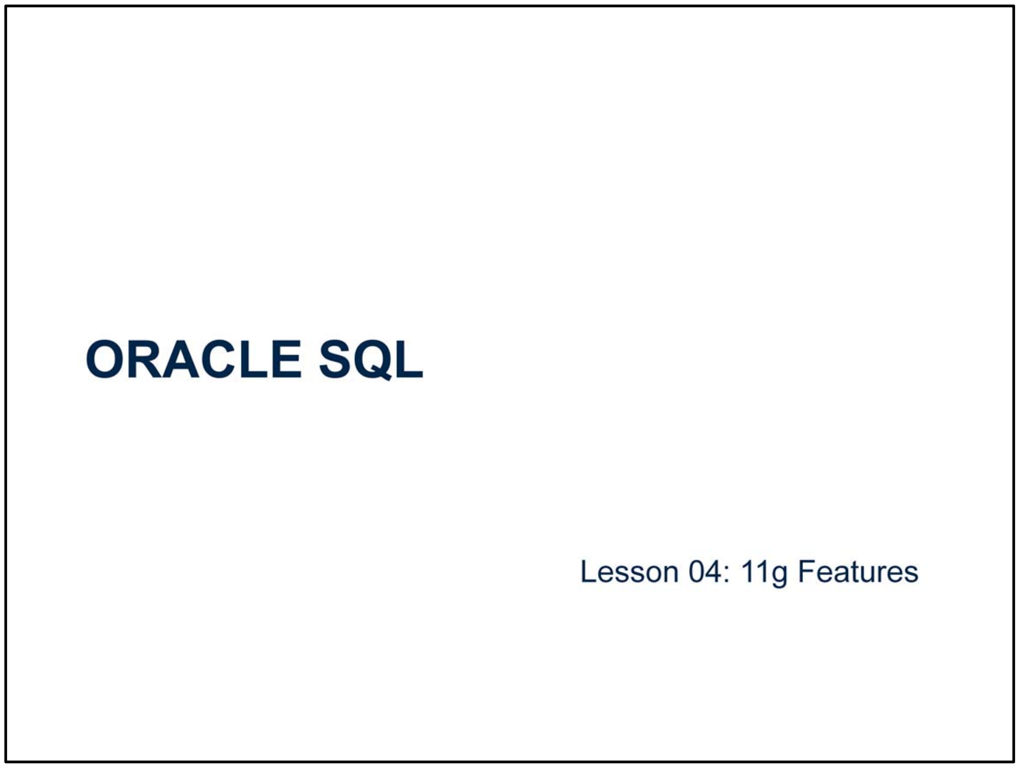
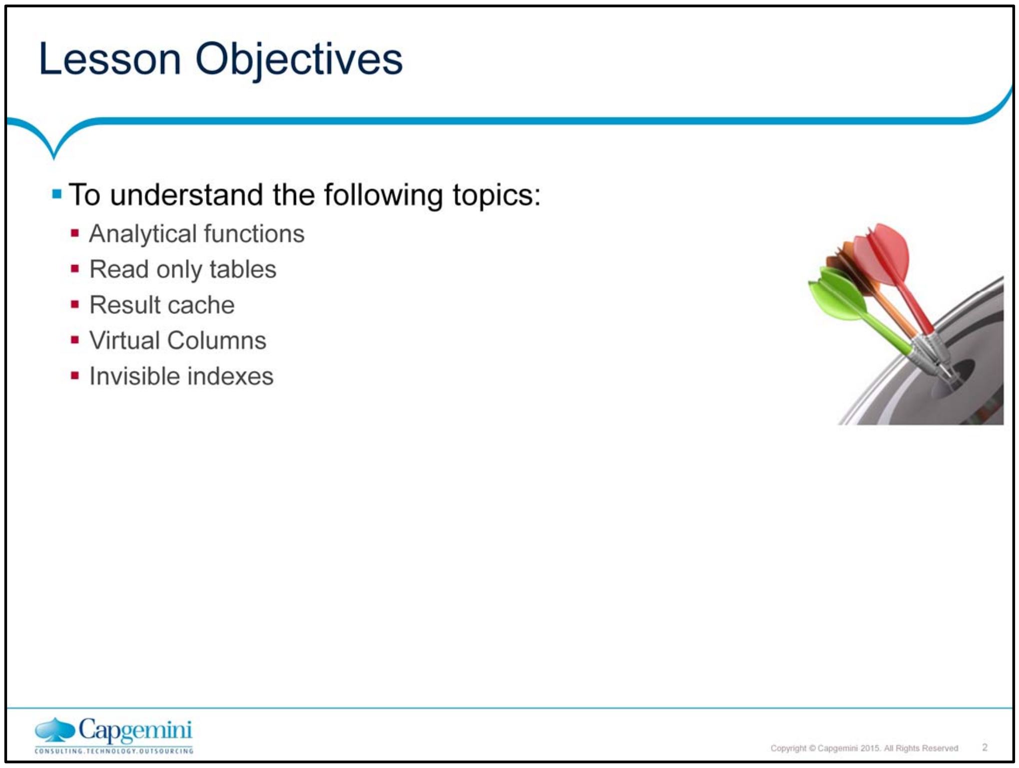


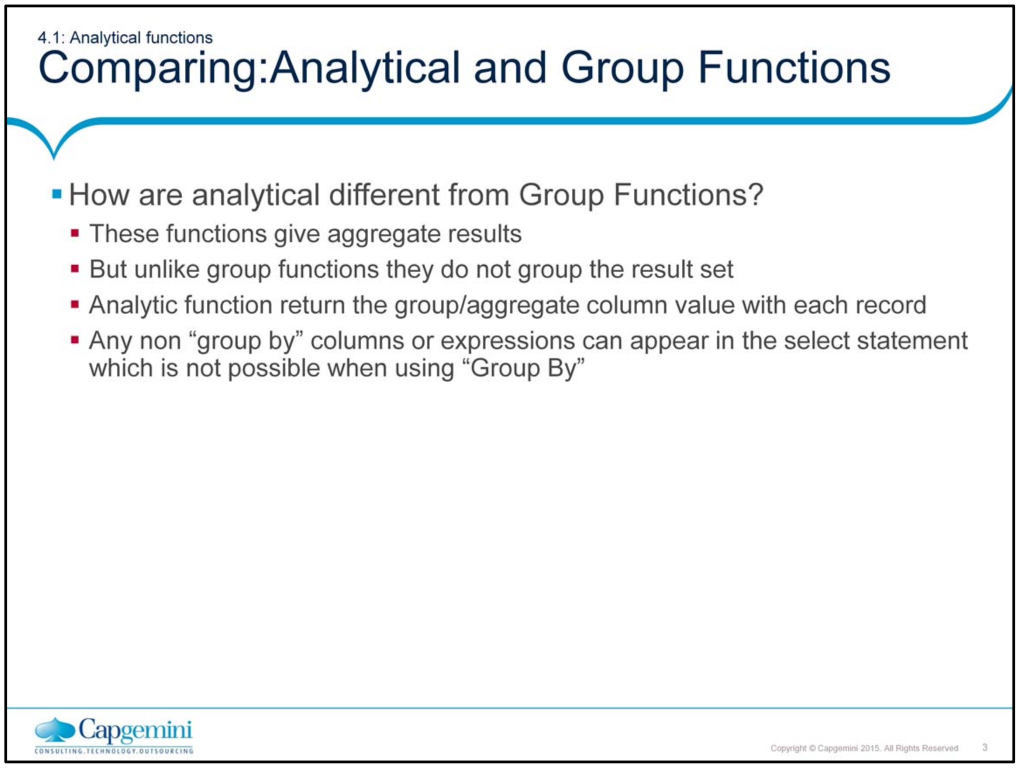
ORACLE SQL

Lesson 04: 11g Features



Lesson Objectives

- To understand the following topics:
  - Analytical functions
  - Read only tables
  - Result cache
  - Virtual Columns
  - Invisible indexes



