Funciones analíticas

by admin - sábado, octubre 03, 2020

https://dbandtech.com/funciones-analiticas/

Introducidas en Oracle 8i, las funciones analíticas, también conocidas como funciones de ventanas, permiten a los desarrolladores realizar tareas en SQL que antes se limitaban a lenguajes de procedimiento.

- **Preparar**
- **Introduccíon**
- **Sintaxis Funcion Analiytica**
- **query_partition_clause**
- **order_by_clause**
- **windowing_clause**
- **Usando Funciones Analiticas**

Preparar

Los ejemplos de este artículo requieren la siguiente tabla.

```
--DROP TABLE emp PURGE;

CREATE TABLE emp (
  empno    NUMBER(4) CONSTRAINT pk_emp PRIMARY KEY,
  ename    VARCHAR2(10),
  job      VARCHAR2(9),
  mgr      NUMBER(4),
  hiredate DATE,
  sal      NUMBER(7,2),
  comm     NUMBER(7,2),
  deptno   NUMBER(2)
);

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,to_date('17-12-1980'
,'dd-mm-yyyy'),800,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,to_date('20-2-198
1','dd-mm-yyyy'),1600,300,30);
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,to_date('22-2-1981
','dd-mm-yyyy'),1250,500,30);
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,to_date('2-4-1981'
,'dd-mm-yyyy'),2975,NULL,20);
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,to_date('28-9-19
81','dd-mm-yyyy'),1250,1400,30);
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,to_date('1-5-1981'
```

```
,'dd-mm-yyyy'),2850,NULL,30);
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,to_date('9-6-1981'
,'dd-mm-yyyy'),2450,NULL,10);
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,to_date('13-JUL-87
','dd-mm-rr')-85,3000,NULL,20);
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,to_date('17-11-19
81','dd-mm-yyyy'),5000,NULL,10);
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,to_date('8-9-198
1','dd-mm-yyyy'),1500,0,30);
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,to_date('13-JUL-87',
 'dd-mm-rr')-51,1100,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,to_date('3-12-1981',
'dd-mm-yyyy'),950,NULL,30);
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,to_date('3-12-1981'
,'dd-mm-yyyy'),3000,NULL,20);
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,to_date('23-1-1982'
,'dd-mm-yyyy'),1300,NULL,10);
COMMIT;
```

Introducción

Probablemente, la forma más fácil de comprender las funciones analíticas es comenzar por observar funciones agregadas. Una función agregada, como su nombre indica, agrega datos de varias filas en una sola fila de resultados. Por ejemplo, podríamos usar la función agregada AVG para darnos un promedio de todos los salarios de los empleados en la tabla EMP.

```
SELECT AVG(sal)
FROM   emp;

  AVG(SAL)
----------
2073.21429

SQL>
```

La cláusula **GROUP BY** nos permite aplicar funciones agregadas a subconjuntos de filas. Por ejemplo, es posible que deseemos mostrar el salario promedio de cada departamento.

```
SELECT deptno, AVG(sal)
FROM   emp
GROUP BY deptno
ORDER BY deptno;
```

```
    DEPTNO   AVG(SAL)
---------- ----------
 10 2916.66667
 20  2175
 30 1566.66667

SQL>
```

En ambos casos, la función agregada reduce el número de filas devueltas por la consulta.

Las funciones analíticas también operan en subconjuntos de filas, similar a las funciones agregadas en las consultas **GROUP BY**, pero no reducen el número de filas devueltas por la consulta. Por ejemplo, la siguiente consulta informa el salario de cada empleado, junto con el salario promedio de los empleados dentro del departamento.

```
SET PAGESIZE 50
BREAK ON deptno SKIP 1 DUPLICATES

SELECT empno, deptno, sal,
       AVG(sal) OVER (PARTITION BY deptno) AS avg_dept_sal
FROM   emp;

     EMPNO     DEPTNO        SAL AVG_DEPT_SAL
---------- ---------- ---------- ------------
      7782         10       2450   2916.66667
      7839         10       5000   2916.66667
      7934         10       1300   2916.66667

      7566         20       2975         2175
      7902         20       3000         2175
      7876         20       1100         2175
      7369         20        800         2175
      7788         20       3000         2175

      7521         30       1250   1566.66667
      7844         30       1500   1566.66667
      7499         30       1600   1566.66667
      7900         30        950   1566.66667
      7698         30       2850   1566.66667
      7654         30       1250   1566.66667

14 rows selected.
```

```
SQL>
```

Esta vez, **AVG** es una función analítica, que opera en el grupo de filas definido por el contenido de la cláusula **OVER**. Este grupo de filas se conoce como ventana, razón por la cual las funciones analíticas a veces se denominan funciones de ventana. Observe cómo la función **AVG** sigue informando el promedio departamental, como lo hizo en la consulta **GROUP BY**, pero el resultado está presente en cada fila, en lugar de reducir el número total de filas devueltas. Esto se debe a que las funciones analíticas se realizan en un conjunto de resultados después de que se completan todas las cláusulas join, **WHERE, GROUP BY y HAVING**, pero antes de que se realice la operación final **ORDER BY**.

