75.15 / 75.28 / 95.05 - Base de Datos

SQL Avanzado:
WITH Recursivo y Funciones de Ventana

Mariano Beiró

Dpto. de Computación - Facultad de Ingeniería (UBA)

11 de abril de 2023

Temas

1 WITH Recursivo

2 Funciones de Ventana
- Ventanas bajo una única partición
- Múltiples particiones

3 Bibliografía

1 WITH Recursivo

2 Funciones de Ventana
- Ventanas bajo una única partición
- Múltiples particiones

3 Bibliografía

hacemos recursión hasta que una tabla (dada una tabla) sea igual a esta última

se llama

PUNTO FIJO

`WITH` recursivo (opcional según estándar)

- La cláusula **`WITH RECURSIVE`** amplia el poder expresivo de SQL permitiendo encontrar la clausura transitiva de una consulta.
- Dada una tabla T que es *input* de una consulta, permite que el resultado de la misma, $T_{new} \leftarrow subquery(T)$, sea utilizado en el lugar de T para volver a ejecutar la misma consulta.
  - Esto se repite hasta encontrar un punto fijo, i.e., $T = subquery(T)$.

WITH RECURSIVE $T[(A_1, A_2, ..., A_n)]$
AS (*<initial_value_query>*
UNION *<subquery>*)
<query with T>;

WITH recursivo 1: Pseudocódigo

$T_0 = initial\_value\_query(R_1, R_2, ...)$;
$T_{new} = T_0$;
do
  $T = T_{new}$;
  $T_{new} = T_0 \cup subquery(T, R_1, R_2, ...)$;
while $T_{new} \neq T$;
return $query(T, ...)$;

- La consulta *<initial_value_query>* no puede depender de T.
- Tanto en **`WITH`** como en **`WITH RECURSIVE`** puede definirse más de una tabla auxiliar antes de la consulta.

WITH recursivo (opcional según estándar)

Ejemplo

Dada la relación Vuelos(codVuelo, ciudadDesde, ciudadHasta) que indica todos los vuelos que ofrece una aerolínea, encuentre todas las ciudades que son 'alcanzables' desde París, independientemente de la cantidad de escalas que sea necesario hacer.

```
WITH RECURSIVE DestinosAlcanzables(ciudad)
AS (VALUES ('Paris')
    UNION
    SELECT v.ciudadHasta AS ciudad
    FROM DestinosAlcanzables d, Vuelos v
    WHERE d.ciudad = v.ciudadDesde
)
SELECT ciudad FROM DestinosAlcanzables;
```

1 WITH Recursivo

2 Funciones de Ventana
- Ventanas bajo una única partición
- Múltiples particiones

3 Bibliografía

Introducción

- Las funciones de ventana (window functions) permiten aplicar un procesamiento final a los resultados de una consulta, siguiendo estos pasos:
  1. ... dividiéndolos en grupos, llamados particiones,
  2. ... ordenando internamente cada partición,
  3. ... y cruzando información entre las filas de cada partición.
- A cada atributo del `SELECT` de una consulta se le puede aplicar una función de ventana distinta, o bien puede no aplicarsele función alguna.

Única partición

- Veamos inicialmente el caso en que se considera a todo el resultado como una única partición (no se utiliza la palabra clave `PARTITION`):
- El esquema básico para esta situación es:

SELECT .., [$f(A_i)$ | $w([A_i], ..)$] **OVER** ({ **ORDER BY** A_j [ASC | DESC] }), ..
FROM ...

- $\rightarrow$ Esto ordena el resultado de la consulta por el atributo A_j y para cada fila imprime el resultado de una función de agregación, $f(A_i)$, ó de una función de ventana, $w([A_i], ..)$.

Única partición

SELECT .., [$f(A_i)$ | $w([A_i], ..)$] **OVER** ({ **ORDER BY** A_j [ASC | DESC] }), ..
FROM ...