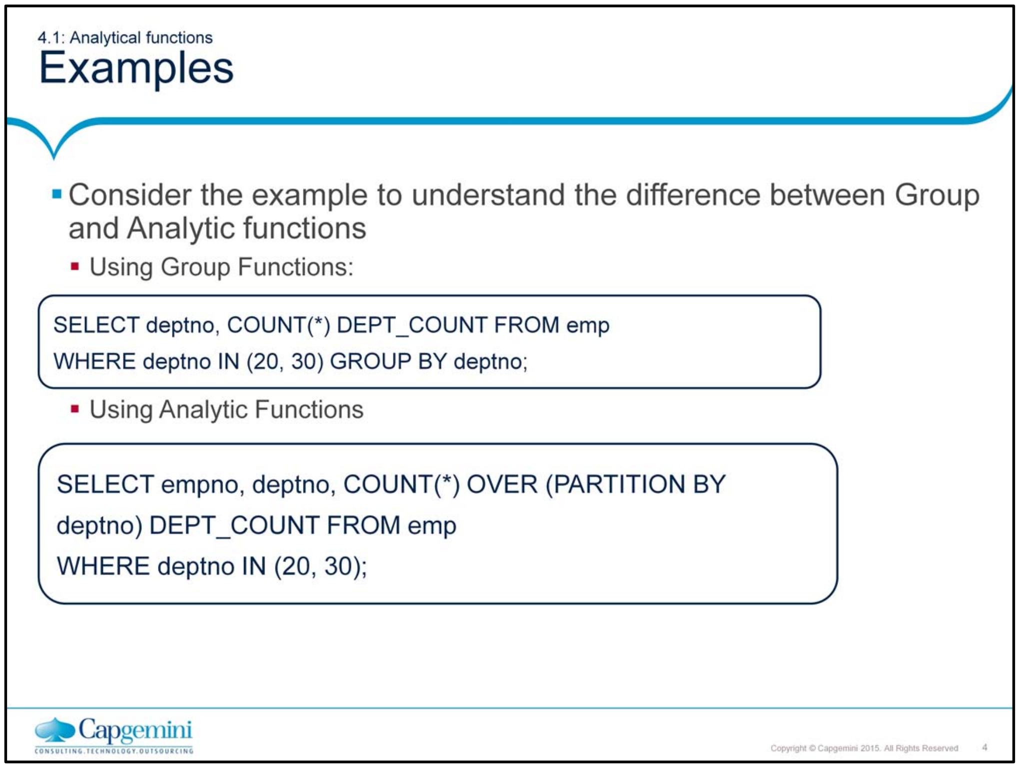
4.1: Analytical functions

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"

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

Analytic functions in Oracle are set of functions and clauses used for arriving statistical information like sum, average . Earlier, these problems were solved using PL/SQL. But when we look for better performance, PL/SQL does not benefit much. Analytical functions provides more benefits with no questions on performance. Analytical functions are capable of providing the functionality that can also be provided by basic SQL using features like group functions , subqueries and joins. But the application is faster when analytical functions are being used. In a query all the clauses such as Where, Group By, Having and joins are evaluated and then the analytic functions are computed. The order by clause is carried out at the end. Analytic functions are mostly used in Data Warehousing environments where you require compute cumulative, moving, centered, and reporting aggregates features.



4.1: Analytical functions

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;
```

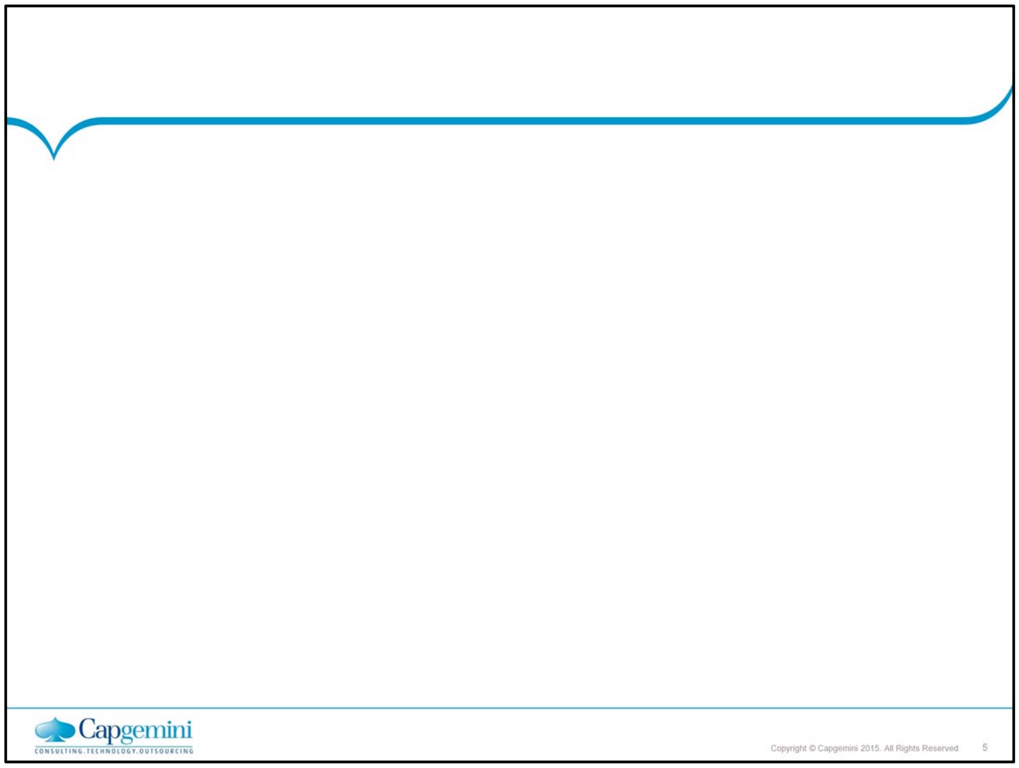
  - Using Analytic Functions

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

In the example considered on the slide, the first query uses the Group By clause. This output of the query is departments and their employee count which is as follows:

```
 DEPTNO DEPT_COUNT
--------    ----------
      20          5
      30          6
```

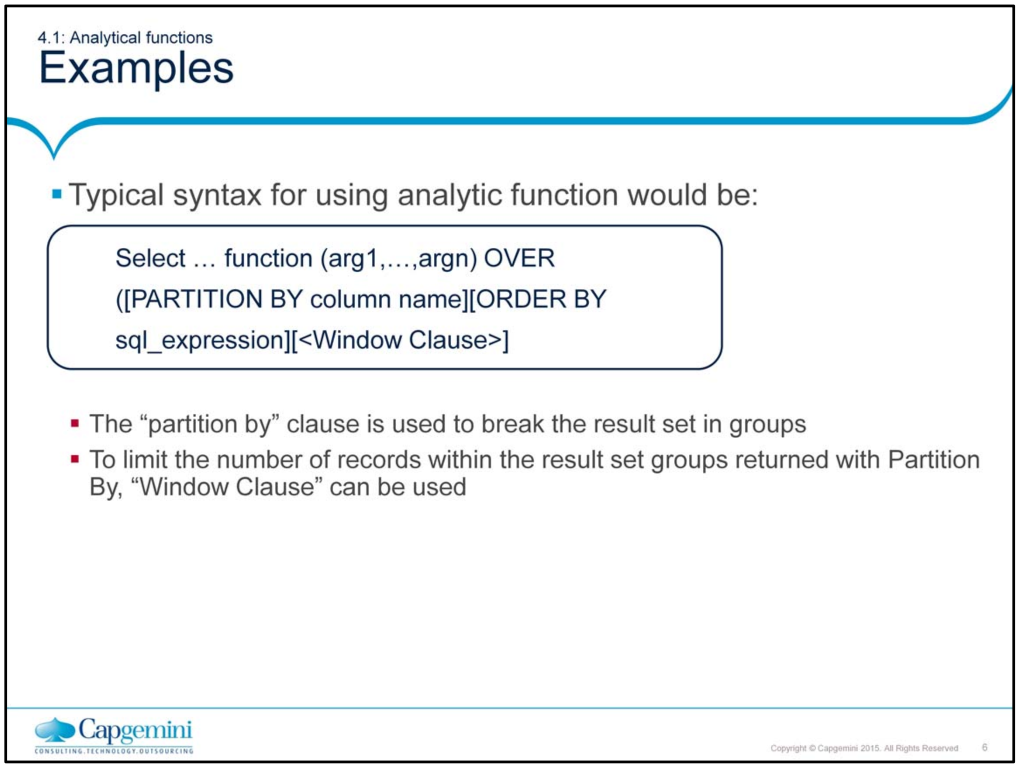
The query groups the records based on the departments and applies the count function on this groups. Also any non group by columns cannot be included in the select statement.



Now consider the second query in which analytic functions is used.

