Window function

What are Window functions ?

> A **window function** (also referred to as **Analytic Function**) performs a calculation across a set of table rows that are somehow related to the current row. ... However, window functions do not cause rows to become grouped into a single output row (not like `group by`). Instead, the rows *retain their separate identities*.

The window function is being applied to more rows than just the current row of the query result.

Let's say we run a SUM() of some column on a whole table and then run a select query on the database.

The goal is not only to calculate the SUM() ...

```sql
SELECT SUM(some_column ) from some_table;
# 12
```

... but to have the result as an extra column collated to the rows returned by the query.

col 1	col 2	...	result of SUM()
abc	cde	...	12
wer	sdf	...	12
...	...	...	...
zyh	yup	...	12

Aggregate functions like MAX(), MIN(), AVG(), SUM(), COUNT() etc can also be used as window functions. And then functions such as ROW NUMBER(), RANK(), DENSE RANK(), LEAD(), LAG() are specific to window functions.

These functions are commonly used across most of the popular RDBMS such as Oracle, MySQL, PostgreSQL, Microsoft SQL Server etc.

Windows function use 2 clauses:

- `OVER()`
- `Partition By` to specify the column based on which different windows needs to be created.

start by loading the data

Attention: We work on the worldhits database in its original **denormalized** version. Just one track table. Reload the csv file into a newly created worldhitsdb database if needed.

> psql into worldhitsdb on your local

Max popularity and the OVER() clause

Say, we want to find the maximum popularity for all tracks.

SQL
```sql
select max(popularity) as max_pop from tracks t
```

Now we want to add that `max_pop` value as a column when returning all the rows.

SQL
```sql
select t.track, t.artist, t.popularity, (select max(popularity) from tr
```

Adding subqueries for extra columns is fine but makes queries hard to understand. (and is not super efficient)

We can have the exact same result by using the `OVER()` clause in the previous query

SQL
```sql
select t.track, t.artist, t.popularity,
max(popularity) OVER() as max_pop
from tracks t ;
```

Here the `OVER()` clause just creates a **window OVER the whole table** and applies the MAX() function OVER that window.

Max Popularity by Artist

Now we want to find the max popularity for each artist. Instead of just returning a 2 column results with max(pop) and artist name that we get with a `GROUP BY`

SQL
```sql
select max(popularity) as max_pop, artist from tracks group by artist
```

max_pop	artist
36	Afro Celt Sound System
46	Al Di Meola
53	Ali Farka Touré

... we want to return more information per row in the db and add the extra `max_pop` column calculated for each artist.

We want to have:

id	artist	track	...	max_pop
301	Salif Keita	Sina	...	32
302	Salif Keita	Tekere	...	32
302	Salif Keita	something else	...	32
203	Jónsi	Tornado	...	30
201	Jónsi	Kolniður	...	30

Partition BY

This is where `partition by` comes in:

> `partition by <column name>` specifies the window, the subset of rows, over which to apply the `max()` function.

This query add `max_pop` by artist column to the list of tracks.

```sql
select t.track, t.artist, t.popularity,
max(popularity) over(partition by artist) as max_pop
from tracks t ;
```

In short, the OVER() clause specifies to SQL that you need to create a *a subset of records*

- without specifying a column in the OVER clause, SQL creates one window function over all the records
- the partition by clause indicates what column you want to group the subsets on

In the above query,

- `OVER(PARTITION BY artist)` creates one window for all tracks (rows) for each given artist
- and calculate the `MAX(popularity)` in each window

General case

The general query is :

```sql
SELECT <list of columns>,
SOME_FCT(<main_column_name>) OVER(PARTITION BY <other_column_name> ) AS
FROM <table> ;
```

This works with other aggregate functions such as: max(), min(), avg(), count(), sum()

Stats

So we can combine them to get some stats for each artist.

```sql
select t.track, t.artist, t.popularity,
max(popularity) over(partition by artist) as max_pop,
min(popularity) over(partition by artist) as min_pop,
avg(popularity) over(partition by artist) as avg_pop
from tracks t ;
```

functions specific to window functions

There are 5 functions that are specific to window functions.

They are used to

- number the in-window rows sequentially : `ROW_NUMBER(), RANK, DENSE RANK()`
- get a column value from a previous or next row: `LAG(), LEAD()`

ROW_NUMBER()

`ROW_NUMBER()` assigns a unique value to each records in each window

Without specifying a partition, we get an index over all the records

```sql
select t.id,  t.artist, t.track,
row_number() over() as rn
from tracks t limit 10;
```

`rn` is exactly the same as the current `id` since `id` is sequential without gaps.

Note that, if the id, the primary key, was not sequential, you could use that to generate a new primary key that is sequential.

More interestingly if we want to restart the row number per artist, we add `PARTITION BY artist` in the `OVER` clause:

```SQL
select t.id, t.artist, t.track,
row_number() over(partition by artist) as rn
from tracks t
order by artist asc
limit 20;
```

The `rn` is reset to 1 for each artist.

When is that useful ?

When ordering the rows by some column, you can fetch the top or bottom artists.