- Hasta acá, $f(A_i)$ no parece nada muy novedoso y que no podamos hacer con un `GROUP BY`.
- Sin embargo, bajo este esquema también podemos usar funciones de ventana, $w([A_i], ..)$, como[1] :
  1. `RANK()`: Nos devuelve el ranking de cada fila, de acuerdo al valor del atributo de ordenamiento de la partición.
  2. `ROW_NUMBER()`: Nos devuelve el número de orden de la fila en la partición.
  3. `LAG(`A_i`, offset)`: Nos devuelve el valor que toma el atributo A_i en una fila a distancia *offset* hacia atrás de la actual.

Otras funciones de ventana se encuentran definidas en `https://www.postgresql.org/docs/9.1/functions-window.html`.

Única partición

Ejemplo

- Supongamos que en la siguiente tabla de la final de 100m llanos queremos ordenar a los atletas de acuerdo con sus tiempos e indicar su posición final en una columna:

FINAL_2009

nombre_atleta	país_origen	marca_seg
Daniel Bailey	Antigua y Barbuda	9,93
Tyson Gay	Estados Unidos	9,71
Marc Burns	Trinidad y Tobago	10,00
Usain Bolt	Jamaica	9,58
Darvis Patton	Estados Unidos	10,34
Asafa Powell	Jamaica	9,84
Richard Thompson	Trinidad y Tobago	9,93
Dwain Chambers	Reino Unido	10,00

```
SELECT RANK() OVER (ORDER BY marca_seg) AS posición,
       nombre_atleta, país_origen, marca_seg
FROM Final_2009;
```

Única partición

Ejemplo

El resultado será:

posición	nombre_atleta	país_origen	marca_seg
1	Usain Bolt	Jamaica	9,58
2	Tyson Gay	Estados Unidos	9,71
3	Asafa Powell	Jamaica	9,84
4	Daniel Bailey	Antigua y Barbuda	9,93
4	Richard Thompson	Trinidad y Tobago	9,93
6	Dwain Chambers	Reino Unido	10,00
6	Marc Burns	Trinidad y Tobago	10,00
8	Darvis Patton	Estados Unidos	10,34

si acá queremos un 5

podemos utilizar

`DENSE_RANK()`

Única partición

Ejemplo

- También podríamos devolver la diferencia de tiempo que cada atleta tuvo con el que llegó antes que él:

```
SELECT RANK() OVER (ORDER BY marca_seg) AS posición,
       nombre_atleta, país_origen, marca_seg,
       marca_seg - LAG(marca_seg, 1) OVER (ORDER BY marca_seg) AS diferencia
FROM Final_2009;
```

Diff entre este y su siguiente

FINAL_2009

posición	nombre_atleta	país_origen	marca_seg	diferencia
1	Usain Bolt	Jamaica	9,58	
2	Tyson Gay	Estados Unidos	9,71	0,13
3	Asafa Powell	Jamaica	9,84	0,13
4	Daniel Bailey	Antigua y Barbuda	9,93	0,09
4	Richard Thompson	Trinidad y Tobago	9,93	0,00
6	Dwain Chambers	Reino Unido	10,00	0,07
6	Marc Burns	Trinidad y Tobago	10,00	0,00
8	Darvis Patton	Estados Unidos	10,34	0,34

Única partición

Observaciones

- Para cada columna a la que queremos aplicar una función de ventana debemos repetir la estructura `OVER (ORDER BY ...)`, a la que llamamos ventana.
- A diferencia del `GROUP BY`, el `OVER (ORDER BY ...)` no agrupa, con lo cual no cambiará la cantidad de filas en el resultado.
- La función de ventana se aplica **antes** del ordenamiento que pueda hacerse en el cláusula `ORDER BY` al final de la consulta. En otras palabras, podríamos utilizar el `ORDER BY` final de la consulta para reordenar nuevamente los resultados.
  - Ejemplo: la siguiente consulta reordena a los atletas por su diferencia en forma decreciente, pero sin alterar los rankings:

```
SELECT RANK() OVER (ORDER BY marca_seg) AS posición,
    nombre_atleta, país_origen, marca_seg,
    marca_seg - LAG(marca_seg, 1) OVER (ORDER BY marca_seg) AS diferencia
FROM Final_2009
ORDER BY diferencia DESC;
```