```
 EMPNO     DEPTNO DEPT_COUNT
-------- ---------- ----------
   7369        20          5
   7566        20          5
   7788        20          5
   7902        20          5
   7876        20          5
   7499        30          6
   7900        30          6
   7844        30          6
   7698        30          6
   7654        30          6
   7521        30          6
```

Notice the output of this query. In this the values of dept count are repeating for all the records. A non group by column empno is include in the query.



4.1: Analytical functions

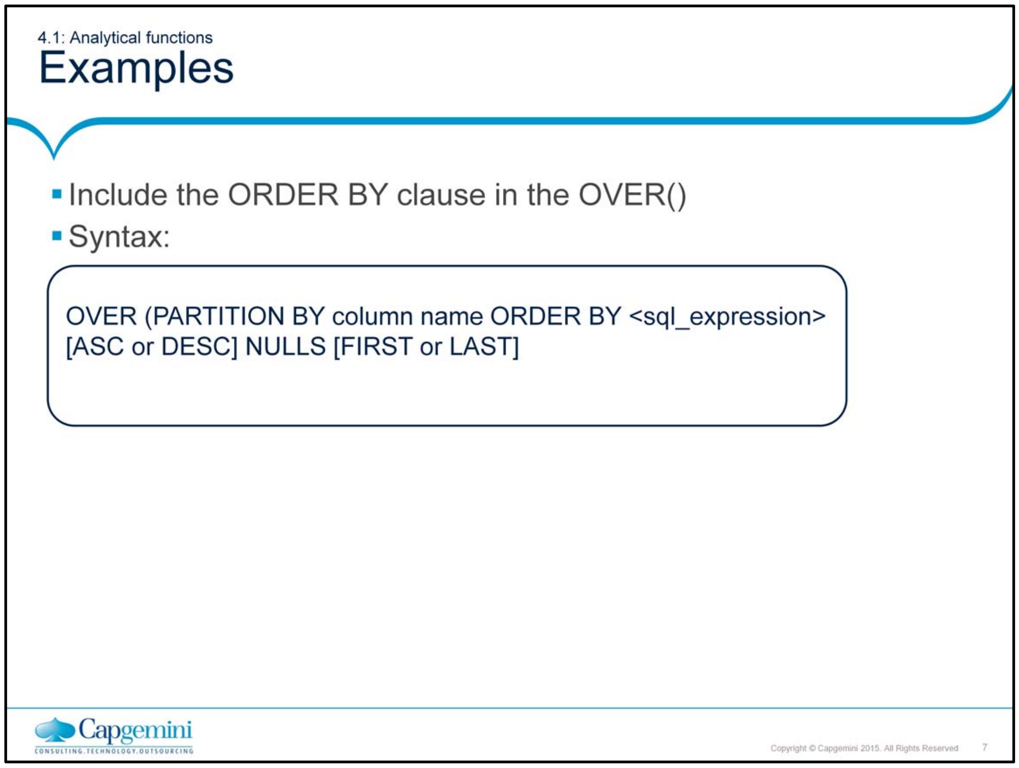
Examples

- 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

The syntax includes lot of keywords and the analytic function that has to be applied on the columns.
The Partition By is used for breaking the result set in groups and it can take any non analytic expression. The Window Clause can be used to limit the number of records returned by the Partition By clause else the partition clause would be applied to all the records fetch due to partition by. If the “partition by” or “window clause” is missing in the OVER portion, it acts on the entire record set.
We will understand in detail as we go along and look at the analytic functions available and how to use the other parameters in conjunction with them.



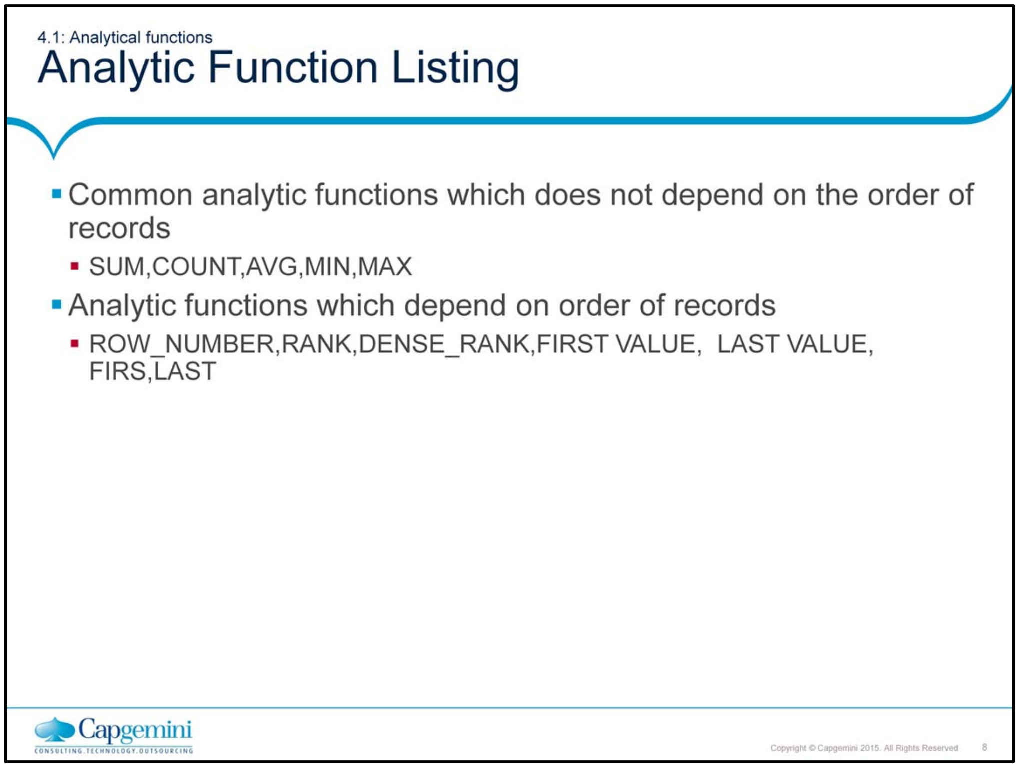
4.1: Analytical functions

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]
```

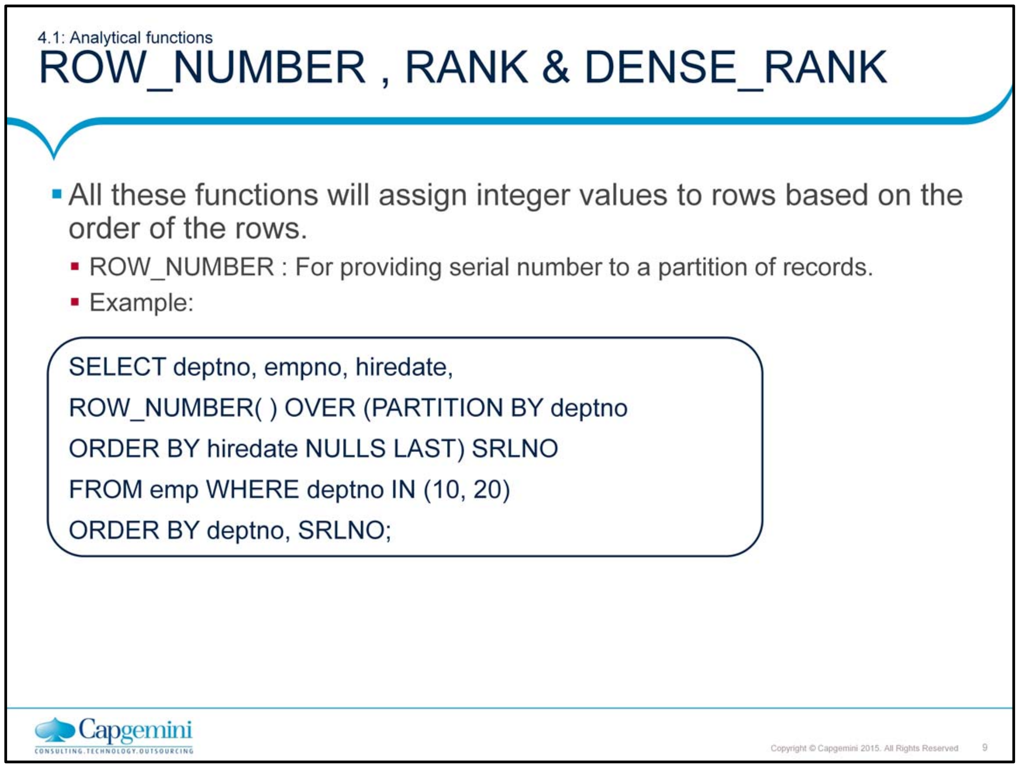
Include the ORDER BY clause in the OVER() to order the records within the partition resultset. This is required for the analytic functions which depend on the order of records.



