Top-N Queries
&
The New Row Limiting Clause
11g & 12c

Sponsored by 'c'. Letter 'c'.

by Przemysław Kruglej
11-2013

przemyslawkruglej.com
przemyslaw.kruglej@gmail.com

Table of Contents

1 Getting the Top-N Records From an Ordered Set & The New Row Limiting Clause – 11g & 12c

and a cup of tea if you're lucky

I bet my cup of raspberry-juiced black tea that, somewhere along your journey with Oracle, you had to write a query which was supposed to return only the top-n rows from an ordered set. Unlike some of the other databases, MySQL, for instance, Oracle does not provide a dedicated solution to this problem.

At least, not before the 12c hit the stage.

Before I introduce you to the nice *Row Limiting Clause*, let me show you why the first solution that comes to mind to solve the problem at hand, in Oracle's versions prior to 12c, is not the right one, and what voodoo tricks one has to perform to achieve the expected result.

1.1 NUM. ROWNUM. – before 12c

We all know Oracle's good ol' `rownum` fella. `Rownum` pseudocolumn is assigned to each fetched row in the result set. `Rownum` values are consecutive and start from 1. It might be tempting to employ him, and the `ORDER BY` clause, to achieve the required functionality:

```
CREATE TABLE emps (
  id          NUMBER NOT NULL,
  first_name  VARCHAR2(20) NOT NULL,
  last_name   VARCHAR2(20) NOT NULL,
  salary      NUMBER NOT NULL
);

INSERT INTO emps VALUES (1, 'Susannah', 'Dean', 8000);
INSERT INTO emps VALUES (2, 'Roland', 'Deschain', 9000);
INSERT INTO emps VALUES (3, 'Eddie', 'Dean', 6000);
INSERT INTO emps VALUES (4, 'Odetta', 'Holmes', 5000);
INSERT INTO emps VALUES (5, 'Detta', 'Walker', 4000);

COMMIT;

SELECT *
  FROM emps
WHERE
  rownum <= 3
ORDER BY salary DESC;

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         2 Roland               Deschain                   9000
         1 Susannah             Dean                       8000
         3 Eddie                Dean                       6000
```

And it turns out, that... it sometimes works! There is just this little word in between... what does it say? Huh? Sometimes? That's it – this approach is *very* wrong. It *might*, where might is the keyword here, return the correct result set (and that is why it is so wrong!). If you are a gambler and you want to take your chances, or you have a particular sense of humour and think that business users have it too, then use the above approach at your own risk.

However, if *sometimes* won't do for you, let's see what is actually wrong and how to fix it. What causes the "*sometimes*" is the fact, that:

> The rows are fetched first, and only then they are ordered.

This is because the `ORDER BY` clause is executed after the conditions of `WHERE` clause are applied and the data is returned, ready for ordering. The implication is that the database will **first fetch three rows** and **then those three rows will be ordered**, instead of ordering all the rows that match the conditions in the `WHERE` clause and returning top three of them.

Now, do you see why the above approach returns the correct result *sometimes*?

It depends on the order of fetching the rows. If it happens that the N fetched rows are those N rows that are in the top, then you will get a correct result – those top-N rows will be ordered after being retrieved and you will see expected set of rows, and this is just the case in the above example.

Now, to prove that I'm not lying here, let's: delete one of the top-3 records, insert another one in its place and run the query again:

```
DELETE FROM emps WHERE first_name = 'Roland';
> 1 rows deleted.

INSERT INTO emps VALUES (6, 'Jake', 'Chambers', 10000);
> 1 rows inserted.

COMMIT;

SELECT *
  FROM emps
WHERE
  rownum <= 3
ORDER BY salary DESC;

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         1 Susannah             Dean                       8000
         3 Eddie                Dean                       6000
         4 Odetta               Holmes                     5000
```

As you can see, the result is incorrect – we got "Odetta Holmes", where we should get the newly inserted record with "Jake Chambers". As explained, first three rows were retrieved and then they were sorted. In this case, one of those three records is not in the top-three. Basic `rownum`/`ORDER BY` approach fails miserably.