Única partición

Otras funciones de agregación. Frames

- Si bien el `ORDER BY` dentro del `OVER()` es opcional, al no indicarlo los datos podrán llegar en cualquier orden, lo que no suele ser el comportamiento deseado.
- Además, cuando usamos funciones de agregación en la columna (`SUM()`, `COUNT()`, ...), el `ORDER BY` de dentro del `OVER()` tiene un comportamiento más disruptivo:
  - Si se emplea `OVER()` a secas, entonces la función de agregación se calcula sobre toda la partición, con lo que no difiere de lo que haría un `GROUP BY`, salvo porque nos repite el valor agregado en todas las filas.
  - Si se emplea `OVER(ORDER BY...)`, entonces la función de agregación en cada fila se calcula sobre un *frame* (marco) que se extiende desde la primera fila de la partición ordenada hasta la actual. Es decir que los resultados devueltos son resultados acumulados (parciales) dentro de la partición.

Única partición

Definición de ventanas

- Si vamos a utilizar una misma ventana muchas veces en la consulta, podemos definirla una única vez con la cláusula **WINDOW**.
- Por ejemplo:

```
SELECT RANK() OVER ventana_marca_seg AS posición,
    nombre_atleta, país_origen, marca_seg,
    marca_seg - LAG(marca_seg, 1) OVER ventana_marca_seg AS diferencia
FROM Final_2009
WINDOW ventana_marca_seg AS (ORDER BY marca_seg)
ORDER BY diferencia DESC;
```

Múltiples particiones

- En el esquema con multiples particiones, antes de aplicar cada ventana se puede agrupar el resultado por el valor de un conjunto de atributos.
- A cada uno de estos grupos (conjunto de filas del resultado) se lo denomina partición.
- El esquema general en este caso es:

SELECT .., [$f(A_i)$ | $w([A_i], ..)$] **OVER** (**PARTITION BY** A_k
{ **ORDER BY** A_j [ASC | DESC] }), ..
FROM ...

Múltiples particiones

Ejemplo 1

- Supongamos que queremos rankear a los países de acuerdo a quiénes son los que más exportan de cada producto, indicando para cada producto cuál es el país que ocupa cada lugar del ranking, y ordenando el resultado final por producto, ranking y país.

EXPORTACIONES

país	producto	cantidad
Brasil	Trigo	720
Argentina	Trigo	440
USA	Maíz	520
Australia	Trigo	900
China	Sorgo	80
Argentina	Maíz	520
China	Trigo	780

Múltiples particiones

Ejemplo 1

```
SELECT producto,
       RANK() OVER (PARTITION BY producto ORDER BY cantidad DESC) AS posición,
       país, cantidad
FROM Exportaciones
ORDER BY producto, posición;
```

EXPORTACIONES

producto	posición	país	cantidad
Maíz	1	Argentina	520
Maíz	1	USA	520
Sorgo	1	China	80
Trigo	1	Australia	900
Trigo	2	China	780
Trigo	3	Brasil	720
Trigo	4	Argentina	440

Múltiples particiones

Ejemplo 2

- Veamos ahora que pasa si queremos utilizar una función de agregación en vez de una función de ventana.
- Disponemos de todas las operaciones realizadas por los clientes de un banco, ordenadas por fecha, incluyendo el saldo inicial de cada uno:

OPERACIONES

fecha	cliente	concepto	monto
2020-04-01	003	Saldo inicial	$10.000
2020-04-01	004	Saldo inicial	$5.000
2020-04-02	003	Depósito en efectivo	$2.000
2020-04-15	004	Farmacia	-$2.100
2020-04-18	003	Adidas	-$3.500
2020-04-23	003	Transferencia a Jorge Mussi	-$2.000
2020-04-28	004	Transferencia de Emilce Vega	$2.800

- Quisieramos generar un resumen ordenado por cliente y fecha en donde además de las operaciones realizadas se indique el saldo del cliente en todo momento.