For instance, let's fetch

> the top 2 tracks with the highest tempo from each artist

by:

- adding an `order on tempo` in the `OVER` clause
- putting these results in a subquery
- and adding a filter on `rn`

1. add order by tempo desc in the OVER() clause

```SQL
select t.id, t.artist, t.track, t.tempo,
row_number() OVER(partition by artist order by tempo desc) as rn
from tracks t
order by artist asc;
```

1. then use that as a subquery (called subset) and filter

```SQL
select * from (
    select t.id, t.artist, t.track, t.tempo,
    row_number() over(partition by artist order by tempo desc) as rn
    from tracks t
    order by artist asc
    ) subset
where subset.rn < 3;
```

RANK()

To fetch the most popular tracks each year and number the tracks by popularity per year :

```SQL
select * from (
    select t.id, t.artist, t.track, t.year, t.popularity,
    row_number() over(partition by year order by popularity desc) as rn
    from tracks t) X
where X.rn < 3;
```

but some tracks have the same popoularity score andd yet have different row numbers.

Using rank() we can assign equal row numbers to tracks with equal popularity scores.

```SQL
select t.id, t.artist, t.track, t.year, t.popularity,
rank() over(partition by year order by popularity desc) as rnk
from tracks t;
```

Equal values will have the same rank (see year 2005);

so rank can be 1, 2, 2, 4

so top 2 most popular tracks per year

```SQL
select * from (
    select t.id, t.artist, t.track, t.year, t.popularity,
    rank() over(partition by year order by popularity desc) as rnk
    from tracks t
) pop
where pop.rnk < 2;
```

1985 and 2005 have 2 tracks ex-aequo for most popular.

dense_rank()

`dense_rank()` similar to rank but will increment the rank without jumps in numbers

We can compare row*number(), rank() and dense*rank() in the following query

SQL
```sql
select t.id, t.artist, t.track, t.year, t.popularity,
row_number() over(partition by year order by popularity desc) as rn,
rank() over(partition by year order by popularity desc) as rnk,
dense_rank() over(partition by year order by popularity desc) as drnk
from tracks t
where t.year in (1985, 2005);
```

And now a release date

To illustrate LAG and LEAD, we need an album release date column.

We could use `id` as proxy but ... meh :(;

Let's create a fake album release date using:

- the row_number by year as the day
- and the month as random
- and the existing year

First create or fetch the day, month, year values for each track

SQL
```sql
select t.id, t.track,
    t.year as rel_year,
    floor(random() * 12 + 1) as rel_month,
    row_number() over ( partition by t.year) as rel_day
from tracks t
order by album;
```

We need to add the release_date column as `DATE` data type to the tracks table

see https://www.postgresql.org/docs/current/datatype-datetime.html

SQL
```sql
ALTER TABLE tracks add column release_date DATE ;
```

and now :

- concatenate `rel_year`, `rel_month`, `rel_day` into a date string
- cast it as a date using the above results as a subquery.

We use the PostgreSQL `MAKE_DATE()` function to transform a set of strings (year, month, day) .

By the way, to get info on a function in PostgreSQL you can `\df function_name` :

```SQL
\df make_date
```

returns

```Bash
                                    List of functions
   Schema    |    Name     | Result data type |            Argument data typ
-------------+-------------+------------------+-----------------------------
 pg_catalog | make_date | date             | day integer, month integer
```

Notice that the function accepts only **INTs** as input values.

So we need to cast the year, month, day values as INTs

Then we use a CTE to *define* the subquery

```SQL
WITH subquery AS (
    SELECT t.id, t.track,
           t.year as rel_year,
           floor(random() * 10 + 1)::int as rel_month,
           row_number() over (partition by t.year) as rel_day
    FROM tracks t
    ORDER BY album
)

UPDATE tracks
SET release_date = MAKE_DATE(rel_year::int, rel_month::int, rel_day::int
FROM subquery
WHERE tracks.id = sq.id;
```

If you don't cast the rel*day, rel*month, ... as int.

you get

```Bash
ERROR:  42883: function make_date(integer, double precision, bigint) doe
```

LEAD() and LAG()

Now that we have a release date for each track we can demo lead() and lag()

We want to find if the popularity of a track is higher, lower or equal than the previously released track year after year.

start with getting the popularity of tracks year by year

```sql
select t.id, t.track, t.release_date, t.popularity,
lag(popularity) over(partition by year order by release_date) as prev_tr
from tracks t
where year BETWEEN 1964 and  1970 ;
```

We see null values when the album was first that year and the previous row popularity in the column `prev_track_pop`

> Note: if we don't partition by year, but just use `OVER(order by releaase_date)`, we have just one NULL value for the first row.

Now we can find out if the popularity was higher, lower or equal than the previously released track.

```sql
select t.id, t.track, t.release_date, t.popularity,
case when t.popularity > lag(popularity) over(order by release_date) the
when t.popularity < lag(popularity) over(order by release_date) then 'lo
when t.popularity = lag(popularity) over(order by release_date) then 'eq
else 'unknown'
END as pop_delta
from tracks t
where year BETWEEN 1964 and  1970;
```

`LAG()` and `LEAD()` take arguments:

- LAG(column)
- LAG(column, N rows)
- LAG(column, N rows, default value)

so lag(popularity, 2, 0) looks 2 rows before the currrent row and 0 is a default value

You can also set a column name as the default value. For instance

- LAG(popularity, 1, popularity)

Then the default value will be the current row popularity

`LEAD()` is the same as `LAG()` except that it gets the value of the row(s) following the current record.

Further readings

see https://www.postgresql.org/docs/current/tutorial-window.html