# Database Systems Lab

## Top *N* Analysis

In real life, we often ask questions such as “who are the top 3 contestants?” or “who are the 100 richest persons in the world?”. In such cases, we want to see the first 3, 100, or generally *N* items in an ordered list. In SQL, these types of queries are called *top N queries* or *top N analysis*. Different database management systems provide different tools for this purpose. We will now look at two methods supported by Oracle (there are more anyway).

The first method is to use `RONNUM`. `ROWNUM` is a so-called pseudocolumn that does not exist in stored tables but is contained in tables that result from queries. It is a column because you can use it in a query just like you can use an actual column. It is pseudo because it does not physically exist, yet we can reference it in any query. It is of type integer, and its value is a row number that starts at 1 and indicates the order in which the row is selected in the query result. Let’s look at an example:

```
SELECT title, publisher, ROWNUM
    FROM BOOK_LIBRARY.books;
```

The rows that appear will show the row numbers neatly in ascending order. But how can we use this for top *N* analysis? Obviously, in the result of a query like the one above, row numbers don’t tell us much. In fact, the rows of the result come up quite randomly (more precisely, in an order not known to us). The row number becomes meaningful if the result is sorted by some aspect. For example:

```
SELECT title, publisher, ROWNUM
    FROM BOOK_LIBRARY.books
    ORDER BY title;
```

If you run this query, you might be surprised: the row numbers for each row are still the same as in the previous query, and since we have now sorted the books by title, the row numbers look shuffled. However, if you think about it a little more, this is not so surprising: the `ORDER BY` clause is executed only *after* evaluating the `SELECT` list, remember? The value of `ROWNUM` is therefore determined *before* sorting is performed.

So, how can we use `ROMNUM` to achieve our goals? The answer is: with a subquery. If the sort is written in a nested `SELECT`, then in the outer `SELECT`, `ROWNUM` will have a meaningful value in each row:

```
SELECT title, publisher, ROWNUM
    FROM (SELECT title, publisher
        FROM BOOK_LIBRARY.books
        ORDER BY title);
```

The rows are now sorted both by title and by the value of `ROWNUM`. After that, it’s child’s play to display only the first 10 books from the list:

```
SELECT title, publisher, ROWNUM
    FROM (SELECT title, publisher
        FROM BOOK_LIBRARY.books
        ORDER BY title)
    WHERE ROWNUM <= 10;
```

Great! And how do we get the *last* 10 books from this list? Well, by sorting the books descending by title and printing the first 10 items:

```
SELECT title, publisher, ROWNUM
    FROM (SELECT title, publisher
        FROM BOOK_LIBRARY.books
        ORDER BY title DESC)
    WHERE ROWNUM <= 10;
```

These are indeed those books, but now, they are not listed in ascending order by title, and that's not pretty. Well, let's fix it with another sort:

```
SELECT title, publisher, ROWNUM
    FROM (SELECT title, publisher
        FROM BOOK_LIBRARY.books
        ORDER BY title DESC)
    WHERE ROWNUM <= 10
    ORDER BY title;
```

Now the row numbers are in descending order, but we have achieved our goal. One would think that the latter list could be obtained in another way:

```
SELECT title, publisher, ROWNUM
    FROM (SELECT title, publisher
        FROM BOOK_LIBRARY.books
        ORDER BY title)
    WHERE ROWNUM >
        (SELECT COUNT(*) FROM BOOK_LIBRARY.books) - 10;
```

However, this query returns an empty list. Similarly, the following query does not produce any rows either:

```
SELECT title, publisher, ROWNUM
    FROM (SELECT title, publisher
        FROM BOOK_LIBRARY.books
        ORDER BY title)
    WHERE ROWNUM = 2;
```

This should print the second item in the list but will also result in an empty table. Why don't these queries work?

The answer lies in "when", i.e., when the `ROWNUM` column gets a value. As we move through the rows in the table specified after the `FROM` keyword, we evaluate the `WHERE` condition first. If it's false, no new row is created in the result, we continue with the next row. If it's true, a new row will be created in the result, we assign the current value of `ROWNUM` to this new row and then increment the value of `ROWNUM` by one.

Take the example above. `ROWNUM` starts from 1. We take the first row, evaluate the condition for it, which is now `ROWNUM = 2`. This is false, so we move on to the next row without changing the value of `ROWNUM`. For the second row, `ROWNUM` is still 1, so the condition is false again, and so on. The same happens in the previous example, where the condition is `ROWNUM >` *n*, where *n* is a positive integer. For the first row, `ROWNUM` is set to 1, for which this condition is false, so there is no new row in the result, and we do not increment `ROWNUM`. However, this will also cause the condition to be false for all other rows, since `ROWNUM` will remain 1 throughout.

