Probably the Coolest SQL Feature: Window Functions

Once you get a hang of the very peculiar syntax, SQL is a highly expressive and rich language offering incredible features at a declarative level. One of the coolest features are window functions, whose coolness is in no proportion to their incredibly low popularity. The low popularity can only be due to developers being oblivious of this cool stuff. Once you know window functions, you risk putting them all over the place.

What is a window function?

A window function looks at "windows" of your data while processing it. For example:

```
FIRST_NAME |
------------
Adam       | <-- UNBOUNDED PRECEDING
...        |
Alison     |
Amanda     |
Jack       |
Jasmine    |
Jonathan   | <-- 1 PRECEDING
Leonard    | <-- CURRENT ROW
Mary       | <-- 1 FOLLOWING
Tracey     |
...        |
Zoe        | <-- UNBOUNDED FOLLOWING
```

In the above example, the processing of a window function might be at the `CURRENT ROW`, which is "Leonard"'s row. Within that row's window, you can then access preceding or following

records. This is so extremely useful, e.g. when you want to show the person who's next to "Leonard".

SQL Syntax:

```
SELECT
  LAG(first_name, 1)
    OVER(ORDER BY first_name) "prev",
  first_name,
  LEAD(first_name, 1)
    OVER(ORDER BY first_name) "next"
FROM people
ORDER BY first_name
```

jOOQ syntax:

```
select(
   lag(PEOPLE.FIRST_NAME, 1)
     .over().orderBy(PEOPLE.FIRST_NAME).as("prev"),
   PEOPLE.FIRST_NAME,
   lead(PEOPLE.FIRST_NAME, 1)
     .over().orderBy(PEOPLE.FIRST_NAME).as("next"))
.from(PEOPLE)
.orderBy(PEOPLE.FIRST_NAME);
```

When executing the above, you can immediately see how each record's `FIRST_NAME` value can refer to the preceding and following first names:

```
|     PREV | FIRST_NAME |     NEXT |
|----------|------------|----------|
|   (null) |       Adam |   Alison |
|     Adam |     Alison |   Amanda |
|   Alison |     Amanda |     Jack |
|   Amanda |       Jack |  Jasmine |
|     Jack |    Jasmine | Jonathan |
|  Jasmine |   Jonathan |  Leonard |
| Jonathan |    Leonard |     Mary |
|  Leonard |       Mary |   Tracey |
|     Mary |     Tracey |      Zoe |
|   Tracey |        Zoe |   (null) |
```

(see this example in action on SQLFiddle)

Such window functions have their own `ORDER BY` clause, which is independent of the outer query's ordering. This fact is extremely useful when doing reporting. Furthermore, Sybase SQL Anywhere and PostgreSQL implement the SQL standard WINDOW clause, which allows for avoiding repetitive window definitions.

SQL Syntax:

```
SELECT
  LAG(first_name, 1) OVER w "prev",
  first_name,
  LEAD(first_name, 1) OVER w "next"
FROM people
WINDOW w AS (ORDER first_name)
ORDER BY first_name DESC
```

jOOQ 3.3 syntax:

```
 WindowDefinition w = name("w").as(
   orderBy(PEOPLE.FIRST_NAME));

 select(
   lag(PEOPLE.FIRST_NAME, 1).over(w).as("prev"),
   PEOPLE.FIRST_NAME,
   lead(PEOPLE.FIRST_NAME, 1).over(w).as("next"))
.from(PEOPLE)
.window(w)
.orderBy(PEOPLE.FIRST_NAME.desc());
```

Note that jOOQ makes the above `WINDOW` clause available to all SQL databases that support window functions, emulating it if it is not natively supported.

