Database Systems Lab

Subqueries

In the following, we will look at some examples of subqueries (nested `SELECT`s) and learn about some new logical operators (`IN`, `EXISTS`, `ALL`, `ANY`).

We have already encountered both a subquery and the `IN` (`NOT IN`) operator before, for example, when querying unmatched rows in a table:

```
SELECT title, publisher, price
    FROM BOOK_LIBRARY.books
    WHERE book_id NOT IN
        (SELECT book_id
            FROM BOOK_LIBRARY.book_items);
```

We also used a subquery to compute an atomic value that cannot be computed in a simple way:

```
SELECT last_name, first_name, birth_date
    FROM BOOK_LIBRARY.customers
    WHERE birth_date =
        (SELECT MIN(birth_date)
            FROM BOOK_LIBRARY.customers);
```

We can use a subquery to determine the number of rows in a calculated table:

```
SELECT COUNT(*) FROM
    (SELECT title, topic, number_of_pages, price, publishing_date
        FROM BOOK_LIBRARY.books
        WHERE extract(year from publishing_date) BETWEEN 1990 AND 2000
            OR price BETWEEN 1000 AND 3000
            OR topic = 'Science Fiction' AND number_of_pages < 20
        ORDER BY topic DESC NULLS LAST, number_of_pages);
```

Of course, this task can be solved without a subquery, in a much simpler way:

```
SELECT COUNT(*)
    FROM BOOK_LIBRARY.books
    WHERE extract(year from publishing_date) BETWEEN 1990 AND 2000
        OR price BETWEEN 1000 AND 3000
        OR topic = 'Science Fiction' AND number_of_pages < 20;
```

The solution using the subquery would rather be preferred only if we already have the inner `SELECT` at our disposal and just want to quickly determine the number of rows in the result.

Subqueries can be used, among others, in the following situations:

- After the `FROM` keyword in a `SELECT` statement (see the third example above): in this case, the query uses a calculated table instead of a stored table. We take advantage of the fact that the result of each `SELECT` is itself a table, so we can write a nested `SELECT` wherever a table is required.
- In the case of operators where one of the operands is a list (e.g., `IN`), the list can be specified either with enumeration or with a subquery. It is only necessary to make sure that the schema of the table defined by the subquery is compatible with the other operand of the operator. In the first example above, the left operand of the `NOT IN` operator is `book_id`, so the query to the right should result in a single-column table in which the type of that single column must be convertible to the type of `book_id`.
- Every atomic value can be replaced with a subquery that results in a single-row and single-column table (see the second example above).

The syntax of subqueries is very simple: we write an inner `SELECT` in parentheses somewhere inside of an outer `SELECT`.

Let's now delve into some operators that require the use of subqueries.

Although the `IN` and `NOT IN` operators are not always used with a subquery, it is still often used to generate the list in which to search for a value (or group of values). As we know, `IN` is a two-operand operator that returns true if its left operand occurs in the list specified in its right operand. And `NOT IN` is true when it doesn't occur. Let's look at two examples.

List all data of the copies of books with a topic of *Thriller*.

```
SELECT *
    FROM BOOK_LIBRARY.book_items
    WHERE book_id IN
        (SELECT book_id
            FROM BOOK_LIBRARY.books
            WHERE topic = 'Thriller');
```

At first, we might not solve this task in this way, but with join:

```
SELECT bi.*
    FROM BOOK_LIBRARY.books b JOIN BOOK_LIBRARY.book_items bi
        ON b.book_id = bi.book_id
    WHERE topic = 'Thriller';
```

In fact, the `IN` operator used in the above way can always be converted to equijoin, and it is also true the other way around if data is retrieved from only one of the tables. The two solutions are equivalent to each other in this case. We could also use natural join:

```
SELECT book_item_id, book_id, theoretical_value
    FROM BOOK_LIBRARY.books NATURAL JOIN BOOK_LIBRARY.book_items
    WHERE topic = 'Thriller';
```

Unfortunately, qualified column reference cannot be used in this case, so we are forced to list all the columns of the `book_items` table in the `SELECT` list.

