ORACLE SQL

Lesson 04: 11g Features

Lesson Objectives

To understand the following topics:

- Analytical functions
- Read only tables
- Result cache
- Virtual Columns
- Invisible indexes

Comparing:Analytical and Group Functions

How are analytical different from Group Functions?

- These functions give aggregate results
- But unlike group functions they do not group the result set
- Analytic function return the group/aggregate column value with each record
- Any non “group by” columns or expressions can appear in the select statement which is not possible when using “Group By”

Examples

Consider the example to understand the difference between Group and Analytic functions

- Using Group Functions:

```
SELECT deptno, COUNT(*) DEPT_COUNT FROM emp
WHERE deptno IN (20, 30) GROUP BY deptno;
```

- Using Analytic Functions

```
SELECT empno, deptno, COUNT(*) OVER (PARTITION BY
deptno) DEPT_COUNT FROM emp
WHERE deptno IN (20, 30);
```

Examples

Typical syntax for using analytic function would be:

```
Select … function (arg1,…,argn) OVER
([PARTITION BY column name][ORDER BY
sql_expression][<Window Clause>]
```

- The "partition by" clause is used to break the result set in groups
- To limit the number of records within the result set groups returned with Partition By, "Window Clause" can be used

Examples

Include the ORDER BY clause in the OVER()

Syntax:

```
OVER (PARTITION BY column name ORDER BY <sql_expression>
[ASC or DESC] NULLS [FIRST or LAST]
```

Analytic Function Listing

Common analytic functions which does not depend on the order of records

- SUM,COUNT,AVG,MIN,MAX

Analytic functions which depend on order of records

- ROW_NUMBER,RANK,DENSE_RANK,FIRST VALUE, LAST VALUE, FIRS,LAST

ROW_NUMBER , RANK & DENSE_RANK

All these functions will assign integer values to rows based on the order of the rows.

- ROW_NUMBER : For providing serial number to a partition of records.
- Example:

```
SELECT deptno, empno, hiredate,
ROW_NUMBER( ) OVER (PARTITION BY deptno
ORDER BY hiredate NULLS LAST) SRLNO
FROM emp WHERE deptno IN (10, 20)
ORDER BY deptno, SRLNO;
```

4.1: Analytical functions
ROW_NUMBER , RANK & DENSE_RANK

RANK : Based on the column value or expression provides rank to the records. If two records at position n have the same value, then RANK assigns the same rank value to both the records and to the next record it will assign value n+2. It does not use the value of n+1 and skips it

Example

```
SELECT empno, deptno, sal, RANK() OVER
(PARTITION BY deptno
ORDER BY sal DESC NULLS LAST) RANK
FROM emp WHERE deptno IN (10, 20)
ORDER BY deptno, RANK
```

4.1: Analytical functions

ROW_NUMBER , RANK & DENSE_RANK

DENSE_RANK : This function also provides rank based on the column value or expression to the records. If two records at position n have the same value, then DENSE_RANK assigns the same rank value to both the records , but unlike RANK, DENSE_RANK assigns n+1 value to the next record

Example

```
SELECT empno, deptno, sal, DENSE_RANK() OVER
(PARTITION BY deptno ORDER BY sal DESC NULLS LAST)
DENSE_RANK
FROM emp WHERE deptno IN (10, 20)
ORDER BY deptno, DENSE_RANK
```

FIRST_VALUE and LAST_VALUE

FIRST_VALUE and LAST_VALUE: In both the functions, sql_expression is computed and the results are returned. The FIRST_VALUE takes the first record from the partition and LAST_VALUE takes the last record of the partition.