4.1: Analytical functions

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



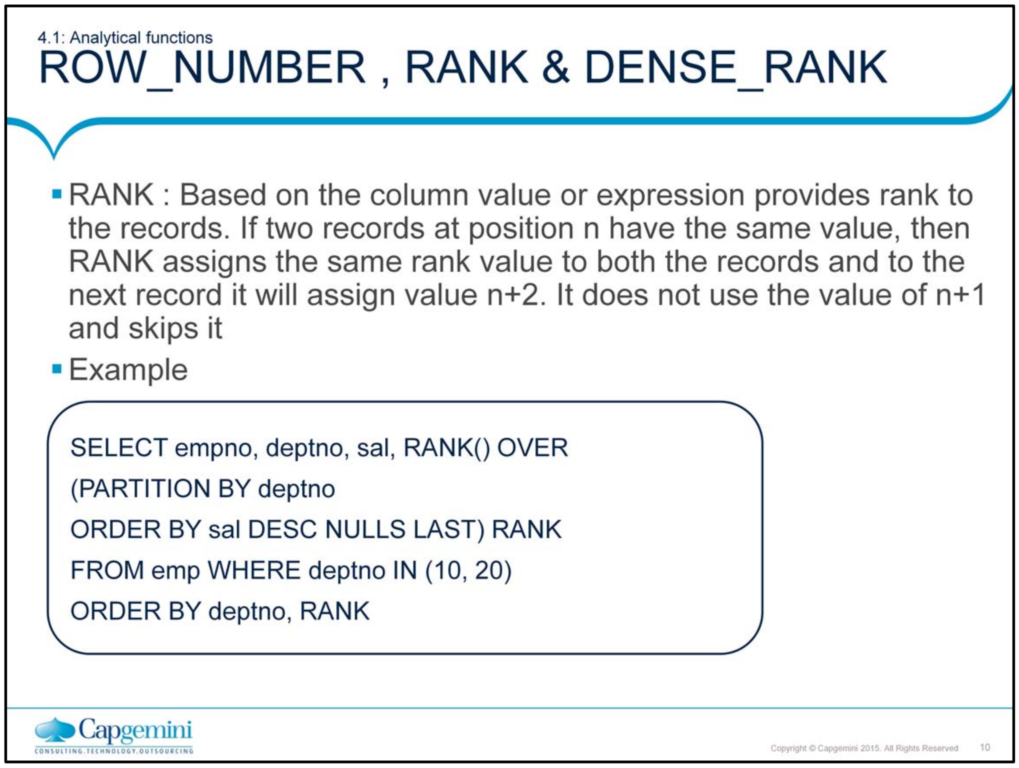
4.1: Analytical functions

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;
```

The ROW_NUMBER, RANK and DENSE_RANK functions are used to assign integer values to rows based on their ordering.
ROW_NUMBER: This function provides a serial number to the partition of records. Normally used in reporting where typically every partition will have its own serial number. In the example shown on the slide, the records are partitioned on deptno and by applying the where clause we have filtered the records to retrieve values for deptno 10 & 20. The records within the partition are ordered on the hiredate and by using the ROW_NUMBER function a serial number is provided to the sets of rows in each group i.e deptno 10 and deptno 20.


4.1: Analytical functions

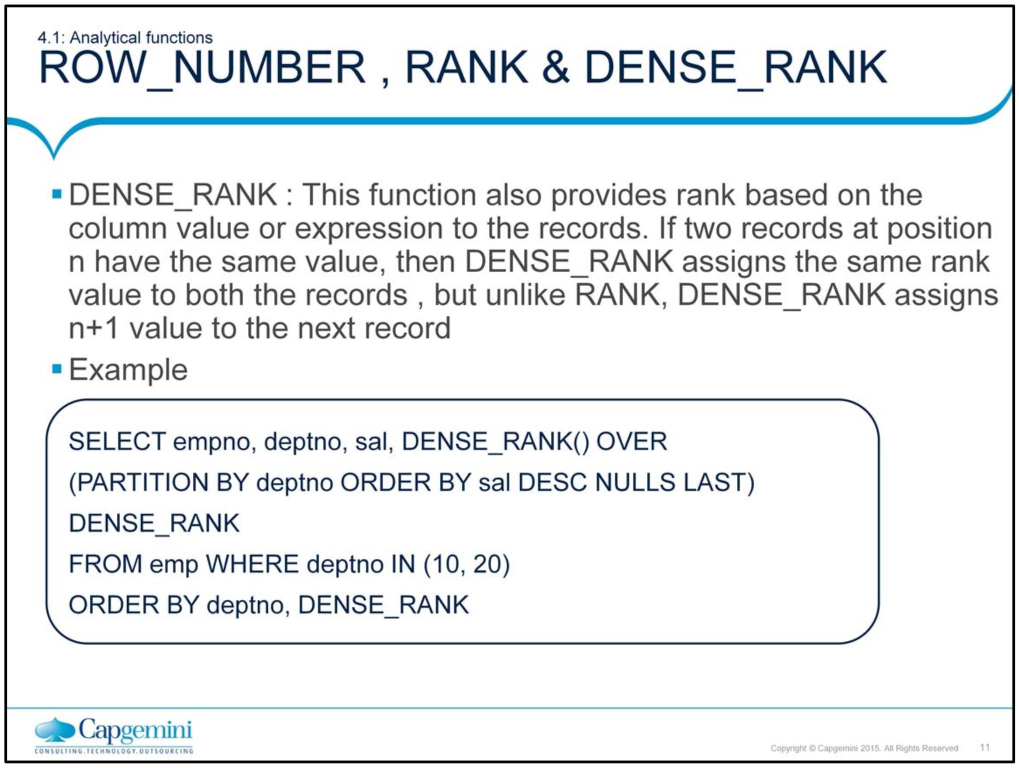
ROW_NUMBER , RANK & DENSE_RANK

- 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
```

RANK: The RANK function ranks the records based on some column value or expression. As mentioned on the slide RANK assigns the same rank value if there are two records with the same ranking.
In the example on the slide the rank is applied for the records based on the salary column in each group. If you notice the output in deptno 20 there are two records at Rank 1. Hence the RANK function assign value 1 to both of them and skips n+1 i.e Rank 2 in this case and provides Rank 3 to the next record which is listed.

```
 EMPNO     DEPTNO        SAL       RANK
---------- ---------- ---------- ----------
      7839         10       5000          1
      7782         10       2450          2
      7934         10       1300          3
      7788        20       3000         1
      7902        20       3000         1
      7566        20       2975         3
      7876         20       1100          4
      7369         20        800          5
```



4.1: Analytical functions

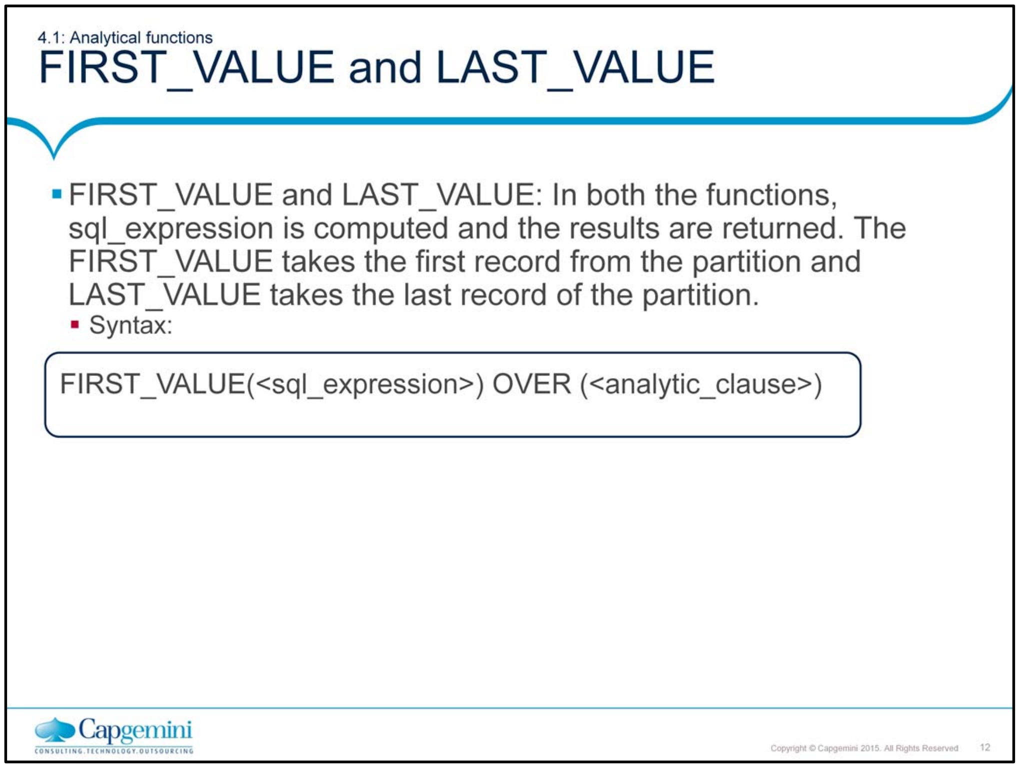
ROW_NUMBER , RANK & DENSE_RANK

- 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
```

DENSE_RANK: The DENSE_RANK function ranks the records based on some column value or expression. As mentioned on the slide DENSE_RANK assigns the same rank value if there are two records with the same ranking. And for the next record it assigns the next rank value i.e n+1.
In the example on the slide the records are ranked according to the salary column in each group. Similar to example considered in the previous slide there are two records at Rank1 position from deptno 20. DENSE_RANK function assign Rank 1 to them and to the next record assigns 2 unlike the RANK function which assigns 3 to the next record.

```
 EMPNO     DEPTNO        SAL DENSE_RANK
---------- ---------- ---------- ----------
      7839         10       5000          1
      7782         10       2450          2
      7934         10       1300          3
      7788        20       3000         1
      7902        20       3000         1
      7566        20       2975         2
      7876         20       1100          3
      7369         20        800         4
```

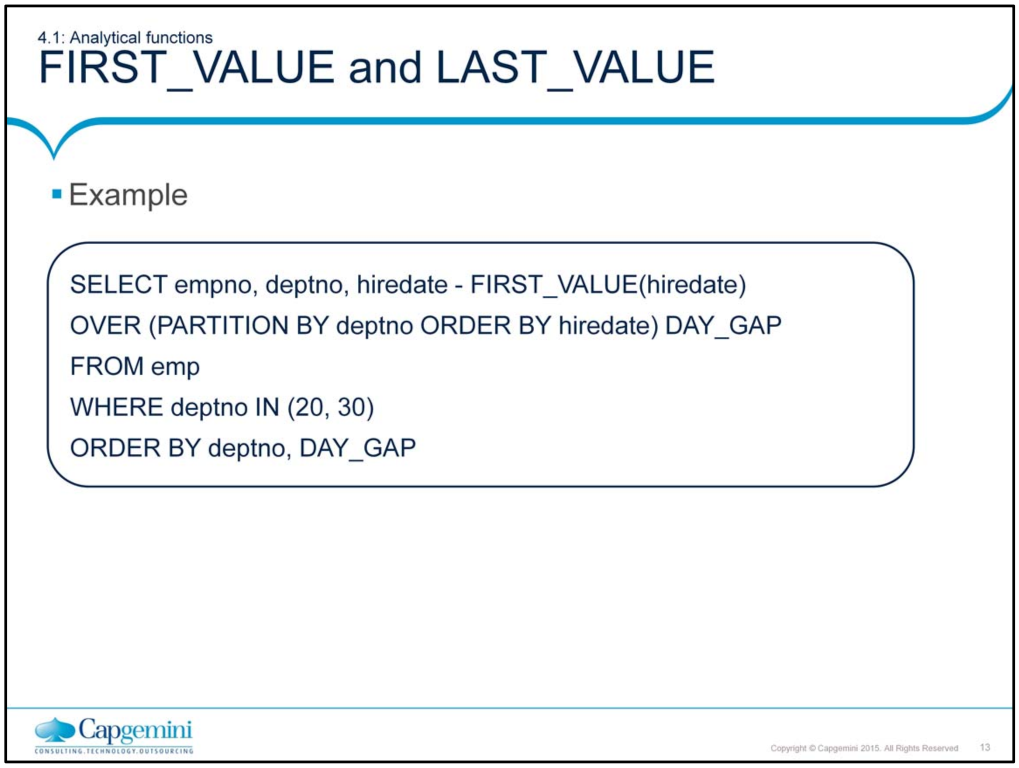


4.1: Analytical functions

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>)
```