Sintaxis de la función analítica

Existen algunas variaciones en la sintaxis de las funciones analíticas individuales, pero la sintaxis básica para una función analítica es la siguiente.

```
analytic_function([ arguments ]) OVER (analytic_clause)
```

La analytic_clause se divide en los siguientes elementos opcionales.

```
[ query_partition_clause ] [ order_by_clause [ windowing_clause ] ]
```

Los sub-elementos de analytic_clause tienen cada uno sus propios diagramas de sintaxis, que se muestran aquí. En lugar de repetir los diagramas de sintaxis, las siguientes secciones describen para qué se utiliza cada sección de la cláusula_analítica.

query_partition_clause

Query_partition_clause divide el conjunto de resultados en particiones, o grupos, de datos. El funcionamiento de la función analítica está restringido al límite impuesto por estas particiones, similar a la forma en que una cláusula **GROUP BY** afecta la acción de una función agregada. Si se omite query_partition_clause, todo el conjunto de resultados se trata como una sola partición. La siguiente consulta utiliza una cláusula **OVER** vacía, por lo que el promedio presentado se basa en todas las filas del conjunto de resultados.

```
CLEAR BREAKS

SELECT empno, deptno, sal,
       AVG(sal) OVER () AS avg_sal
FROM   emp;
```

```
     EMPNO     DEPTNO        SAL    AVG_SAL
---------- ---------- ---------- ----------
      7369         20        800 2073.21429
      7499         30       1600 2073.21429
      7521         30       1250 2073.21429
      7566         20       2975 2073.21429
      7654         30       1250 2073.21429
      7698         30       2850 2073.21429
      7782         10       2450 2073.21429
      7788         20       3000 2073.21429
      7839         10       5000 2073.21429
      7844         30       1500 2073.21429
      7876         20       1100 2073.21429
      7900         30        950 2073.21429
      7902         20       3000 2073.21429
      7934         10       1300 2073.21429

SQL>
```

Si cambiamos la cláusula **OVER** para incluir una **query_partition_clause** basada en el departamento, los promedios presentados son específicamente para el departamento al que pertenece el empleado también.

```
BREAK ON deptno SKIP 1 DUPLICATES

SELECT empno, deptno, sal,
       AVG(sal) OVER (PARTITION BY deptno) AS avg_dept_sal
FROM   emp;

     EMPNO     DEPTNO        SAL AVG_DEPT_SAL
---------- ---------- ---------- ------------
      7782         10       2450   2916.66667
      7839         10       5000   2916.66667
      7934         10       1300   2916.66667

      7566         20       2975         2175
      7902         20       3000         2175
      7876         20       1100         2175
      7369         20        800         2175
      7788         20       3000         2175

      7521         30       1250   1566.66667
      7844         30       1500   1566.66667
      7499         30       1600   1566.66667
      7900         30        950   1566.66667
```

```
      7698         30       2850   1566.66667
      7654         30       1250   1566.66667

SQL>
```

order_by_clause

Order_by_clause se utiliza para ordenar filas, o hermanos, dentro de una partición. Entonces, si una función analítica es sensible al orden de los hermanos en una partición, debe incluir un **order_by_clause**. La siguiente consulta utiliza la función **FIRST_VALUE** para devolver el primer salario informado en cada departamento. Observe que hemos dividido el conjunto de resultados por el departamento, pero no hay **order_by_clause**.

```
BREAK ON deptno SKIP 1 DUPLICATES

SELECT empno, deptno, sal,
       FIRST_VALUE(sal IGNORE NULLS) OVER (PARTITION BY deptno) AS fir
st_sal_in_dept
FROM   emp;

     EMPNO     DEPTNO        SAL FIRST_SAL_IN_DEPT
---------- ---------- ---------- -----------------
      7782         10       2450              2450
      7839         10       5000              2450
      7934         10       1300              2450

      7566         20       2975              2975
      7902         20       3000              2975
      7876         20       1100              2975
      7369         20        800              2975
      7788         20       3000              2975

      7521         30       1250              1250
      7844         30       1500              1250
      7499         30       1600              1250
      7900         30        950              1250
      7698         30       2850              1250
      7654         30       1250              1250

SQL>
```