- Syntax:

```
FIRST_VALUE(<sql_expression>) OVER (<analytic_clause>)
```

FIRST_VALUE and LAST_VALUE

Example

```
SELECT empno, deptno, hiredate - FIRST_VALUE(hiredate)
OVER (PARTITION BY deptno ORDER BY hiredate) DAY_GAP
FROM emp
WHERE deptno IN (20, 30)
ORDER BY deptno, DAY_GAP
```

FIRST and LAST

FIRST and LAST: This two functions enable the users to apply an aggregate function on a group of records which have the same ranking. The first function is used for First ranked record and LAST for the last ranked records

- Syntax:

```
function( ) KEEP (DENSE_RANK FIRST/LAST
ORDER BY <expression>) OVER (<partitioning_clause>)
```

FIRST and LAST

Example

```
SELECT empno, deptno, TO_CHAR(hiredate,'YYYY') HIRE_YEAR,
sal, TRUNC( AVG(sal) KEEP (DENSE_RANK FIRST
ORDER BY TO_CHAR(hiredate,'YYYY') )
OVER (PARTITION BY deptno)) Average_Sal_Hire_Yr1
FROM emp
WHERE deptno IN (20, 10)
ORDER BY deptno, empno, HIRE_YEAR
```

Window Clause

The analytical functions can include the Window clause

This clause allows to further limit the rows which are returned by the partition clause and the analytic function can then be applied on this limited set of rows

Syntax:

```
[ROW or RANGE] BETWEEN <start_expression>
AND <end_expression>
```

ROW Type Windows

Syntax:

```
function( ) OVER (PARTITION BY <expr1> ORDER BY <expr2,..>
ROWS BETWEEN <start_expr> AND <end_expr>)
```

```
function( ) OVER (PARTITON BY <expr1> ORDER BY <expr2,..>
ROWS [<start_expr> PRECEDING or UNBOUNDED PRECEDING]
```

RANGE Type Windows

Syntax:

```
function( ) OVER (PARTITION BY <expr1> ORDER BY <expr2>
RANGE BETWEEN <start_expr> AND <end_expr>)
```

OR

```
function( ) OVER (PARTITION BY <expr1> ORDER BY <expr2>
RANGE [<start_expr> PRECEDING or UNBOUNDED PRECEDING]
```

