlated subqueries
- EXISTS
- Subqueries in WHERE clause
- Views

Subqueries

Subqueries are very intutive way of writing SQL queries. They break a problem into smaller problems and combine their result to get complete answer. Let's take few examples.

1. Given a `students` table, find all the students who have psp greater than the maximum psp of student of batch 2.

   `students`

id	name	psp	batch_id

~~~
Algorithm:

```python
Find maximum psp of batch 2, store it in x.
ans = []

for all students(S):
    if S.psp > x:
        ans.add(S)

return ans
```

The query will be:

```sql
SELECT *
FROM students
WHERE psp > (SELECT max(psp)
             FROM students
             WHERE batch_id = 2
);
```
~~~

2. Find all students whose psp is geater than psp of student with id = 18
   Algorithm:

```
Find the psp of student with id = 18, store it in x.
Find all students with psp > x and give the answer.
```

Subqueries will be:

```sql
SELECT psp
FROM students
WHERE id = 18;
```

```sql
SELECT *
FROM students
WHERE psp > x;
```

The final query will be:

```sql
SELECT *
FROM students
WHERE psp > (SELECT psp
             FROM students
             WHERE id = 18);
```

Subqueries should always be enclosed in parenthesis. The above query is readable and intuitive becaue of subqueries.

In reality, we prefer to

- break problems into parts.
- solve smaller problems and use their result to solve bigger problems.

Most of the times, problems that we solve via subqueries can also be solved via some other smart trick. But subqueries make our queries easier to understand and create.

Tradeoff of subqueries

The tradeoff is bad performance. For example, consider the following:

```sql
SELECT *
FROM students
WHERE psp > 18;
```

Without subquery:

```
students = []
ans = []

for student S: students:
    if S.psp > 18:
        ans.add(S)

return ans
```

With subquery:

```
for student S: students:
    ans = []
    for student O: students:
        if O.id = 18:
            ans.add(O)

return ans[0][psp]
```

The above query takes O(N^2). The subquery gets executed for every row. Hence, it leads to bad performance. Although, SQL optimizers help with performance improvement.

Example: Consider the table `film`. Find all the years where the average of the `rental_rate` of films of that year was greater than the global average of `rental_rate` (average of all films).

Algorithm:

1. Find global average.

```
SELECT avg(rental_rate)
FROM film;
```

2. Find average of every year.

   ```
   SELECT release_year, avg(rental_rate)
   FROM film
   GROUP BY release_year;
   ```

3. Get filtered groups.

   ```
   SELECT release_year, avg(rental_rate)
   FROM film
   GROUP BY release_year
   HAVING avg(rental_rate) > (
    SELECT avg(rental_rate)
    FROM film
   );
   ```

The subqueries we wrote till now gave a single value as output. But a query can give 4 types of outputs:

Number of rows	Number of columns	Output
1	1	Single value
1	m	Single row
m	1	Single column
m	m	Table

We were using >, <, =, <>, >=, <= operations because there was a single value. But what if, it is not just a single value? Let's take an example.

Subqueries and IN clause

Let's say there is a table called `users`. Find the names of students that are also the names of a TA. `users`

id	name	is_student	is_TA

This means, if there is a Naman who is a student and also a Naman who is a TA, show Naman in the output. It does not have to be the same Naman, just the name should be same.

If we had two different tables, `students` and `tas`, the query would have been like this:

```
SELECT DISTINCT S.name
FROM students S
JOIN tas T
ON S.name = T.name;
```

But here we have just one table. So, consider the following query:

```
SELECT DISTINCT S.name
FROM users S
JOIN users T
ON S.is_student = true
    AND T.is_TA = true
    AND S.name = T.name;
```

How many of you think this query using JOIN is complex? Let's try to solve it using subqueries.

Algorithm:

```
Get Name of all TA, store it in ans[].

Get students whose name is in ans.
```

Subqueries will be:

```
SELECT DISTINCT name
FROM users U
WHERE U.is_TA = true;
```

```
SELECT DISTINCT name
FROM users U
WHERE U.is_student = true
    AND U.name IN (/*result of first subquery*/);
```

Combining both of these:

```
SELECT DISTINCT name
FROM users U
WHERE U.is_student = true
```

```
    AND U.name IN (
        SELECT DISTINCT name
        FROM users U
        WHERE U.is_TA = true
    );
```

Subqueries in FROM clause

Now, we saw in last example that we got multiple values as output. Could we have comparison condition on these values? Let's look into it.