Ahora compare los valores de la columna **FIRST_SAL_IN_DEPT** cuando incluimos un

order_by_clause para ordenar a los hermanos por salario ascendente.

```
SELECT empno, deptno, sal,
       FIRST_VALUE(sal IGNORE NULLS) OVER (PARTITION BY deptno ORDER B
Y sal ASC NULLS LAST) AS first_val_in_dept
FROM   emp;

     EMPNO     DEPTNO        SAL FIRST_VAL_IN_DEPT
---------- ---------- ---------- -----------------
      7934         10       1300              1300
      7782         10       2450              1300
      7839         10       5000              1300

      7369         20        800               800
      7876         20       1100               800
      7566         20       2975               800
      7788         20       3000               800
      7902         20       3000               800

      7900         30        950               950
      7654         30       1250               950
      7521         30       1250               950
      7844         30       1500               950
      7499         30       1600               950
      7698         30       2850               950

SQL>
```

En este caso, las palabras clave "**ASC NULLS LAST**" son innecesarias, ya que ASC es el valor predeterminado para un order_by_clause y **NULLS LAST** es el valor predeterminado para los pedidos ASC. Al hacer un pedido por DESC, el valor predeterminado es **NULLS FIRST**.

Es importante comprender cómo el **order_by_clause** afecta el orden de visualización. Se garantiza que **order_by_clause** afectará el orden de las filas a medida que son procesadas por la función analítica, pero no siempre puede afectar el orden de visualización. Como resultado, siempre debe usar una cláusula **ORDER BY** convencional en la consulta si el orden de visualización es importante. No confíe en ningún ordenamiento implícito realizado por la función analítica. Recuerde, la cláusula **ORDER BY** convencional se realiza después del procesamiento analítico, por lo que siempre tendrá prioridad.

windowing_clause

Hemos visto anteriormente que **query_partition_clause** controla la ventana, o grupo de filas, en el que opera la analítica. La Windowing_clause le da a algunas funciones analíticas un mayor grado de control sobre esta ventana dentro de la partición actual, o el conjunto de resultados completo si no se usa una

cláusula de partición. **Windowing_clause** es una extensión de **order_by_clause** y, como tal, solo se puede usar si hay un **order_by_clause** presente. La Windowing_clause tiene dos formas básicas.
RANGE BETWEEN start_point AND end_point
ROWS BETWEEN start_point AND end_point

Cuando usa **ROWS BETWEEN**, está indicando un número específico de filas en relación con la fila actual, ya sea directamente o mediante una expresión. Suponiendo que no cruza el límite de una partición, ese número de filas es fijo. Por el contrario, cuando usa **RANGE BETWEEN** se está refiriendo a un rango de valores en una columna específica en relación con el valor en la fila actual. Como resultado, Oracle no sabe cuántas filas se incluyen en el rango hasta que se crea el conjunto ordenado.
Es posible omitir la palabra clave **BETWEEN** y especificar un solo punto final **RANGE/ROWS**. En este caso, Oracle asume que su **RANGE/ROWS** especificado es el punto de inicio y el punto final es la fila actual. No recomendaría usar esta sintaxis, ya que no estará claro para cualquiera que no entienda esta acción predeterminada.

Los valores posibles para "start_point" y "end_point" son:

UNBOUNDED PRECEDING : La ventana comienza en la primera fila de la partición, o en todo el conjunto de resultados si no se utiliza una cláusula de partición. Solo disponible para puntos de inicio.
UNBOUNDED FOLLOWING : La ventana termina en la última fila de la partición, o en todo el conjunto de resultados si no se utiliza una cláusula de partición. Solo disponible para puntos finales.
CURRENT ROW : La ventana comienza o termina en la fila actual. Se puede utilizar como punto de inicio o finalización.
value_expr PRECEDING: Un desplazamiento físico o lógico antes de la fila actual que utiliza una constante o expresión que se evalúa como un valor numérico positivo. Cuando se usa con RANGE, también puede ser un literal de intervalo si order_by_clause usa una columna DATE.
value_expr FOLLOWING : Como arriba, pero un desplazamiento después de la fila actual.
El punto de inicio debe ser anterior o igual al punto final. Además, la fila actual no tiene que ser parte de la ventana. La ventana se puede definir para que comience y termine antes o después de la fila actual.
Para las funciones analíticas que admiten la windowing_clause, la acción predeterminada es **RANGE BETWEEN UNBOUNDED PRECEDING Y CURRENT ROW**. La siguiente consulta es similar a la utilizada anteriormente para informar el salario del empleado y el salario promedio del departamento, pero ahora hemos incluido un order_by_clause, por lo que también obtenemos el windowing_clause predeterminado.

```
SELECT empno, deptno, sal,
       AVG(sal) OVER (PARTITION BY deptno ORDER BY sal) AS avg_dept_sa
l_sofar
FROM   emp;

     EMPNO     DEPTNO        SAL AVG_DEPT_SAL_SOFAR
---------- ---------- ---------- ------------------
      7934         10       1300               1300
      7782         10       2450               1875
      7839         10       5000         2916.66667
```

```
      7369         20        800                800
      7876         20       1100                950
      7566         20       2975               1625
      7788         20       3000               2175
      7902         20       3000               2175

      7900         30        950                950
      7654         30       1250               1150
      7521         30       1250               1150
      7844         30       1500             1237.5
      7499         30       1600               1310
      7698         30       2850          1566.66667

SQL>
```