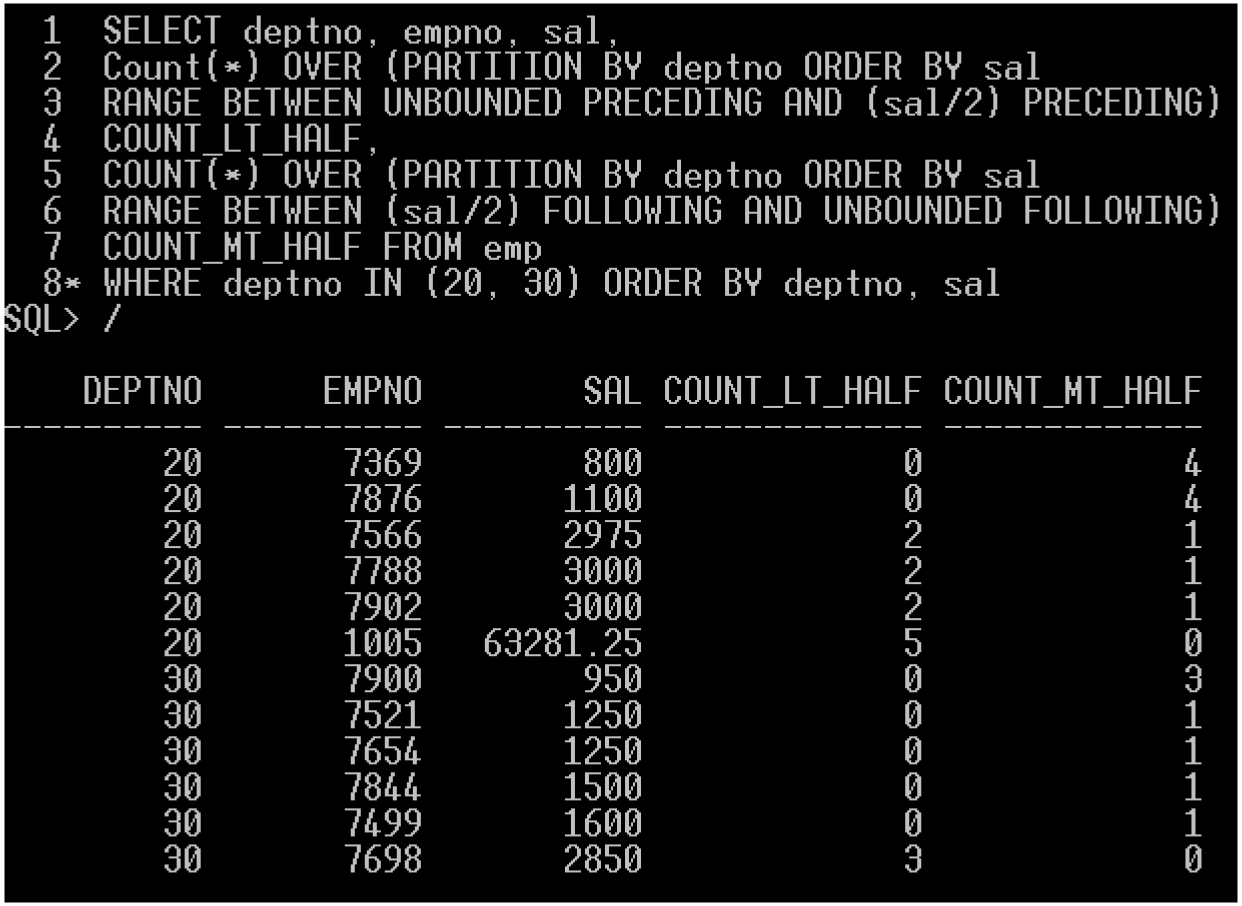
4.1: Analytical functions
RANGE Type Windows

Example

```
SELECT deptno, empno, sal,
Count(*) OVER (PARTITION BY deptno ORDER BY sal
RANGE BETWEEN UNBOUNDED PRECEDING AND (sal/2)
PRECEDING)
COUNT_LT_HALF,
COUNT(*) OVER (PARTITION BY deptno ORDER BY sal
RANGE BETWEEN (sal/2) FOLLOWING AND UNBOUNDED
FOLLOWING)
COUNT_MT_HALF FROM emp
WHERE deptno IN (20, 30) ORDER BY deptno, sal
```

4.1: Analytical functions

RANGE Type Windows



```
  1  SELECT deptno, empno, sal,
  2  Count(*) OVER (PARTITION BY deptno ORDER BY sal
  3  RANGE BETWEEN UNBOUNDED PRECEDING AND (sal/2) PRECEDING)
  4  COUNT_LT_HALF,
  5  COUNT(*) OVER (PARTITION BY deptno ORDER BY sal
  6  RANGE BETWEEN (sal/2) FOLLOWING AND UNBOUNDED FOLLOWING)
  7  COUNT_MT_HALF FROM emp
  8* WHERE deptno IN (20, 30) ORDER BY deptno, sal
SQL> /

    DEPTNO      EMPNO        SAL COUNT_LT_HALF COUNT_MT_HALF
---------- ---------- ---------- ------------- -------------
        20       7369        800             0             4
        20       7876       1100             0             4
        20       7566       2975             2             1
        20       7788       3000             2             1
        20       7902       3000             2             1
        20       1005   63281.25             5             0
        30       7900        950             0             3
        30       7521       1250             0             1
        30       7654       1250             0             1
        30       7844       1500             0             1
        30       7499       1600             0             1
        30       7698       2850             3             0
```

4.2: Read-Only Tables

Overview

One of the more useful but simple to implement new features in Oracle Database 11g is the read-only table feature

This allows us to make a table read-only across the database

We can use the read-only feature to prevent changes to a table’s data during maintenance operations.