1.2 Query. Subquery. (this never gets old)

Since the rows are fetched first and then ordered, maybe we could try another approach. Let's use a subquery to return an ordered set, and then, in the outer query, restrict the number of rows to three using `rownum`:

```
SELECT id, first_name, last_name, salary
  FROM (
    SELECT emps.*, rownum AS rn
      FROM emps
    ORDER BY salary DESC
  )
WHERE rn <= 3;

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         1 Susannah             Dean                       8000
```

```
         3 Eddie                Dean                        6000
         4 Odetta               Holmes                      5000
```

O, my! The result is incorrect! I put the `rownum` inside the subquery on purpose to show you one more thing you should mind when using `rownum` pseudocolumn:

> ROWNUM value is assigned ***before*** ordering results.

The proof follows:

```
SELECT emps.*, rownum
  FROM emps
ORDER BY salary DESC;

     ID FIRST_NAME        LAST_NAME           SALARY     ROWNUM
------- ----------------- --------------- ---------- ----------
      6 Jake              Chambers             10000          5
      1 Susannah          Dean                  8000          1
      3 Eddie             Dean                  6000          2
      4 Odetta            Holmes                5000          3
      5 Detta             Walker                4000          4
```

Since we inserted the record with "Jake Chambers" as the last one, it is in this case fetched as the last one, and Jake gets value "5" as his `rownum`. After the records were retrieved and a `rownum` was assigned to each of them, the result set was sorted. Because of that fact, condition on `rn` in the outer query failed to deliver expected result.

To get our query to finally return what we need, the condition on `rownum` should be based on the `rownum` assigned in the outer query, not on the one assigned in the subquery:

```
SELECT *
  FROM (
    SELECT *
      FROM emps
    ORDER BY salary DESC
  )
WHERE rownum <= 3;

        ID FIRST_NAME           LAST_NAME                 SALARY
---------- -------------------- -------------------- ----------
         6 Jake                 Chambers                  10000
         1 Susannah             Dean                       8000
         3 Eddie                Dean                       6000
```

The correct result, at last! What happens here? First, all rows are ordered in the subquery, and then, in the outer query, `rownum` is assigned and only rows with `rownum` less than or equal to three are returned as the result set.

1.3 MIT. LIMIT.

Yes, *meet* the new *Row Limiting Clause*, introduced in Oracle 12c.

The *row limiting clause* allows you to put away the old `ORDER BY`/subquery/`rownum` approach for a much more convenient and easier to understand syntax for controlling the number of returned rows.

The new functionality allows you to select the top-N rows from the result set, with an optional offset from the beginning. The "N" may either be a number of rows to be fetched, or a given percentage of them.

If you want exactly N rows to be returned, you use the `ONLY` option. On the other hand, if several rows have the same values in columns used for ordering as the last row fetched (i. e. there a tie on the last position of the N rows returned), the `WITH TIES` option will tell Oracle to also fetch those records.

You cannot use the *row limiting clause* with either `FOR UPDATE` clause or in queries with `NEXTVAL` or `CURRVAL` pseudocolumns of a sequence. However, if you declare a cursor with both `FOR UPDATE` and *row limiting* clauses, you won't get an error until you try to open the cursor.

Let's take a look at the syntax:

```
SELECT *
  FROM emps
ORDER BY salary DESC
OFFSET {NUMBER} {ROW|ROWS}
FETCH {FIRST|NEXT} {<empty>|NUMBER [PERCENT]} {ROW|ROWS} {ONLY|WITH TIES}
```

Now, let me walk you through it, but before I do that, let's quickly glance at the data the examples will be based on:

```
SELECT * FROM emps;

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         1 Susannah             Dean                       8000
         3 Eddie                Dean                       6000
         4 Odetta               Holmes                     5000
         5 Detta                Walker                     4000
         6 Jake                 Chambers                  10000
```

1.3.1 OFFSET

`OFFSET` is an **optional** clause, which tells Oracle how many rows should be skipped from the beginning of the result set. The number of the rows is required, and must be followed by either of the keywords: `ROW` or `ROWS` – they are interchangeable, but one of them must be there.