Find all of the students whose psp is not less than the smallest psp of any batch.

Algorithm:

```
Find minimum psp of every batch, store it in x[].

Find maximum psp from x[], store in y.

Print the details where psp is greater than y.
```

Subqueries will be:

```
SELECT min(psp)
FROM students
GROUP BY batch_id;
```

```
SELECT max(psp)
FROM x;
```

```
SELECT * FROM students
WHERE psp > y;
```

Combining the above subqueries:

```
SELECT *
FROM students
WHERE psp > (
    SELECT max(psp)
    FROM (
        SELECT min(psp)
        FROM students
        GROUP BY batch_id
    ) minpsps
);
```

Whenever you have a subquery in FROM clause, it is required to give it a name, hendce, `minpsps`.

We can have subquery in FROM clause as well. This subquery's output is considered like a table in itself upon which you can write any other read queries.

> *Note: You should name a subquery in FROM clause.*

ALL and ANY

Now let us say we want to find if a value x is greater than all the values in a set, we use ALL clause in such cases.

To understand this, consider `psp > ALL (10, 20, 30, 45)`. ALL compares left hand side with every value of the right hand side. If all of them return `True`, ALL will return `True`.

```
SELECT *
FROM students
WHERE psp > ALL (
    SELECT min(psp)
    FROM students
    GROUP BY batch_id
);
```

Similar to how AND has a pair with OR, ALL has a pair with ANY. ANY compares the left hand side with every value on the right hand side. If any of them returns `True`, ANY returns `True`.

Quiz 1

What is output of `x = ANY(a, b, c)` same as

Choices

- ☐ =
- ☐ ANY
- ☐ IN
- ☐ ALL

Correlated subqueries

Let's take an example first. Find all students whose psp is greater than average `psp` of their batch.

Algorithm: Query 1: Get students whose `psp` > x (say). Query 2: x stores average `psp` of student's batch.

Based upon which student we are considering, query 2 will give varying answers. Are the two subqueries independent of each other?

No, these two are correlated. Correlated subqueries are queries where the subquery uses a variable from the parent query.

```
SELECT *
FROM students
WHERE psp > x;
```

Here, the value of `x` (avg psp of batch) is dependent upon which student we are calculating it for as each student can have different batches.

Let's see a different set of subqueries:

```sql
SELECT avg(psp)
FROM students
WHERE batch_id = n;
```

```sql
SELECT *
FROM students
WHERE psp > y;
```

By putting the first query in place of `y`, will the final query work? No. Because there is no `n`. Assume you had the value of `n`, then? Put `S.batch_id` instead of `n`.

```sql
SELECT *
FROM students S
WHERE psp > (
    SELECT avg(psp)
    FROM students
    WHERE batch_id = S.batch_id
);
```

Here, this subquery is using a variable `S.batch_id` from the parent query. For every row from `students` table, we will be able to use the value of `batch_id` in the subquery.

EXISTS

There is another clause we can use. Let's say we want to find all students who are also TA given the two tables. Here `st_id` can be `NULL` if the TA is not a student.

`students`

id	name	psp

`tas`

id	name	st_id

Let's make the subquery:

```sql
SELECT st_id
FROM tas
WHERE st_id IS NOT NULL;
```

Final query will use the above subquery:

```sql
SELECT *
FROM students
WHERE id IN (
    SELECT st_id
    FROM tas
    WHERE st_id IS NOT NULL
);
```

Now see how we can use EXISTS for the above query.

```
SELECT *
FROM students S
WHERE EXISTS (
    SELECT st_id
    FROM tas
    WHERE tas.st_id = S.id
);
```

What EXISTS does is, for every row of `students` it will run the subquery. If the subquery returns any number of rows greater than zero, it returns `True`. In this query, finding `tas.st_id = S.id` is faster because of indexes. And as soon as MySQL finds one such row, EXISTS will return `True`. Whereas, in the previous query had to go through all of the rows to get the answer of subquery. So, this query is faster than the previous one.