It is clear from these examples that you must be very careful with this method. Since Oracle 12c, there is a much simpler solution to the problem of top *N* analysis: the `SELECT` statement has been extended with the `FETCH` clause, which can be used to refer directly to the first *N* elements of a list. Without going into details about the syntax (for that, see the documentation), let's see some examples of what `FETCH` can do:

The first 10 (top 10) items in a list:

```
SELECT title, publisher
    FROM BOOK_LIBRARY.books
    ORDER BY title
    FETCH FIRST 10 ROWS ONLY;
```

The second 10 items in a list:

```
SELECT title, publisher
    FROM BOOK_LIBRARY.books
    ORDER BY title
    OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;
```

Second item in a list:

```
SELECT title, publisher
    FROM BOOK_LIBRARY.books
    ORDER BY title
    OFFSET 1 ROW FETCH NEXT 1 ROW ONLY;
```

Top 10% of a list:

```
SELECT title, publisher
    FROM BOOK_LIBRARY.books
    ORDER BY title
    FETCH FIRST 10 PERCENT ROWS ONLY;
```

The last 10 (bottom 10) items in a list:

```
SELECT * FROM
    (SELECT title, publisher
        FROM BOOK_LIBRARY.books
        ORDER BY title DESC
        FETCH FIRST 10 ROWS ONLY)
    ORDER BY title;
```

The first 6 items in a list, or more if item #6 matches the following based on the sort criteria:

```
SELECT title, publisher
    FROM BOOK_LIBRARY.books
    ORDER BY publisher
    FETCH FIRST 6 ROWS WITH TIES;
```

In this latter case, we first sort by publisher, then we take the first 6 rows, plus the additional rows where the publisher value is equal to the publisher value in row #6. In this example, row #6 has a publisher of “Amazon Digital Services LLC”, and the next 2 rows show the same publisher, resulting in a total of 8 rows.

As you can see, the `FETCH` clause makes top *N* analysis much easier and more convenient, since most of the time it does not require a subquery, and it is also capable of things that would be much more difficult to accomplish with `RONUM`.

I would like to draw your attention to a common mistake. For questions where you are looking for rows with minimum or maximum values in some data (e.g., oldest patron, author earning the most, patron borrowing the most, etc.), `ROWNUM` is not the best choice since you do not know how many rows meet your criterion (there may be several oldest patrons, several highest-earning authors, etc.). In such cases, you can use `FETCH` with the `WITH TIES` option or the `MIN` or `MAX` aggregate functions. For example, the query that determines the oldest patron(s) might be written like this:

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

Or:

```
SELECT last_name, first_name
    FROM BOOK_LIBRARY.customers
    ORDER BY birth_date
    FETCH FIRST ROW WITH TIES;
```

Finally, here are some more examples:

**List the ID and title of three (different) books that have been borrowed by students.**

```
SELECT DISTINCT book_id, title
    FROM BOOK_LIBRARY.books NATURAL JOIN BOOK_LIBRARY.book_items
        NATURAL JOIN BOOK_LIBRARY.borrowing
        JOIN BOOK_LIBRARY.customers ON library_card_number = customer_id
    WHERE category = 'student'
    FETCH FIRST 3 ROWS ONLY;
```

There was no need to sort here, because three arbitrary books were suitable. However, we needed three joins. The solution using `ROMNUM` does not work directly now, either:

```
SELECT DISTINCT book_id, title
    FROM BOOK_LIBRARY.books NATURAL JOIN BOOK_LIBRARY.book_items
        NATURAL JOIN BOOK_LIBRARY.borrowing
        JOIN BOOK_LIBRARY.customers ON library_card_number = customer_id
    WHERE category = 'student' AND ROWNUM <= 3;
```

This solution is not correct because if there are identical rows among the three selected books, `DISTINCT` will remove one (or even two) of them, so we will see fewer than three books in the result — even if there are three different books borrowed by students. Again, the correct solution is to use a subquery:

```
SELECT * FROM
    (SELECT DISTINCT book_id, title
        FROM BOOK_LIBRARY.books NATURAL JOIN BOOK_LIBRARY.book_items
            NATURAL JOIN BOOK_LIBRARY.borrowing JOIN
            BOOK_LIBRARY.customers ON library_card_number = customer_id
        WHERE category = 'student')
    WHERE ROWNUM <= 3;
```

**Who are the three highest-earning authors (i.e., those who received the most royalty for their books overall)? Also list the authors’ total earnings.**

```
SELECT last_name, first_name, SUM(royalty) total_income
    FROM BOOK_LIBRARY.authors NATURAL JOIN BOOK_LIBRARY.writing
    GROUP BY author_id, last_name, first_name
    ORDER BY total_income DESC NULLS LAST
    FETCH FIRST 3 ROWS WITH TIES;
```

The `NULLS LAST` option is very important here in sorting, because if we omit it, many total earnings with `NULL` values can end up at the top of the list. A better solution is to filter out authors whose total earnings are not known:

```
SELECT last_name, first_name, SUM(royalty) total_income
    FROM BOOK_LIBRARY.authors NATURAL JOIN BOOK_LIBRARY.writing
    GROUP BY author_id, last_name, first_name
    HAVING SUM(royalty) IS NOT NULL
    ORDER BY total_income DESC
    FETCH FIRST 3 ROWS WITH TIES;
```