Hay dos cosas a tener en cuenta aquí.

• La adición de **order_by_clause** sin una **windowing_clause** significa que la consulta ahora está devolviendo un promedio móvil.
• La una windowing_clause predeterminada es **RANGE BETWEEN UNBOUNDED PRECEDING Y CURRENT ROW**, no **ROWS BETWEEN UNBOUNDED PRECEDING Y CURRENT ROW**. El hecho de que sea RANGE, no ROWS, significa que incluye todas las filas con el mismo valor que el valor en la fila actual, incluso si están más abajo en el conjunto de resultados. Como resultado, la ventana puede extenderse más allá de la fila actual, aunque no crea que este sea el caso.
Para ilustrar el último punto, veamos los valores si comparamos **RANGE y ROWS** para la última consulta. Observe las diferencias entre esas líneas en negrita.

```
SELECT empno, deptno, sal,
       AVG(sal) OVER (PARTITION BY deptno ORDER BY sal
                      RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT RO
W) AS range_avg,
       AVG(sal) OVER (PARTITION BY deptno ORDER BY sal
                      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
) AS rows_avg
FROM   emp;

     EMPNO     DEPTNO        SAL  RANGE_AVG   ROWS_AVG
---------- ---------- ---------- ---------- ----------
      7934         10       1300       1300       1300
      7782         10       2450       1875       1875
      7839         10       5000 2916.66667 2916.66667

      7369         20        800        800        800
      7876         20       1100        950        950
```

```
      7566         20       2975       1625       1625
      7788         20       3000       2175    1968.75
      7902         20       3000       2175       2175

      7900         30        950        950        950
      7654         30       1250       1150       1100
      7521         30       1250       1150       1150
      7844         30       1500     1237.5     1237.5
      7499         30       1600       1310       1310
      7698         30       2850 1566.66667 1566.66667

SQL>
```

En mi opinión, la windowing_clause predeterminada debería haber sido **ROWS BETWEEN UNBOUNDED PRECEDING Y UNBOUNDED FOLLOWING**. Esto haría que la inclusión accidental de windowing_clause fuera mucho menos confusa.

La siguiente consulta muestra un método para acceder a los datos de las filas anteriores y siguientes dentro de la fila actual utilizando la cláusula_windowing. Esto también se puede lograr con **LAG y LEAD**.

```
CLEAR BREAKS

SELECT empno, deptno, sal,
       FIRST_VALUE(sal) OVER (ORDER BY sal ROWS BETWEEN 1 PRECEDING AN
D CURRENT ROW) AS previous_sal,
       LAST_VALUE(sal) OVER (ORDER BY sal ROWS BETWEEN CURRENT ROW AND
 1 FOLLOWING) AS next_sal
FROM   emp;

     EMPNO     DEPTNO        SAL PREVIOUS_SAL   NEXT_SAL
---------- ---------- ---------- ------------ ----------
      7369         20        800          800        950
      7900         30        950          800       1100
      7876         20       1100          950       1250
      7521         30       1250         1100       1250
      7654         30       1250         1250       1300
      7934         10       1300         1250       1500
      7844         30       1500         1300       1600
      7499         30       1600         1500       2450
      7782         10       2450         1600       2850
      7698         30       2850         2450       2975
      7566         20       2975         2850       3000
      7788         20       3000         2975       3000
```

```
      7902          20        3000          3000        5000
      7839          10        5000          3000        5000

SQL>
```

PDF generated by Kalin's PDF Creation Station