Sample Tables

users

id	name	age
1	John	40
2	May	30
3	Jack	22

orders

id	user_id	note
1	1	some information
2	2	some comments
3	2	no comments
4	3	more comments

`Subquery` is a technique used to simplify SQL statements, and it is the first threshold for advanced SQL syntax. In the actual database system, a table often has dozens of columns. If `JOIN` is used to combine tables according to certain conditions, it may cause many redundant columns to be fetched.

Consider a situation: Suppose we want to find all the order records for a user whose name is "John", but I don't want to get the columns of the `users` table, and I don't want to use `SELECT` to define the columns to display one by one, what should I do?

At this point, subquery is a great tool for you.

To help solidify the new concepts, you can follow along by editing the provided table below:

Schema SQL

```
CREATE SCHEMA `new_schema` DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;

CREATE TABLE `new_schema`.`users` (

 `id` INT NOT NULL AUTO_INCREMENT COMMENT 'This is the primary index',

 `name` VARCHAR(45) NOT NULL DEFAULT 'N/A',

 `age` INT NULL,

 PRIMARY KEY (`id`)

);

INSERT INTO `new_schema`.`users` (`id`, `name`, `age`) VALUES

 (1, 'John', 40),

 (2, 'May', 30),

 (3, 'Tim', 22);
```

Query SQL

```
SELECT * FROM `new_schema`.`users`;

SELECT * FROM `new_schema`.`orders`;
```

Result

Query #1 **Execution time: 0ms**

id	name	age
1	John	40
2	May	30
3	Tim	22

Query #2 **Execution time: 0ms**

id	user_id	note
1	1	some information
2	2	some comments

3	2	no comments
4	3	more comments

Equal Condition

First of all, let's take a look at a SQL statement used to address the above situation:

```
SELECT * FROM `new_schema`.`orders`

WHERE user_id = (

  SELECT id FROM `new_schema`.`users`

  WHERE name = 'John'

);
```

Result:

id	user_id	note
1	1	some information

The special part of this SQL is the query condition of the `WHERE` keyword. Its condition is the result of another SQL statement. This is the subquery technique, which uses nested query statements to complete a conditional query.

So if we want to analyze this SQL statement, we can look at the SQL statement of the subquery independently first:

```
SELECT id FROM `new_schema`.`users`

WHERE name = 'john';
```

Its result is:

id
1

Since the retrieved data is very simple, with only one record, the original SQL statement can be transformed into the following search statement through the "=" operator:

```
SELECT * FROM `new_schema`.`orders`
WHERE user_id = 1;
```

Contain Condition

In addition to the determination of equality, we can also change the SQL into a statement that can solve the condition that contains lots of data. In order to do that, we have to update the operator from `=` to `IN`. The reason is that we can use the `IN` keyword to filter out the data with multiple conditions.

Suppose our situation is: We need to find all order records for the user whose name contains the letter "j".

```
SELECT * FROM `new_schema`.`orders`
WHERE user_id IN (
  SELECT id FROM `new_schema`.`users`
  WHERE name LIKE '%j%'
);
```

Just like the previous equal example, we can first split the above SQL statement into:

```
SELECT id FROM `new_schema`.`users`
WHERE name LIKE '%j%'
```

This gives us the following result:

id
1
3

Through the result, our entire SQL can be transformed into:

```
SELECT * FROM `new_schema`.`orders`
WHERE user_id IN (1, 3);
```

And the execution result is:

id	user_id	note
1	1	some information
4	3	more comments

Common Mistakes

Singular and Plural problem

In the previous examples that use "equal" and "contain", it is important to note that both can be used when only one record is fetched by the subquery. But if multiple records are fetched, we can only use the contain method. The reason is that if we use the `=` operator, only a single value is accepted. That is to say, the following SQL will cause an execution error if the subquery has multiple records:

```
SELECT * FROM `new_schema`.`orders`
WHERE user_id = (
  SELECT id FROM `new_schema`.`users`
  WHERE name LIKE '%j%'
);
```

Therefore, we should update the statement to:

```
SELECT * FROM `new_schema`.`orders`
WHERE user_id IN (
  SELECT id FROM `new_schema`.`users`
  WHERE name LIKE '%j%'
);
```

Redundant Columns in Subqueries

Another common mistake is to use subqueries that do not properly use `SELECT` to get the appropriate columns. As previously demonstrated, `SELECT *` can help us avoid writing out all of the columns one by one, but it may also cause us to forget to write them out when we only want a few specific columns.

For example, in the example used in this article, our subquery only needs the `id` column. Suppose it is written as `*`, so the `name`, `age`, and other columns are all retrieved. This will lead to an execution error:

```
SELECT * FROM `new_schema`.`orders`
WHERE user_id IN (
  SELECT * FROM `new_schema`.`users`
  WHERE name LIKE '%j%'
);
```

These two mistakes are very common when first learning how to use subqueries. We want to share them with you, and we hope you can use them smoothly in the future!