With `ROMNUM`, we have to use a subquery again, but this solution will randomly select authors in case of a tie (instead of listing all of them):

```
SELECT * FROM
    (SELECT last_name, first_name, SUM(royalty) total_income
        FROM BOOK_LIBRARY.authors NATURAL JOIN BOOK_LIBRARY.writing
        GROUP BY author_id, last_name, first_name
        HAVING SUM(royalty) IS NOT NULL
        ORDER BY total_income DESC)
    WHERE ROWNUM <= 3;
```

## Data Dictionary Views

The concept of a data dictionary may be familiar to you from the lecture. A database is divided into two main parts: the *physical database* and the *data dictionary* (or *metadatabase*). The physical database contains data directly related to the modeled real-world problem (e.g., data about books, authors, patrons, and relationships between them), while the data dictionary contains metadata that describes data in the physical database (e.g., what attributes are stored about a book or author, which of them cannot take a `NULL` value, which should be unique, and what relationships exist between the entities). Thus, a metadatabase contains the *schema* of the database (description of entity types, attribute types, and relationship types, as well as various integrity constraints imposed on the database) and also some database-independent data, such as information about users, permissions, sessions, etc.

You may also be familiar with the concept of a *view* from the lecture. Most users do not have access to the entire database, they can see only a part of it based on their role. In other words, each user (or role) has their own view, which is a limited part of the entire database. We can specify this limitation with a `SELECT` statement in which we filter on certain columns and certain rows of certain tables. In SQL, a view is nothing more than a named `SELECT` statement. (How to create a view will be discussed in the next chapter.) Specifying a view in the `FROM` clause of a `SELECT` statement is equivalent to writing a nested `SELECT` in place of the view, where the nested `SELECT` is the query specified in the view definition.

*Data dictionary views* are views that make certain parts of the data dictionary available to us. Some require no special privileges to query, and some are visible only to users with DBA (database administrator) role. We'll look at some of them below.

The most general data dictionary view is `dictionary` (or `dict` for short), which contains the data dictionary tables and data dictionary views available to us (including itself) with a brief description. Let's look at its contents:

```
SELECT * FROM dictionary;
```

Running it as a regular user, this query resulted in 997 rows for me. If you notice, most of them have names beginning with `all_` or `user_`. (There are many others beginning with `dba_`, but we would only see them with DBA role.) Those beginning with `user_` contain the database objects (tables, views, synonyms, etc.) of the user performing the query, and those beginning with `all_` contain all objects to which the user has access. There are also other views with a name starting with `v$` or `gv$`, which are called dynamic performance views or fixed views.

They are dynamic because they are constantly changing while the database is in use. They are performance views because their contents relate primarily to performance. They are fixed because even the DBA cannot delete or modify them.

Note that the `dual` table is a data dictionary table, it is also included among the rows of the `dictionary` data dictionary view.

To learn more about data dictionary views, let's look at some tasks.

**List our own tables.**

```
SELECT * FROM user_tables;   -- or tabs
```

"New" users (who got access to the database server this semester) will probably get an empty table here, since we haven't created our own tables yet (we will soon). "Old bikers", on the other hand, may see their own tables. We can also query all of our own database objects (i.e., not just tables):

```
SELECT * FROM tab;
```

**List the tables of all users.**

```
SELECT * FROM all_tables;
```

Now we will see a lot of tables. Let's filter for the tables of the `BOOK_LIBRARY` user:

```
SELECT * FROM all_tables
    WHERE owner = 'BOOK_LIBRARY';
```

We can see the six well-known tables.

**Who am I?**

```
SELECT * FROM user_users;
```

**What other users may I know?**

```
SELECT * FROM all_users;
```

**List users with a name beginning with `U_`.**

```
SELECT * FROM all_users
    WHERE username LIKE 'U\_%' ESCAPE '\';
```

Of course, instead of escaping, this solution would also do the job:

```
SELECT * FROM all_users
    WHERE substr(username, 1, 2) = 'U_';
```

**Let's look at the structure of our tables.**

```
SELECT * FROM user_tab_columns;   -- or cols
```

Since "new" users still can't see anything, let's see something that everyone sees.

**Let's look at the structure of the `books` table of the `BOOK_LIBRARY` user.**

```
SELECT * FROM all_tab_columns
    WHERE owner = 'BOOK_LIBRARY' AND table_name = 'BOOKS';
```

Here you can see the name of the columns in the table, their type, whether they can take a `NULL` value, as well as other data.

**List the network connections of active sessions.**

```
SELECT * FROM v$session_connect_info;
```

This is one of the few dynamic performance views that are accessible to regular users. You may also want to look at `v$version`, `v$option`, and `v$nls_parameters`. The latter contains the current national language settings, including the default date and time format.