The following examples shows the use of the alter table command along with the read only keywords to make a table read-only, and then the use of the read write keywords to make the table read-write:

Code Snippet

```
SQL> create table test (name varchar2(30));
Table created.
SQL> alter table test read only;
Table altered.
SQL>
```

4.2: Read-Only Tables

Overview (Contd…)

Once we put a table in a read-only mode, we can’t issue any DML statements such as update, insert, or delete.

We also can’t issue a select for update statement involving a read only table.

We can issue DDL statements such as drop table and alter table on a read-only table, however.

We can use this feature for security reasons, where we want to grant users the ability to read but not modify table data.

Overview

New hint in SQL result_cache

Caches the resultset of select statement

The result will be returned from the cache if any other user session issues the same SQL statement

This improves the performance of the SQL statement.

SQL Hint /*+result_cache*/

Result_cache can be enabled in following ways:

- Issue the command to cache the session data
  - Alter session cache results;
- Add the result_cache hint in SQL statement
  - Select /*+ result_cache */……
- Create a PL/SQL function using the result_cache keyword
- It is also a scalable execution feature since result cache can function both at client or server side.

Parameters to be set for result_cache

Result_cache_max_size: specifies the maximum area in bytes to be allocated for result_cache storage within the SGA

Result_cache_max_result: specifies maximum percentage of the result_cache are that can be used by a single resultset

Result_cache_mode: specified when ResultCache operator is used in the query execution plan. The valid values for this parameter are “manual” or “force”

Result_cache_remote_expiration: timespan in minutes that a result cache will be alive

4.3 Result_cache

Benefits of using result_cache

Can be useful in Datawarehouse application since result_cache is mainly used for static data
Reduces the overhead of repeated computation of static values for deterministic functions
The reads to db_cache is reduced and file I/O is reduced.

Limitations of result_cache

Result_cache is active till the number of minutes specified by parameter result_cache_remote_expiration

The result cache is likely to become stale quickly if the table data undergoes change

Use result_cache in a procedure which does not have OUT or INOUT parameters

When using result_cache in procedure/function where parameter or return type should not LOBs, REF CURSOR, Record, Object, PL/SQL collection

4.4: Virtual Columns
Overview

The ability to create virtual columns is a new feature in Oracle Database 11g

- It provides the ability to define a column that contains derived data, within the database
- Derived values for virtual columns are calculated by defining a set of expressions or functions that are associated with the virtual column
- They do not consume any storage, as they are computed on the fly.
- Virtual columns can be used in queries, DML, and DDL statements
- They can be indexed.
- We can collect statistics on them.

Thus, they can be treated much as other normal columns

Creating Tables with Virtual Columns

To create a virtual column within a create table or alter table statement, you use the new ‘AS’ command, which is part of the virtual column definition clause.

The syntax for defining a virtual column is listed below

```
column_name [datatype] [GENERATED ALWAYS] AS
(expression) [VIRTUAL]
```

Creating Tables with Virtual Columns (Contd…)

Code Snippet

```
SQL> Create table emp
  2  ( emp_id number primary key,
  3  salary number (8,2) not null,
  4  years_of_service number not null,
  5  curr_retirement as (salary*.0005 *
years_of_service) );

Table created.
```

Creating Tables with Virtual Columns (Contd…)

Code Snippet

```
SQL> insert into emp values(1,10000, 1,default);
1 row created.

SQL> insert into emp(emp_id, salary, years_of_service)
values(2,20000, 2);
1 row created.

SQL> select * from emp;
EMP_ID     SALARY YEARS_OF_SERVICE CURR_RETIREMENT
---------- ---------- ---------------- ---------------
         1      10000                1               5
         2      20000                2              20
```

Restrictions on Virtual Columns

You can create virtual columns only in regular tables. Virtual columns are not supported for index-organized, external, object, cluster, or temporary tables.

