Datetime and Window Functions

1 Date/Time Functions

Dates and times are useful when calculating changes or data over time, like weather trends or stock price movements.

1.1 CAST

`CAST` converts between datatypes.

```
CAST(<expression> AS <datatype>)
```

Example: applying `CAST` to date and time:

```
SELECT
    NOW(),
    CAST(NOW() AS TIMESTAMP),
    CAST(NOW() AS DATE),
    CAST(NOW() AS TIME),
    CURRENT_DATE,
    CURRENT_TIME
```

This will give the following output in Postgres:

- 2020-11-17 20:10:11.54299-06
- 2020-11-17 20:10:11.54299
- 2020-11-17
- 20:10:11.54299
- 2020-11-17
- 20:10:11.542990-06:00

The `NOW()` function returns a timestamp that also includes the timezone information. By performing a `CAST` to it we can either get a `TIMESTAMP` type without the timezone information, a `DATE` type in YYYY-MM-DD format, or a `TIME` type in HH:MM:SS format.

1.2 INTERVAL

`INTERVAL` can be used to add or substract time:

```
SELECT
    CAST(NOW() AS DATE) AS TODAY,
    CAST((INTERVAL '3 DAYS' + NOW()) AS DATE) AS three_days,
    CAST((NOW()- INTERVAL '3 WEEKS') AS DATE) AS three_weeks,
    CAST((INTERVAL '3 MONTHS' + NOW()) AS DATE) AS three_months,
    CAST((INTERVAL '3 YEARS' + NOW()) AS DATE) AS three_years
```

Output:

- 2020-11-17
- 2020-11-20
- 2020-10-27
- 2021-02-17
- 2023-11-17

1.3 EXTRACT

`EXTRACT(part FROM date)`

We can use `EXTRACT` to pull *part* from the source *date* we want to extract from.

```
SELECT
    EXTRACT(MINUTE FROM NOW()) AS MINUTE,
    EXTRACT(HOUR FROM NOW()) AS HOUR,
    EXTRACT(DAY FROM NOW()) AS DAY,
    EXTRACT(WEEK FROM NOW()) AS WEEK,
    EXTRACT(MONTH FROM NOW()) AS MONTH,
    EXTRACT(YEAR FROM NOW()) AS YEAR,
    EXTRACT(DOW FROM NOW()) AS DAY_OF_WEEK,
    EXTRACT(DOY FROM NOW()) AS DAY_OF_YEAR,
    EXTRACT(QUARTER FROM NOW()) AS QUARTER,
```

Output:

minute double precision	hour double precision	day double precision	week double precision	month double precision	year double precision	day_of_week double precision	day_of_year double precision	quarter double precision
49	20	17	47	11	2020	2	322	4

1.4 DATE_TRUNC

`DATE_TRUNC` can be used for rounding date and time values in *datetime-column-name* to a precision specified by *interval* in terms of years, months, days, etc.

`DATE_TRUNC(<interval>, <datetime-column-name>)`

```
SELECT
    NOW() AS NOW,
    CAST(DATE_TRUNC('MINUTE', NOW()) AS TIME) AS MINUTE,
    CAST(DATE_TRUNC('DAY', NOW()) AS DATE) AS DAY,
    CAST(DATE_TRUNC('WEEK', NOW()) AS DATE) AS WEEK,
    CAST(DATE_TRUNC('MONTH', NOW()) AS DATE) AS MONTH,
    CAST(DATE_TRUNC('YEAR', NOW()) AS DATE) AS YEAR
```

Output:

now timestamp with time zone	minute time without time zone	day date	week date	month date	year date
2020-11-17 21:00:38.787042-06	21:00:00	2020-11-17	2020-11-16	2020-11-01	2020-01-01

Think of `DATE_TRUNC` as cutting a timeline into chunks of intervals and determining which interval the input date-time currently lies.

2 Window Functions

Window functions help to run operations on a selection of rows and return a value from the results of another original query. The term *window* is used to describe the set of rows on which the function operates. A window function uses rows in a window to calculate the result of the query. For instance, we can use window functions to calculate cumulative sums, moving averages, ranking results within a window (e.g. per-group ranking), etc.