Example 2

Find all the student names that have taken a mentor sesion.

`students`

id	name

`mentor_sessions`

s_id	stud_id	mentor_id

Consider this query:

```
SELECT *
FROM students
WHERE id IN (
    SELECT stud_id
    FROM mentor_sessions
);
```

The subquery will give a huge number of rows. Whereas, we just need to find if there is even a single row in `mentor_sessions` with some `stud_id`.

```
SELECT *
FROM students S
WHERE EXISTS (
    SELECT *
    FROM mentor_sessions
    WHERE stud_id = S.id
);
```

In this way, MySQL allows EXISTS to make faster queries.

Views:

Imagine in sakillaDB, I frequently have queries of the following type:

- Given an actor, give me the name of all films they have acted in.
- Given a film, give me the name of all actors who have acted in it.

Getting the above requires a join across 3 tables, `film`, `film_actor` and `actor`.

Why is that an issue?

- Writing these queries time after time is cumbersome. Infact imagine queries that are even more complex - requiring joins across a lot of tables with complex conditions. Writing those everytime with 100% accuracy is difficult and time-taking.
- Not every team would understand the schema really well to pull data with ease. And understanding the entire schema for a large, complicated system would be hard and would slow down teams.

So, what's the solution? Databases allow for creation of views. Think of views as an alias which when referred is replaced by the query you store with the view.

So, a query like the following:

```sql
CREATE OR REPLACE view actor_film_name AS

SELECT
   concat(a.first_name, a.last_name) AS actor_name,
   f.title AS file_name
FROM actor a
  JOIN film_actor fa
    ON fa.actor_id = a.actor_id
  JOIN film f
    ON f.film_id = fa.film_id
```

Note that a view is not a table. It runs the query on the go, and hence data redundancy is not a problem.

Operating with views

Once a view is created, you can use it in queries like a table. Note that in background the view is replaced by the query itself with view name as alias. Let's see with an example.

```sql
SELECT film_name FROM
actor_film_name WHERE actor_name = "JOE SWANK"
```

OR

```sql
SELECT actor_name FROM
actor_file_name WHERE film_name = "AGENT TRUMAN"
```

If you see, with views it's super simple to write queries that I write frequently. Lesser chances to make an error. Note that however, actor_file_name above is not a separate table but more of an alias.

An easy way to understand that is that assume every occurrence of `actor_file_name` is replaced by

```
(SELECT
   concat(a.first_name, a.last_name) AS actor_name,
   f.title AS file_name
FROM actor a
  JOIN film_actor fa
    ON fa.actor_id = a.actor_id
  JOIN film f
    ON f.film_id = fa.film_id) AS actor_file_name
```

Caveat: Certain DBMS natively support materialised views. Materialised views are views with a difference that the views also store results of the query. This means there is redundancy and can lead to inconsistency / performance concerns with too many views. But it helps drastically improve the performance of queries using views. MySQL for example does not support materialised views. Materialised views are tricky and should not be created unless absolutely necessary for performance.

How to best leverage views

Imagine there is an enterprise team at Scaler which helps with placements of the students. Should they learn about the entire Scaler schema? Not really. They are only concerned with student details, their resume, Module wise PSP, Module wise Mock Interview clearance, companies details and student status in the companies where they have applied.

In such a case, can we create views which gets all of the information in 1 or 2 tables? If we can, then they need to only understand those 2 tables and can work with that.

More operations on views

How to get all views in the database:

```
SHOW FULL TABLES WHERE table_type = 'VIEW';
```

Dropping a view

```
DROP VIEW actor_file_name;
```

Updating a view

```
ALTER view actor_film_name AS

 SELECT
    concat(a.first_name, a.last_name) AS actor_name,
    f.title AS file_name
 FROM actor a
   JOIN film_actor fa
     ON fa.actor_id = a.actor_id
   JOIN film f
     ON f.film_id = fa.film_id
```

Note: Not recommended to run update on views to update the data in the underlying tables. Best practice to use views for reading information.

See the original create statement for a view

```sql
SHOW CREATE TABLE actor_film_name
```

That is all for today, thanks!

Solution to Quizzes:

> *-- Quiz1: Option C (IN) --*