4.1: Analytical functions

FIRST_VALUE and LAST_VALUE

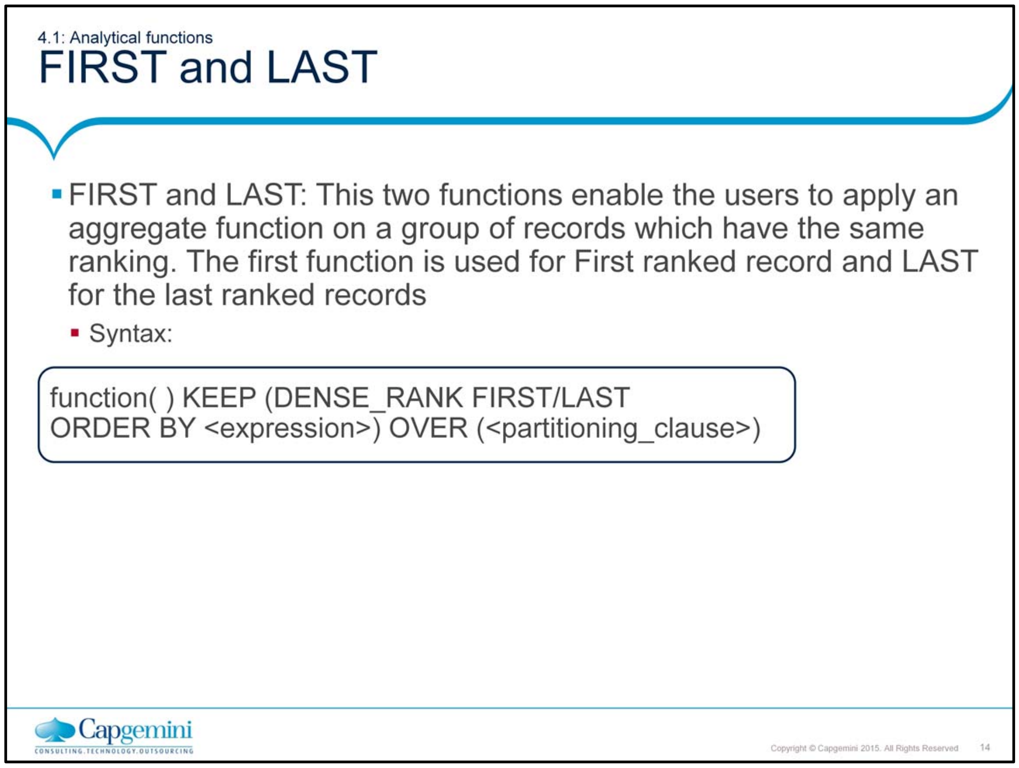
- Example

```
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
```

13

The example on the slide shows the gap between hiring of the first employee and the next employee in deptno 20 & 30

```
 EMPNO     DEPTNO    DAY_GAP
---------- ---------- ----------
      7369        20          0
      7566        20        106
      7902        20        351
      7788        20        722
      7876        20        756
      7499        30          0
      7521        30          2
      7698        30         70
      7844        30        200
      7654        30        220
      7900        30        286
```



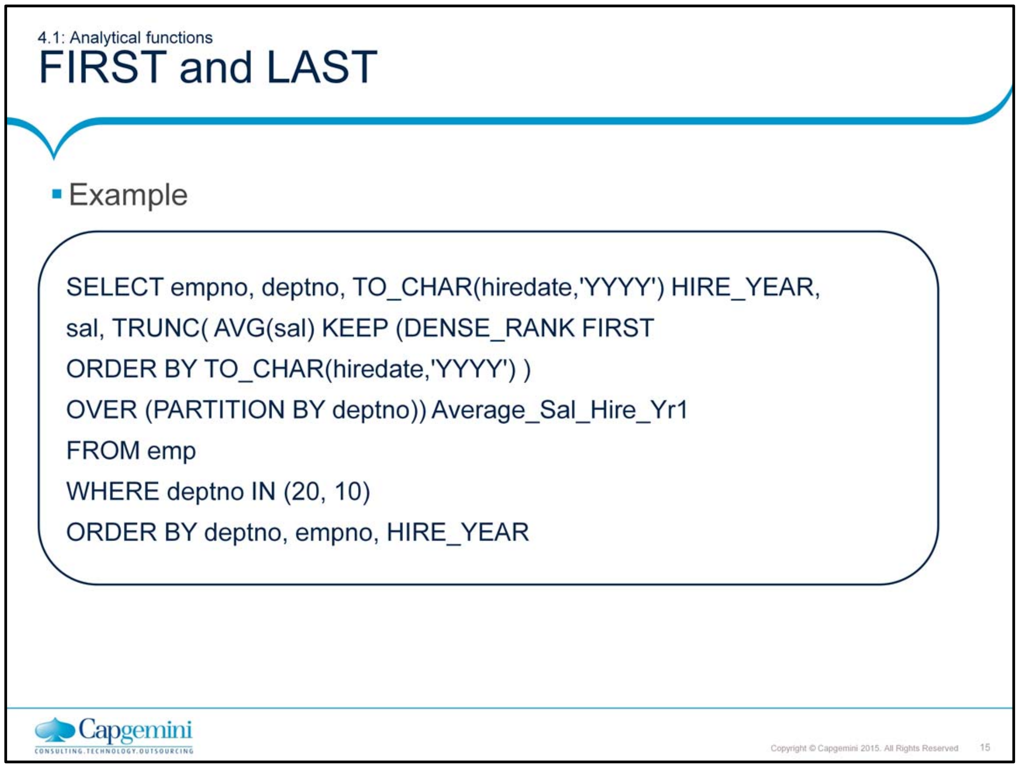
4.1: Analytical functions

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>)
```

The FIRST and the LAST function perform the similar task. The users can apply aggregate functions on a group of records which have equal ranking. More appropriately these functions can be called as KEEP FIRST or KEEP LAST. Both these functions do not follow the general syntax of analytic functions i.e they do not have order by clause inside the OVER clause. Also they do not support and <window> clause



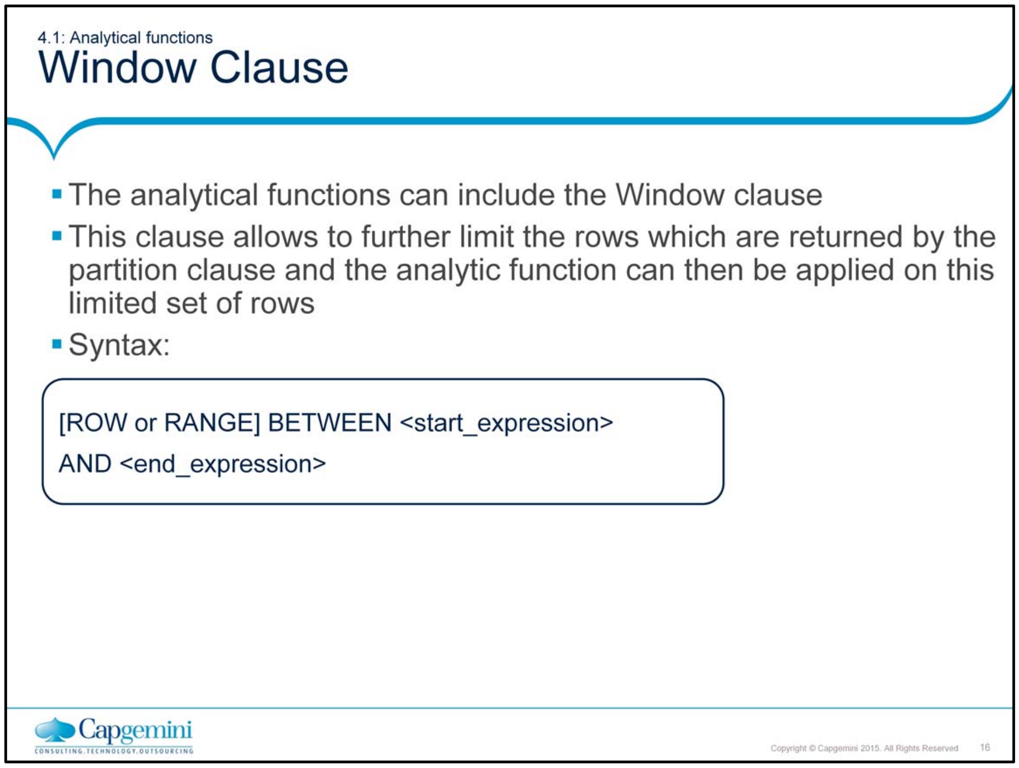
4.1: Analytical functions

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
```

In the example shown on the slide it compares the employees salary with the average salary of the employees who are hired in the first year in the respective departments.