List the full name and date of birth of patrons whose initials match the initials of an author. Sort the result alphabetically by name.

```
SELECT last_name, first_name, birth_date
    FROM BOOK_LIBRARY.customers
    WHERE (substr(last_name, 1, 1), substr(first_name, 1, 1)) IN
        (SELECT substr(last_name, 1, 1), substr(first_name, 1, 1)
            FROM BOOK_LIBRARY.authors)
    ORDER BY last_name, first_name;
```

We have nine such patrons. Now let's list the authors with the same initials, i.e., swap the input tables for the outer and inner `SELECT`:

```
SELECT last_name, first_name, birth_date
    FROM BOOK_LIBRARY.authors
    WHERE (substr(last_name, 1, 1), substr(first_name, 1, 1)) IN
        (SELECT substr(last_name, 1, 1), substr(first_name, 1, 1)
            FROM BOOK_LIBRARY.customers)
    ORDER BY last_name, first_name;
```

There are ten of them, because one pair of initials occurs twice. The left operand of the `IN` operator was now a two-element list of values (value lists must always be enclosed in parentheses), so the right side had to be a `SELECT` resulting in a two-column table. We could have solved the problem with one value if we had concatenated the two letters of the initials into one string.

It would also be nice to see a list in which we could see patron-author pairs with the same initials in one row. This can be solved with an equijoin that is not based on the equality of a primary key and a foreign key:

```
SELECT c.last_name, c.first_name, c.birth_date,
    a.last_name, a.first_name, a.birth_date
    FROM BOOK_LIBRARY.customers c JOIN BOOK_LIBRARY.authors a
        ON substr(c.last_name, 1, 1) = substr(a.last_name, 1, 1)
        AND substr(c.first_name, 1, 1) = substr(a.first_name, 1, 1)
    ORDER BY c.last_name, c.first_name;
```

Let's move on to the `EXISTS` operator. This is a single-operand operator whose only operand is a subquery and returns true if the table resulting from the subquery has at least one row, i.e., it is not an empty table. Its counterpart (`NOT EXISTS`) returns true if the subquery results in an empty table. `IN` can be converted to `EXISTS`, and `NOT IN` can be converted to `NOT EXISTS`. Consider the following example again:

List all data of the copies of books with a topic of *Thriller*.

```
SELECT *
    FROM BOOK_LIBRARY.book_items bi
    WHERE EXISTS
        (SELECT *
            FROM BOOK_LIBRARY.books
            WHERE book_id = bi.book_id AND topic = 'Thriller');
```

A subcondition has been added to the condition of the inner `SELECT`, which compares the column previously to the left of `IN` with the column specified in the `SELECT` list of the subquery to the right of `IN`. Note that in the inner `SELECT`, we did not need to qualify the `book_id` attribute of the `books` table, because the inner `SELECT` takes columns from the table(s) specified in its own `FROM` by default.

There is an important difference between the subquery shown in this example and all previous subqueries: here, in the inner `SELECT`, we referenced a column from the table specified in the outer `SELECT`. Such nested `SELECT`s cannot be run on their own. If you try to select it in SQL Developer and press Ctrl+Enter, you will receive an error message. This is because the subquery depends on the outer `SELECT`. Such subqueries are called *correlated subqueries*, while those that can be run independently are called *independent subqueries*.

Now let's look again at the first task of this chapter.

List the title, publisher, and price of books that don't have a single copy in our library.

```
SELECT title, publisher, price
    FROM BOOK_LIBRARY.books b
    WHERE NOT EXISTS
        (SELECT *
            FROM BOOK_LIBRARY.book_items
            WHERE book_id = b.book_id);
```

Here, we have converted the `NOT IN` operator to the `NOT EXISTS` operator as described above.

One more note about the `EXISTS` operator: Since in this case the only question is whether there is a row in the result of the subquery at all, it does not matter what we write in the `SELECT` list of the subquery. For this reason, we most often use `*`, as it would be difficult to find anything shorter.