Window functions are initiated with the `OVER` clause:

```
function (expression) OVER
    ( PARTITION BY <expression-list>
    ORDER BY <order-list>
    ROWS <frame-clause>)
```

and are configured using:

- window partition `PARTITION BY`: group rows into partitions
- window ordering `ORDER BY`: defines the order or sequence of rows within each window
- window frame `ROWS`: defines the window frames by an offset from a specified row

Vs. `GROUP BY`:

- Window functions do not reduce the number of rows in the output.
- Window functions can obtain values from other rows while `GROUP BY` functions are limited to their own groups.

2.1 ROW_NUMBER()

`ROW_NUMBER()` assigns a number to each row in a table.

Suppose we have a table: summer_medals(year, city, sport, discipline, athlete, country, gender, event, medal) which records all Olympic medalists.

```
SELECT athlete, event, ROW_NUMBER() OVER() AS Row_Number
FROM summer_medals
ORDER BY Row_Number ASC;
```

	athlete	event	Row_Number
1	K�HLER, Christa	3m springboard	1
2	KOSENKOV, Aleksandr	3m springboard	2
3	BOGGS, Philip George	3m springboard	3
4	CAGNOTTO, Giorgio Franco	3m springboard	4
5	WILSON, Deborah Keplar	10m platform	5
6	LOUGANIS, Gregory	10m platform	6
7	VAYTSEKHOVSKAYA, Elena	10m platform	7
8	POTTER-MCINGVALE, Cynthia	3m springboard	8
9	DIBIASI, Klaus	10m platform	9
10	ALEINIK, Vladimir	10m platform	10
11	KNAPE-LINDBERGH, Ulrika	10m platform	11

Essentially we used the `ROW_NUMBER()` function to create and assign a row number to the selected variables. We also alias the window function as *Row_Number* so that we can perform an `ORDER BY` operation.

If we want to assign a row number to every sport but also make sure that the row numbers are assigned in alphabetical order:

```
SELECT sport, ROW_NUMBER() OVER(ORDER BY sport ASC) AS Row_N
FROM (SELECT DISTINCT sport
      FROM summer_medals) AS sports
ORDER BY sport ASC;
```

	sport	Row_N
1	Aquatics	1
2	Archery	2
3	Athletics	3
4	Badminton	4
5	Baseball	5
6	Basketball	6
7	Boxing	7
8	Canoe / Kayak	8
9	Cycling	9
10	Equestrian	10
11	Fencing	11

Note that without the `ORDER BY` statement inside `OVER()`, the row numbers are not assigned correctly.

2.2 PARTITION BY

We can use `PARTITION BY` to split the table into partitions based on unique values from a column. This is useful when we want to perform calculations on individual rows of a group using data from other rows of that group. In addition to specific column names, we can also partition by window functions.

2.2.1 LAG()

`LAG()` is a window function that outputs a row before the current row. Suppose we want to find players separated by gender, who won the gold medal in singles for tennis from 1996 onwards but also include in each row the winner for the previous Olympics.

```
-- CTE for isolating gold medal tennis winners
WITH Tennis_Gold AS (
    SELECT athlete, gender, year
    FROM summer_medals
    WHERE year >= 1996 AND sport = 'Tennis' AND event = 'singles' AND
       medal = 'Gold')
-- Window Function to find the lag and partition by gender
SELECT athlete as Champion, gender, year,
LAG(athlete) OVER (PARTITION BY gender ORDER BY year ASC) AS
   Last_Champion
FROM Tennis_Gold
```

```
ORDER BY gender ASC, year ASC;
```

	Champion	gender	year	Last_Champion
1	AGASSI, Andre	Men	1996	*NULL*
2	KAFELNIKOV, Eugueni	Men	2000	AGASSI, Andre
3	MASSU, Nicolas	Men	2004	KAFELNIKOV, Eugueni
4	NADAL, Rafael	Men	2008	MASSU, Nicolas
5	DAVENPORT, Lindsay	Women	1996	*NULL*
6	WILLIAMS, Venus	Women	2000	DAVENPORT, Lindsay
7	HENIN-HARDENNE, Justine	Women	2004	WILLIAMS, Venus
8	DEMENTIEVA, Elena	Women	2008	HENIN-HARDENNE, Justine

Note that the scope of our `LAG()` window function is limited to each of the partitions defined by the `PARTITION BY` statement, here the Men and Women genders.