4.1: Analytical functions

Window Clause

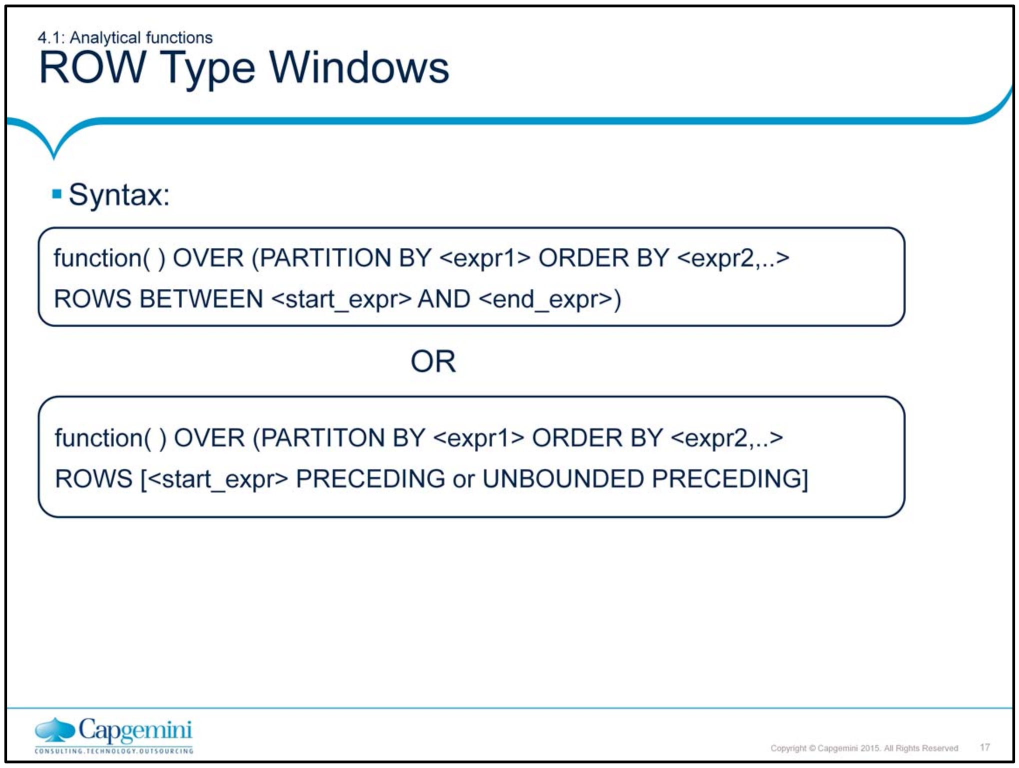
- The analytical functions can include the Window clause
- This clause allows to further limit the rows which are returned by the partition clause and the analytic function can then be applied on this limited set of rows
- Syntax:

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

If the user wants to limit or further partition the rows which are returned by the PARTITION BY clause then the Window clause can be used. The window clause is flexible.
The syntax of the window clause is:

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

ROW or RANGE : type of window. Start_expression – This can be Unbounded Preceding, Current Row or <sql_expression> Preceding or Following
End_expression - This can be Unbounded Following, Current Row or <sql_expression> Preceding or Following
Row type windows use row numbers before or after the current row. Range type windows us values before or after current order. Also Row or Range cannot appear together in one OVER clause. Whenever the window clause is specified with respect to the current row, but may not include the current row. The start expression and end expression can finish after the current row or before the current row. If the start or end expression is undefined the default value is UNBOUNDED PRECEDING FOR <start_expression> and UNBOUNDED FOLLOWING for <end_expression>.



4.1: Analytical functions

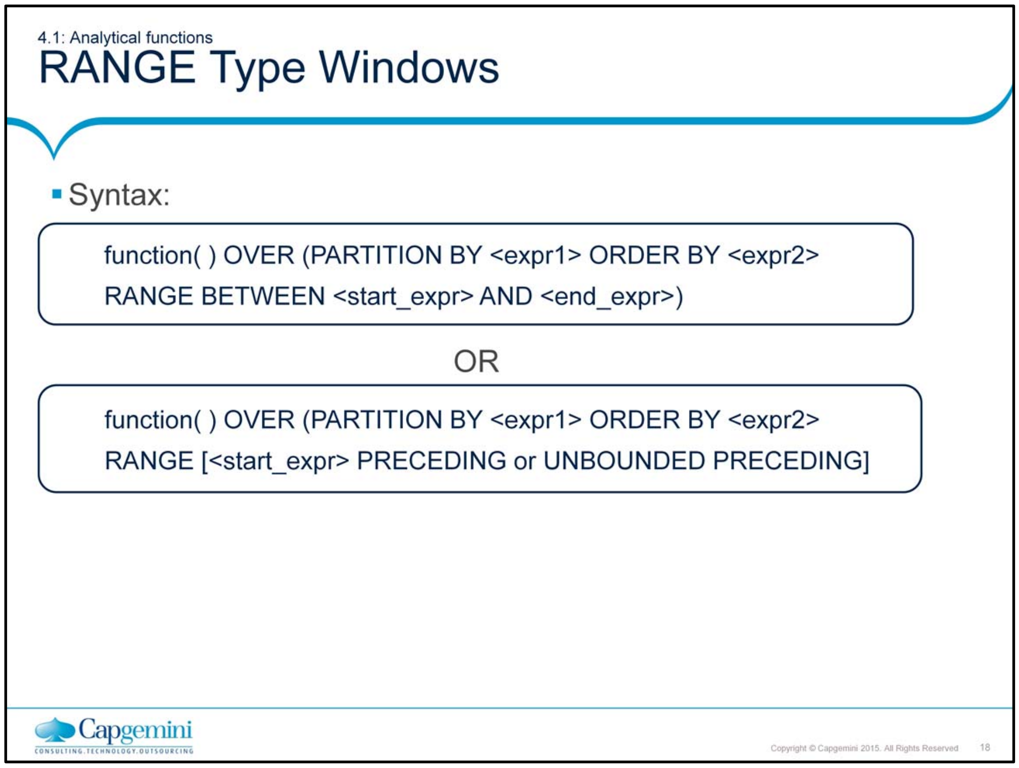
ROW Type Windows

- Syntax:

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

OR

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



4.1: Analytical functions

RANGE Type Windows

- Syntax:

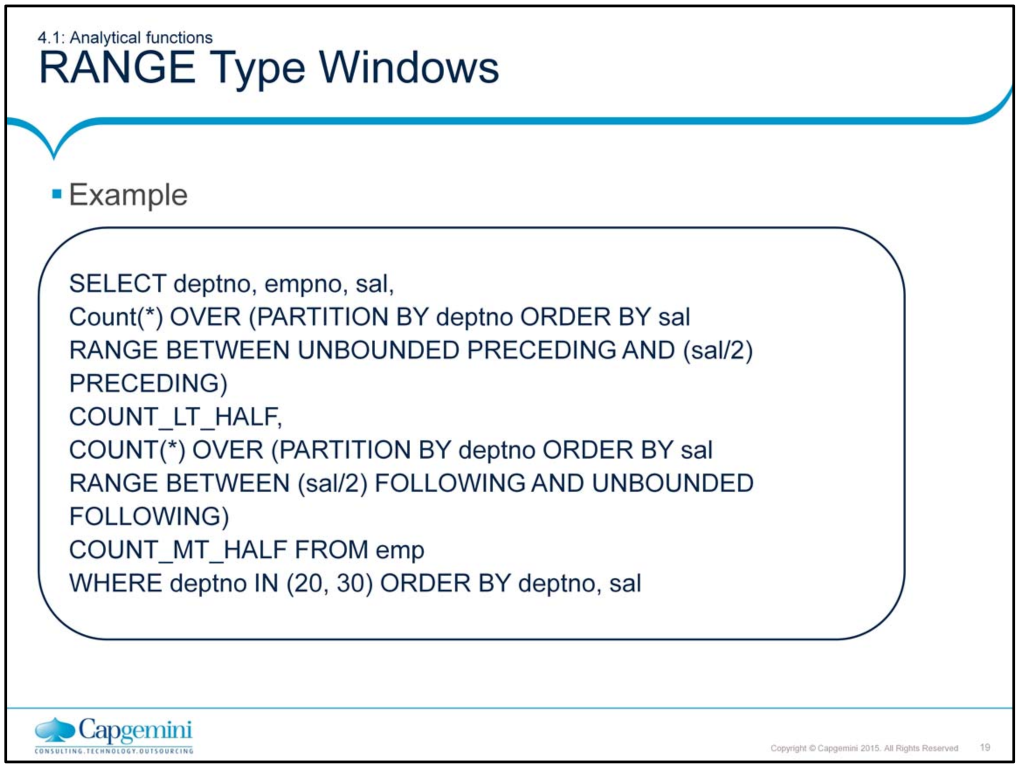
```
function( ) OVER (PARTITION BY <expr1> ORDER BY <expr2>
RANGE BETWEEN <start_expr> AND <end_expr>)
```