Múltiples particiones

Ejemplo 2

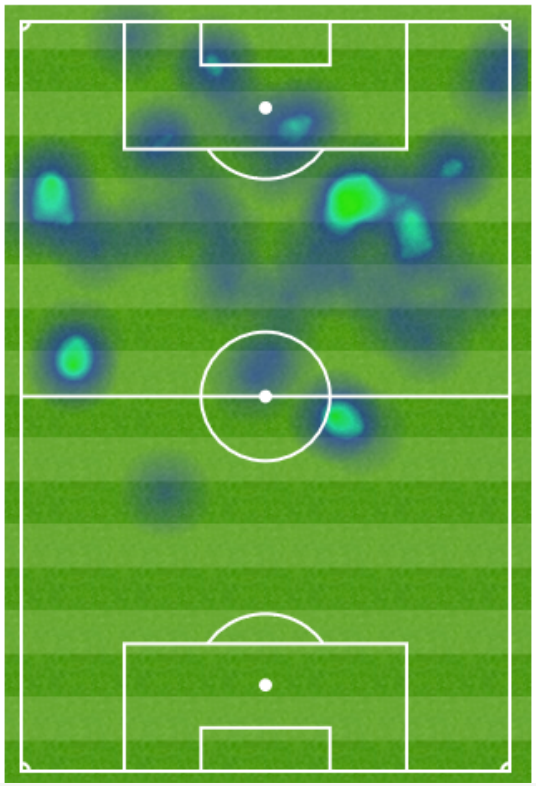
```
SELECT cliente, fecha, concepto, monto,
       SUM(monto) OVER (PARTITION BY cliente ORDER BY fecha) AS saldo
FROM Operaciones
ORDER BY cliente, fecha;
```

cliente	fecha	concepto	monto	saldo
003	2020-04-01	Saldo inicial	$10.000	$10.000
003	2020-04-02	Depósito en efectivo	$2.000	$12.000
003	2020-04-18	Adidas	-$3.500	$8.500
003	2020-04-23	Transferencia a Jorge Mussi	-$2.000	$6.500
004	2020-04-01	Saldo inicial	$5.000	$5.000
004	2020-04-15	Farmacia	-$2.100	$2.900
004	2020-04-28	Transferencia de Emilce Vega	$2.800	$5.700

Observemos que la función `SUM()` utiliza en cada fila un marco *(frame)* que va desde el inicio del mes de ese cliente hasta la fecha actual, calculando así las “sumas parciales”.

Múltiples particiones

Ejercicio



Supongamos que en una tabla tenemos el listado de toques de balón durante un partido de fútbol. Cada entrada de esta tabla indica el instante en que un futbolista toca el balón.

TOQUES

timestamp	jugador
16:21:18.418	Lionel Messi
16:21:22.637	Giovanni Lo Celso
16:21:42.402	Dani Alves
16:21:46.535	Nicolás Otamendi
16:21:49.388	Lionel Messi
...	...

Queremos encontrar al jugador que pasó más tiempo sin tocar el balón, indicando su nombre y cuál fue dicho tiempo.

Múltiples particiones

Solución

```
WITH Tiempos_Sin_Balon AS
    (SELECT jugador,
        timestamp - LAG(timestamp, 1)
        OVER (PARTITION BY jugador ORDER BY timestamp ASC) AS tiempo_sin_balón
    FROM Toques)
SELECT DISTINCT t1.jugador, t1.tiempo_sin_balón
FROM Tiempos_Sin_Balon t1
WHERE t1.tiempo_sin_balón = (SELECT MAX(t2.tiempo_sin_balón)
                             FROM Tiempos_Sin_Balon t2);
```

1 WITH Recursivo

2 Funciones de Ventana
- Ventanas bajo una única partición
- Múltiples particiones

3 Bibliografía

Bibliografía

7.8.2. WITH Recursive

PostgreSQL Documentation

https://www.postgresql.org/docs/current/queries-with.html#QUERIES-WITH-RECURSIVE

3.5. Window Functions

PostgreSQL Documentation

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