You may not specify a percentage of rows to be skipped – it must be a number of rows.

The `OFFSET` part of the *row limiting clause* **does not require** the `FETCH` part to be present – if this is the case, all rows starting with the row at `OFFSET + 1` position will be returned. Time for examples.

1.3.1.1 Skip the first three rows

```
SELECT *
  FROM emps
OFFSET 3 ROW; -- ROW or ROWS - no difference!

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         5 Detta                Walker                     4000
         6 Jake                 Chambers                  10000
```

1.3.1.2 Skip the first three rows in an ordered set

```
SELECT *
  FROM emps
```

```
ORDER BY salary DESC
OFFSET 3 ROWS; -- ROW or ROWS - no difference!

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         4 Odetta               Holmes                     5000
         5 Detta                Walker                     4000
```

1.3.2 FETCH

Once again, the `FETCH` part syntax:

```
FETCH {FIRST|NEXT} {<empty>|NUMBER [PERCENT]} {ROW|ROWS} {ONLY|WITH TIES}
```

The optional `FETCH` part of the *row limiting clause* tells Oracle how many rows should be returned.

The `FETCH` keyword must be followed by either `FIRST` or `NEXT` – there's no difference between them – they are interchangeable, but either of them is required. `FIRST` or `NEXT` can be followed by either:

- a number of rows,
- percentage of rows to be fetched, which functionality is specified by adding the `PERCENT` keyword after the number,
- nothing! In this case, the default number of rows will be returned, which is 1.

Next, just like in the `OFFSET` part, either `ROW` or `ROWS` keyword must follow, and, again, they are interchangeable.

At the end, you must specify if you want the exact number of rows returned with the `ONLY` keyword, or, if there are ties in your data on the last position in your result set, also the rows with the same sort key as the row on the last position, using the `WITH TIES` keywords.

The `FETCH` part of the *row limiting clause* **does not require** the `OFFSET` part to be present. If this is the case, then the row limiting starts with the first row.

Let's add one more row with salary equal to 6000 to present differences between `ONLY` and `WITH TIES` options in the examples that follow after:

```
INSERT INTO emps VALUES (7, 'Oy', 'the Billy-bumbler', 6000);
COMMIT;
```

1.3.2.1 Fetch only the first row from an ordered set

```
SELECT *
  FROM emps
ORDER BY salary DESC
FETCH FIRST ROW ONLY; -- FIRST or NEXT, ROW or ROWS - no difference!

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         6 Jake                 Chambers                  10000
```

Note: if there was another record with salary equal to 10000, and we would change the `ONLY` to `WITH TIES` in the above example, both records would be returned.

1.3.2.2 Fetch only the first three rows from an ordered set

```
SELECT *
  FROM emps
ORDER BY salary DESC
FETCH NEXT 3 ROWS ONLY; -- FIRST or NEXT, ROW or ROWS - no difference!

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         6 Jake                 Chambers                  10000
         1 Susannah             Dean                       8000
         3 Eddie                Dean                       6000
```

Because we wanted `ONLY` three rows, "Oy the Billy-bumbler" was not returned, who has the same salary as "Eddie" on the last position.

What is important here is that the result is not deterministic, because the sorting key (`salary` in this case), is not unique, what may lead to a different data being returned when the query is run later, since there is a tie on the last position in the result set. This depends on the order of rows being fetched by Oracle and shouldn't be relied on.