2.2.2 LEAD()

`LEAD()` is the opposite of `LAG()`, returning the row after the current row.

2.2.3 RANK()

`RANK()` is similar to `ROW_NUMBER()` but it assigns the same number to rows with identical values, while skipping over the next ranking. `DENSE_RANK` does not skip over the next ranking number.

Suppose we want to rank the 5 countries (Argentina, South Korea, France, Italy, Kazakhstan) by the number of Olympics in which they have won at least a medal.

```
WITH countries AS (
    SELECT country, COUNT(DISTINCT year) AS participated
    FROM summer_medals
    WHERE country in ('Argentina', 'Korea, South', 'France', 'Italy','
       Kazakhstan')
    GROUP BY country)
SELECT country, participated,
    ROW_NUMBER()
        OVER(ORDER BY participated DESC) AS Row_Number,
    RANK()
        OVER(ORDER BY participated DESC) AS Rank,
    DENSE_RANK()
        OVER(ORDER BY participated DESC) AS Dense_Rank
    FROM countries
    ORDER BY participated DESC;
```

	country	participated	Row_Number	Rank	Dense_Rank
1	France	9	1	1	1
2	Italy	9	2	1	1
3	Korea, South	8	3	3	2
4	Argentina	6	4	4	3
5	Kazakhstan	4	5	5	4

2.3 Window Frames

Syntax of the frame-clause:

```
mode BETWEEN frame_start AND frame_end [frame_exclusion]
```

This allows us to divide partitions into smaller subsets that are specific to each row. This allows us to flexibly narrow down the set of rows being used for our window function calculation.

`mode` defines the way we treat input rows.

- `ROWS` mode allows us to treat each row separately as individual entities. We can then define `frame_start` and `frame_end` to specify which rows the window fram starts and ends with.
- `GROUPS` mode can be useful when the column we are sorting on has duplicates, where we group rows according to the column specified by `ORDER BY`
- `RANGE` mode does not tie rows together in any way. Instead the rows in our window frame are entirely determined by the values of the sorting column

`frame_start` and `frame_end`:

- `UNBOUND PRECEDING`: start with the first row of the partition
- `offset PRECEDING`: start/end with either 1) the given number of rows *before* the current row in `ROWS` mode, 2) the given number of groups *before* the group that the current row is in, in `GROUPS` mode, or 3) the row which has a difference in values specified by `offset` w.r.t. the current row's values in `RANGE` mode
- `CURRENT ROW`: start/end with the current row
- `offset FOLLOWING`: start/end with either 1) the given number of rows *after* the current row in `ROWS` mode, 2) the given number of groups *after* the group that the current row is in, in `GROUPS` mode, or 3) the row which has a difference in values specified by `offset` w.r.t. the current row's values in `RANGE` mode
- `UNBOUND FOLLOWING`: end with the last row of the partition

`frame_exclusion` allows us to exclude some specific rows from the window frame, even if they would have been included according to how we defined the `frame_start` and `frame_end` options. `frame_exclusion` works in the same way regardless of the selected mode. The possible values are:

- `EXCLUDE CURRENT ROW`: exclude the current row only
- `EXCLUDE GROUP`: exclude the current row and all peer rows (rows that have the same value in the sorting column)
- `EXCLUDE TIES`: exclude all peer rows, but not the current row
- `EXCLUDE NO OTHERS`: excludes nothing. This is the default option if `frame_exlusion` is omitted.

It's important to remember that window frames are constructed for every single input row separately, so the frames for each row can be different from row to row. Therefore it is essential to consider a window frame with respect to the row that the frame is built for.