The column_expression in the AS clause has the following restrictions:

- It cannot refer to another virtual column by name.
- Any columns referenced in column_expression must be defined on the same table.
- It can refer to a deterministic user-defined function, but if it does, then you cannot use the virtual column as a partitioning key column.

The output of column_expression must be a scalar value.

The virtual column cannot be an Oracle supplied data type, a user-defined type, or LOB or LONG RAW.

You cannot specify a call to a PL/SQL function in the defining expression for a virtual column that you want to use as a partitioning column.

Indexes on Virtual Columns

We can create indexes on Virtual Columns

```
SQL> CREATE TABLE t
  2  ( n1 INT
  3  , n2 INT
  4  , n3 INT GENERATED ALWAYS AS (n1 + n2) VIRTUAL
  5  );
Table created
SQL> CREATE INDEX t_pk ON t(n3);
Index created.
```

Constraints on Virtual Columns

```
SQL> create table p ( b varchar2(2) primary key);
SQL> create table c  ( a varchar2(10),b as (substr( a, 5, 2
)) references p);
SQL> insert into p values('RD');
SQL> insert into c(a) values('PLSQLRD');
insert into c(a) values('PLSQLRD')
*
```

```
ERROR at line 1:
ORA-02291: integrity constraint (SCOTT.SYS_C009544) violated –
parent key not found
SQL> insert into c(a) values('PLSQRD');
1 row created.
```

4.5: Invisible Indexes

Overview

When we create indexes, optimizer starts using it…

Perhaps when you dropped that index, performance declines because existing execution plans were dependent on it

Dropping an index can be difficult if a number of concurrent processes are using it

Rebuilding a dropped index can also be time-consuming

Oracle 11g allows indexes to be marked as invisible.

- Invisible indexes are maintained like any other index, but they are ignored by the optimizer unless the OPTIMIZER_USE_INVISIBLE_INDEXES parameter is set to TRUE at the instance or session level
- therefore will not consider when generating execution plans even if the index is specifically mentioned in a hint.
- Default value for this parameter is FALSE

Set an index to Visible or Invisible

You can make an index invisible when you create it with the create index command by using the invisible keyword as seen in this example

Code Snippet

```
Create index ix_test on test(id) invisible;
Alter index ix_test visible;
alter index ix_test invisible
```

4.5: Invisible Indexes
Example

Make the index as visible
Code Snippet:

```
SQL> alter index ix_test visible;
Index altered.
SQL>SET AUTOTRACE ON
SQL> select * from test where id=1;
------------------------------------------------------------------------------
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time
|
------------------------------------------------------------------------------
| 0 | SELECT STATEMENT | | 1 | 13 | 1 (0)| 00:00:01 |
|* 1 | INDEX RANGE SCAN| IX_TEST | 1 | 13 | 1 (0)|
00:00:01 |
------------------------------------------------------------------------------
```

Example (Contd…)

Now, make it invisible

Code Snippet:

```
SQL> alter index ix_test invisible;
Index altered.
SQL> select * from test where id=1;
--------------------------------------------------------------------------
-----
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time
|
--------------------------------------------------------------------------
-----
| 0 | SELECT STATEMENT | | 1 | 13 | 24 (5)| 00:00:01 |
|* 1 | TABLE ACCESS FULL| TEST | 1 | 13 | 24 (5)| 00:00:01
|
--------------------------------------------------------------------------
-----
```

SUMMARY

- In this lesson, you should have learned how to use,
  - Analytical functions
  - Read only tables
  - Result cache
  - Virtual Columns
  - Invisible indexes

Review Question

❖Question 1: Which of the following analytic functions depend on order of records

- Option 1: Sum
- Option 2: Avg
- Option 3: Rank

❖Question 2 : The ________ clause allows to further limit the number of records retrieved by partition clause

❖Question 3: enable the users to apply an aggregate function on a group of records which have the same ranking