1.3.2.3 Fetch the first three rows from an ordered set, and additional ties on the last position, if there are any

```
SELECT *
  FROM emps
ORDER BY salary DESC
FETCH FIRST 3 ROWS WITH TIES; -- FIRST or NEXT, ROW or ROWS - no difference!

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         6 Jake                 Chambers                  10000
         1 Susannah             Dean                       8000
         3 Eddie                Dean                       6000
         7 Oy                   the Billy-bumbler          6000
```

Note, that if there were two records with salary equal to 8000, each of them would be counted, and as a result, only 3 rows, in this situation, would be returned. Let's insert additional record with salary equal to 8000:

```
INSERT INTO emps VALUES (8, 'Susan', 'Delgado', 8000);
COMMIT;
```

And check the results:

```
SELECT *
  FROM emps
ORDER BY salary DESC
FETCH FIRST 3 ROWS WITH TIES; -- FIRST or NEXT, ROW or ROWS - no difference!

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         6 Jake                 Chambers                  10000
         1 Susannah             Dean                       8000
         8 Susan                Delgado                    8000
```

1.3.2.4 Fetch only 10% from an ordered set

```
SELECT *
  FROM emps
ORDER BY salary DESC
FETCH FIRST 10 PERCENT ROWS ONLY; -- FIRST or NEXT, ROW or ROWS

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         6 Jake                 Chambers                  10000
```

Note: row would also be returned if `1 PERCENT` was specified – always at least one row will be returned, if it exists.

1.3.3 OFFSET & FETCH Combined

When used together, the `FETCH` part starts working on the rows starting at row at position defined by value of `OFFSET` + 1. Finally, a couple of examples to show the full potential of row limiting clause.

1.3.3.1 Get the top three rows after skipping the first record from the top

```
SELECT *
  FROM emps
ORDER BY salary DESC
OFFSET 1 ROW
FETCH FIRST 3 ROWS ONLY;

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         1 Susannah             Dean                       8000
         8 Susan                Delgado                    8000
         3 Eddie                Dean                       6000
```

1.3.3.2 Get the top three rows after skipping the first record from the top, but if there are ties on the last position, get those additional rows

```
SELECT *
  FROM emps
ORDER BY salary DESC
OFFSET 1 ROW
FETCH FIRST 3 ROWS WITH TIES;

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         1 Susannah             Dean                       8000
         8 Susan                Delgado                    8000
         3 Eddie                Dean                       6000
         7 Oy                   the Billy-bumbler          6000
```

1.3.3.3 Offset exceeds number of rows

```
SELECT *
  FROM emps
ORDER BY salary DESC
OFFSET 10 ROWS
FETCH FIRST 3 ROWS ONLY;
```

```
no rows selected
```

1.3.3.4 Fewer records returned because of the offset (there are 7 rows total, we skip 5, and want 3)

```
SELECT *
  FROM emps
ORDER BY salary DESC
OFFSET 5 ROWS
FETCH FIRST 3 ROWS ONLY;

        ID FIRST_NAME           LAST_NAME                SALARY
---------- -------------------- -------------------- ----------
         4 Odetta               Holmes                     5000
         5 Detta                Walker                     4000
```

2 Further Reading & Useful Links

This is it. You wish! If you'd like to learn far more about the top-n queries and the new *row limiting clause*, I highly recommend the following three great articles in Oracle Magazine, all written by Thomas Kyte:

- http://www.oracle.com/technetwork/issue-archive/2013/13-sep/o53asktom-1999186.html – overview of three new features of Oracle 12c, including the FETCH FIRST/NEXT and OFFSET clauses.
- http://www.oracle.com/technetwork/issue-archive/2007/07-jan/o17asktom-093877.html – about Top-n and pagination queries.
- http://www.oracle.com/technetwork/issue-archive/2006/06-sep/o56asktom-086197.html – about ROWNUM and limiting results.

Row limiting clause in Oracle Documentation:

- http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_10002.htm#SQLRF55636 –row limiting clause.

I hope you enjoyed my article. If you have found any errors in it (even typos), you think that I haven't explained anything clearly enough or you have an idea how I could make the article better – please, do not hesitate to contact me, or leave a comment.