There are two more operators to mention, ALL and ANY. They also have a subquery as their only operand, but they are always preceded by a relational (comparing) operator with a left operand. So, their general form is as follows:

```
<expr> <op> {ALL|ANY} (<subquery>)
```

Suppose the subquery produces the following list: (v_1, v_2, …, v_n). The above condition, when using ALL, is equivalent to this:

<expr> <op> v_1 AND <expr> <op> v_2 AND ... AND <expr> <op> v_n

When using ANY, it is equivalent to this:

<expr> <op> v_1 OR <expr> <op> v_2 OR ... OR <expr> <op> v_n

Let's look at an example:

List the title, publisher, and price of books that have at least one copy that is worth more than the price of the book.

```
SELECT title, publisher, price
    FROM BOOK_LIBRARY.books b
    WHERE price < ANY
        (SELECT theoretical_value
            FROM BOOK_LIBRARY.book_items
            WHERE book_id = b.book_id);
```

We see that there is no such book. If we replace the < operator with the <= operator, we will only get one book, i.e., we only have one book with a copy that is worth exactly the price of the book, and all the other books have only copies that are worth less than the price of the book or copies of an unknown value (or the price of the book is unknown in the first place).

We can rewrite the above solution by using the MAX function instead of ANY:

```
SELECT title, publisher, price
    FROM BOOK_LIBRARY.books b
    WHERE price <
        (SELECT MAX(theoretical_value)
            FROM BOOK_LIBRARY.book_items
            WHERE book_id = b.book_id);
```

The two solutions are completely equivalent. In the same way, any relational operator combined with ALL or ANY can be converted into some kind of condition without ALL / ANY. For example, < ANY can be converted to < MAX, >= ANY to >= MIN, < ALL to < MIN, <= ALL to = MIN, = ANY to IN, and <> ALL to NOT IN. So, our first example querying the unmatched rows of the books table by using the NOT IN operator can also be written like this:

```
SELECT title, publisher, price
    FROM BOOK_LIBRARY.books
    WHERE book_id <> ALL
        (SELECT book_id
            FROM BOOK_LIBRARY.book_items);
```

What is the name of our oldest patron(s)?

We've solved this task before by using the MIN operator:

```
SELECT last_name, first_name
    FROM BOOK_LIBRARY.customers
    WHERE birth_date =
        (SELECT MIN(birth_date)
            FROM BOOK_LIBRARY.customers);
```

Now we know that this is equivalent to the following solution:

```
SELECT last_name, first_name
    FROM BOOK_LIBRARY.customers
    WHERE birth_date <= ALL
        (SELECT birth_date
            FROM BOOK_LIBRARY.customers);
```

But we can also solve the task with a correlated subquery:

```
SELECT last_name, first_name
    FROM BOOK_LIBRARY.customers c1
    WHERE NOT EXISTS
        (SELECT *
            FROM BOOK_LIBRARY.customers c2
            WHERE c2.birth_date < c1.birth_date);
```

If you like joins, you can use join:

```
SELECT last_name, first_name
    FROM BOOK_LIBRARY.customers JOIN
        (SELECT MIN(birth_date) min_birth_date
            FROM BOOK_LIBRARY.customers)
        ON birth_date = min_birth_date;
```

Here, we have joined (using equijoin) the `customers` table with a table that has a single row and a single column, containing the value of the smallest date of birth. Because inner join was used, only rows in which the date of birth equals this minimum date of birth are retained from the `customers` table.

Finally, let's see a solution that uses theta join to solve the problem:

```
SELECT c1.last_name, c1.first_name
    FROM BOOK_LIBRARY.customers c1 JOIN BOOK_LIBRARY.customers c2
        ON c1.birth_date <= c2.birth_date
    GROUP BY c1.library_card_number, c1.last_name, c1.first_name
    HAVING COUNT(*) =
        (SELECT COUNT(*)
            FROM BOOK_LIBRARY.customers);
```

We now joined the `customers` table with itself using the join condition that the date of birth in `c1` is less than or equal to the date of birth in `c2`, then formed groups based on patrons in `c1`, leaving only those groups (patrons in `c1`) for which all patrons remained on the side of `c2`, i.e., those who are paired with all patrons in the join, i.e., whose date of birth is less than or equal to that of all patrons. Don't forget that you cannot omit the library card number when grouping, otherwise any patrons with the same name would be grouped together. The first and last name must be included in the grouping because otherwise they could not be referenced in the `SELECT` list.