OR

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

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

18



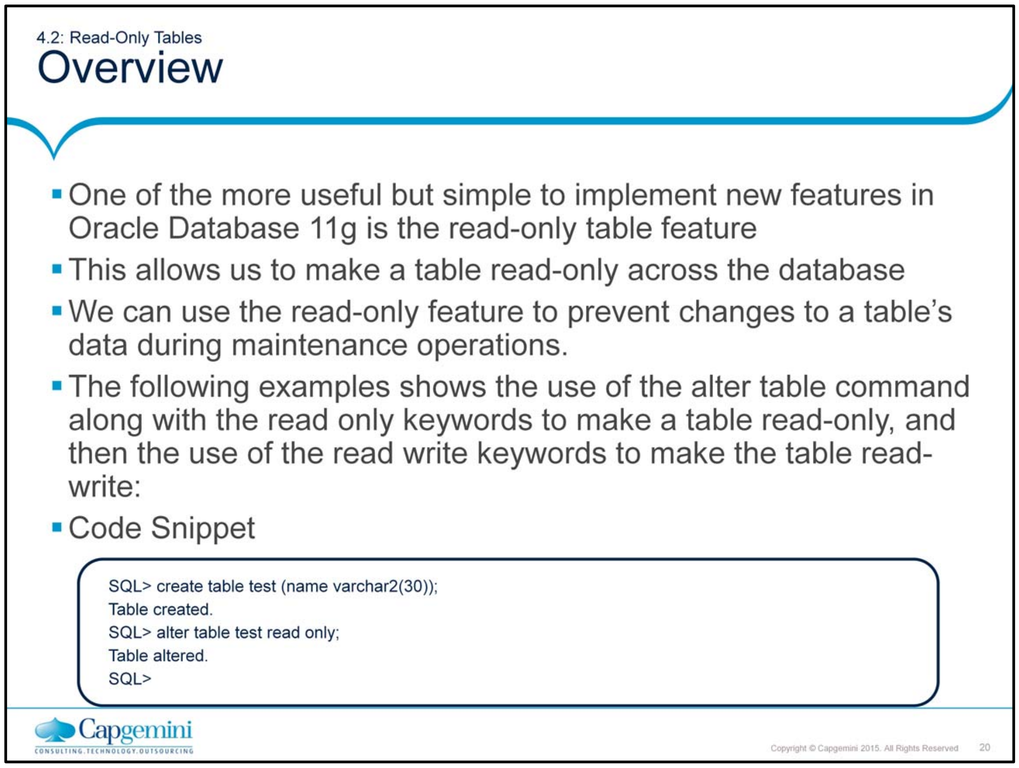
4.1: Analytical functions

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
```

In the example shown on the slide the query displays the count of employees who get less then half salary of the current record and salary more than half of the current record



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>
```

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

In earlier releases of the Oracle database, we have seen the concept of read-only tablespaces but not readonly tables. The new column in the [DBA / ALL / USER]_TABLES view, READ_ONLY, has been added to help us determine if a table is read-write or read-only. Valid values for the READ_ONLY column are YES and NO.
Now, we are going to check the READ_ONLY property of our TEST table:

```
SQL> select table_name,read_only from user_tables  where table_name='TEST';
TABLE_NAME                    REA
------------------------------ ---
TEST                          YES
```

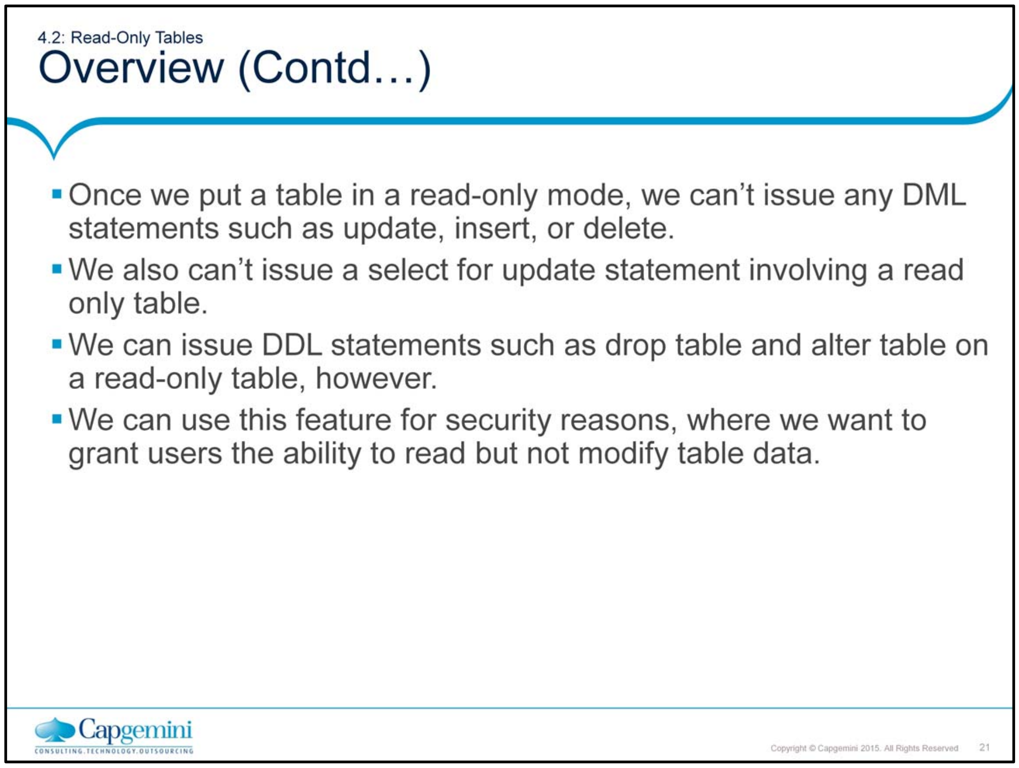
You can return a read-only table to a read-write status by using the read write clause in the alter table statement, as shown in the following example:

```
SQL> alter table test read write;
Table altered.
SQL>
```

Let's check the status again.

```
SQL> select table_name,read_only from user_tables where table_name='TEST';
TABLE_NAME                    REA
------------------------------ ---
TEST                          NO
```

We can see the NO - it's mean that table is not read only.



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.

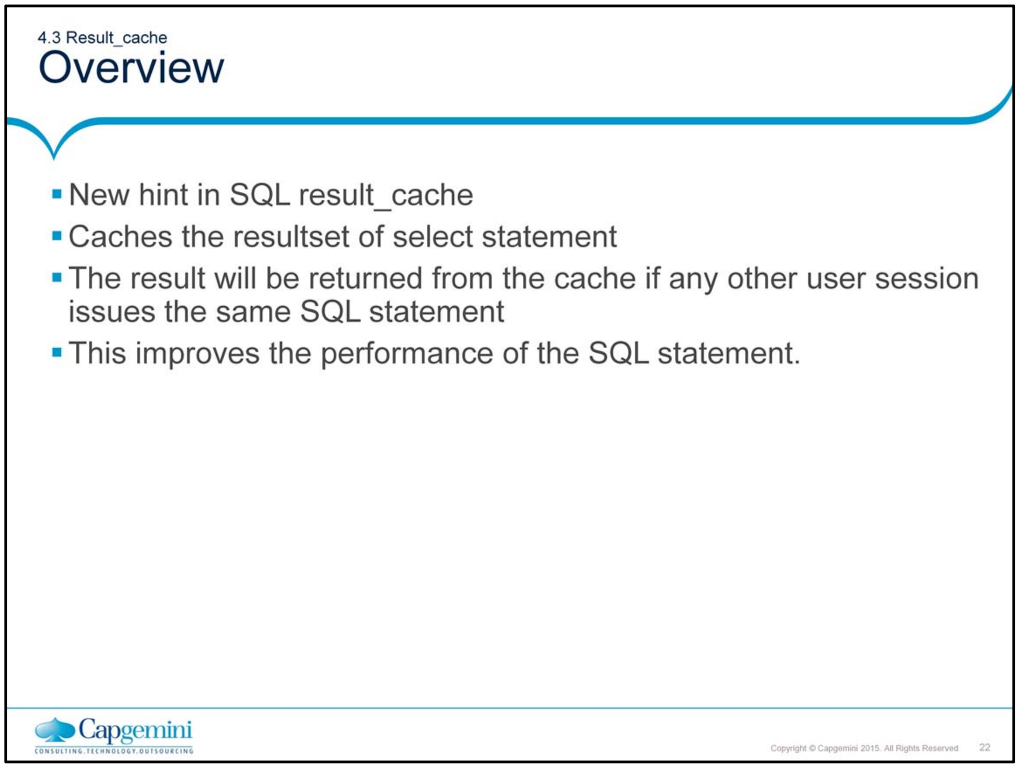
Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

```
SQL> Alter table emp read only;

Table altered.

SQL> update emp set sal=3000;
update emp set sal=3000
       *
ERROR at line 1:
ORA-12081: update operation not allowed on table "SCOTT"."EMP"
```

When we try to perform DML on the table, an error is produced.