The above query results in:

```
|     PREV | FIRST_NAME |     NEXT |
|----------|------------|----------|
|   Tracey |        Zoe |   (null) |
|     Mary |     Tracey |      Zoe |
|  Leonard |       Mary |   Tracey |
| Jonathan |    Leonard |     Mary |
|  Jasmine |   Jonathan |  Leonard |
|     Jack |    Jasmine | Jonathan |
```

```
|   Amanda |       Jack |  Jasmine |

|   Alison |     Amanda |     Jack |

|     Adam |     Alison |   Amanda |

|   (null) |       Adam |   Alison |
```

Using frame definitions

Windows can have bounded or unbounded frames as illustrated previously using the`PRECEDING` and `FOLLOWING` keywords. This can be seen in action in an example that is almost equivalent to the previous `LEAD()` / `LAG()` examples:

SQL syntax:

```
SELECT
  FIRST_VALUE(first_name)
    OVER(ORDER BY first_name ASC
         ROWS 1 PRECEDING) "prev",
  first_name,
  FIRST_VALUE(first_name)
    OVER(ORDER BY first_name DESC
         ROWS 1 PRECEDING) "next"
FROM people
ORDER BY first_name ASC
```

jOOQ syntax:

```
select(
  firstValue(PEOPLE.FIRST_NAME)
    .over().orderBy(PEOPLE.FIRST_NAME.asc())
           .rowsPreceding(1).as("prev"),
  PEOPLE.FIRST_NAME,
  firstValue(PEOPLE.FIRST_NAME)
    .over().orderBy(PEOPLE.FIRST_NAME.desc())
           .rowsPreceding(1).as("next"))
.from(PEOPLE)
.orderBy(FIRST_NAME.asc());
```

The above example uses different `ORDER BY` clauses to access a `CURRENT ROW`‘s `PRECEDING`rows, and then just retaining the `FIRST_VALUE()`. As can be seen in the result, this has a slightly different semantics when it comes to the “first” and “last” records:

```
|     PREV | FIRST_NAME |     NEXT |

|----------|------------|----------|

|     Adam |       Adam |   Alison |
```

```
|     Adam |     Alison |   Amanda |
|   Alison |     Amanda |     Jack |
|   Amanda |       Jack |  Jasmine |
|     Jack |    Jasmine | Jonathan |
|  Jasmine |   Jonathan |  Leonard |
| Jonathan |    Leonard |     Mary |
|  Leonard |       Mary |   Tracey |
|     Mary |     Tracey |      Zoe |
|   Tracey |        Zoe |      Zoe |
```

Using PARTITION BY to create multiple windows

Often, you do not want a single window over your complete data set. Instead, you might prefer to `PARTITION` your data set into several smaller windows. The following example creates partitions for every first letter in a first name, similar to a phone book:

SQL syntax:

```
SELECT
  first_name,
  LEFT(first_name, 1),
  COUNT(*) OVER(PARTITION BY LEFT(first_name, 1))
FROM people
ORDER BY first_name
```

jOOQ 3.3 syntax:

```
select(
   PEOPLE.FIRST_NAME,
   left(PEOPLE.FIRST_NAME, 1),
   count().over().partitionBy(
     left(PEOPLE.FIRST_NAME, 1)))
.from(PEOPLE)
.orderBy(FIRST_NAME);
```

As can be seen below, the `COUNT(*)` window function counts all people with the same first letter:

```
| FIRST_NAME | LEFT | COUNT |
|------------|------|-------|
|       Adam |    A |     3 |
|     Alison |    A |     3 |
```

```
|     Amanda |    A |     3 |
|       Jack |    J |     3 |
|    Jasmine |    J |     3 |
|   Jonathan |    J |     3 |
|    Leonard |    L |     1 |
|       Mary |    M |     1 |
|     Tracey |    T |     1 |
|        Zoe |    Z |     1 |
```

Window functions SQLFiddle

Window functions vs. aggregate functions

In standards-compliant SQL databases, every aggregate function (even user-defined aggregate functions) can be turned into a window function by adding the `OVER()` clause. Such a function can then be used without any `GROUP BY` clause and without any other constraints imposed on aggregate functions. Instead, however, window functions can only be used in `SELECT` or `ORDER BY` clauses, as they operate on a materialised table source.

In addition to aggregate functions turned into window functions, there are also a variety of ranking functions and analytical functions, which are only available *with* an `OVER()`clause.

Your best choice is to start up your CUBRID, DB2, Oracle, PostgreSQL, SQL Server, or Sybase SQL Anywhere database, and start playing around with window functions right away!