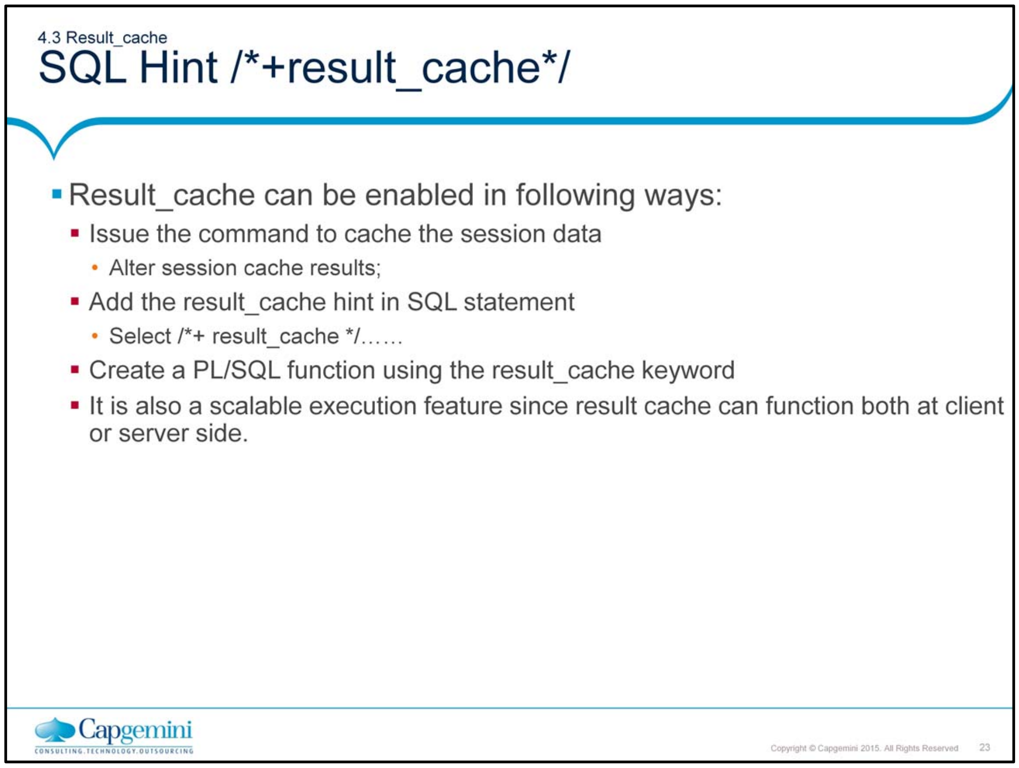
4.3 Result_cache

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.

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

The result_cache concept is quite similar to other existing Oracle concepts like shared PL/SQL collection or materialized view. Let us compare result_cache with bothMaterialized Views and Result_cache: In traditional materialized views tables are pre-joined and stored on slow disk whereas the result_cache hint stores the rows in super-fast RAM. In materialized view, Oracle created a transparent "query rewrite" mechanism to allow SQL to automatically re-write SQL to access the summary .The disk I/O is slower wheras the read from the memory is ought to be faster.
Shared PL/SQL and Result_cache: With Oracle PL/SQL the result sets can be saved in RAM for later use by the session using an array heap. The result_cache hint expands this functionality to allow for inter-session access to pre-summarized row data in RAM. Also, PL/SQL collections (arrays) were stored in the PGA, whereas the result_cache output is stored in the SGA region.

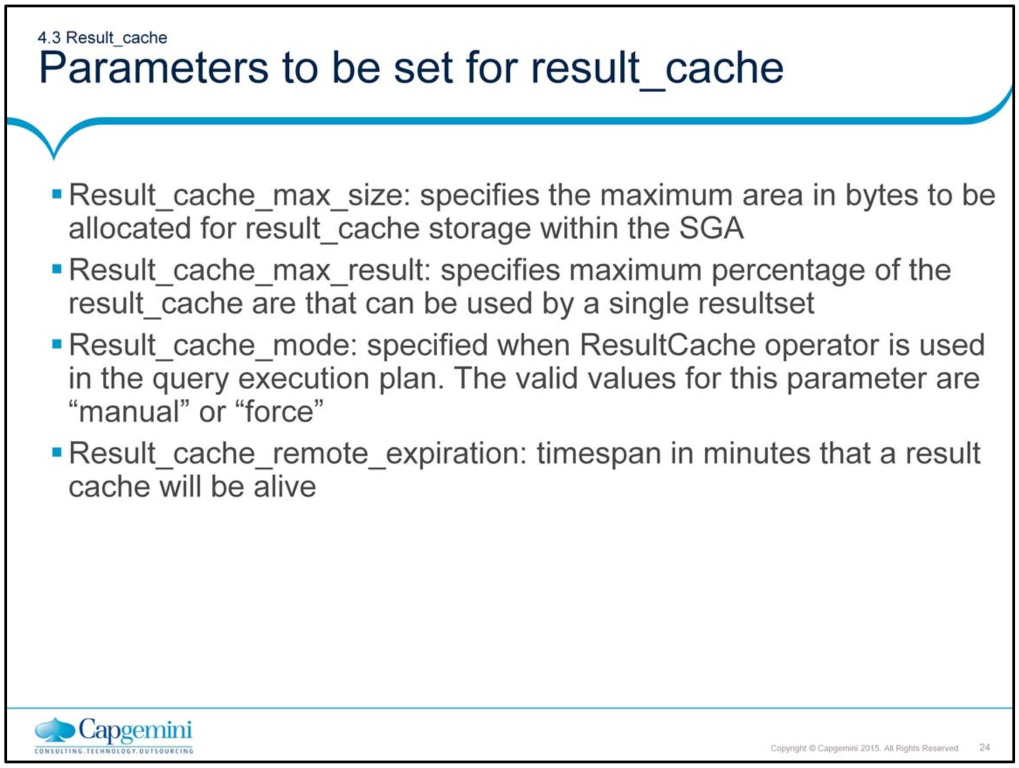


4.3 Result_cache

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.

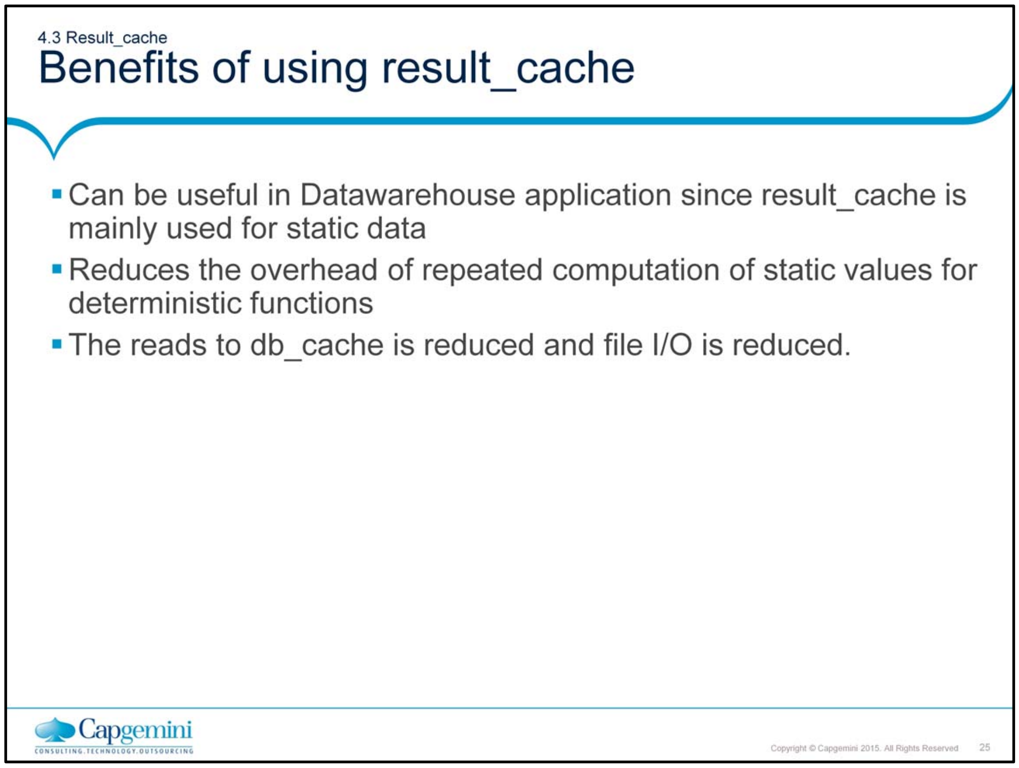
Using the result_cache keyword in PL/SQL function is much faster than repetitive querying.



4.3 Result_cache

Parameters to be set for result_cache

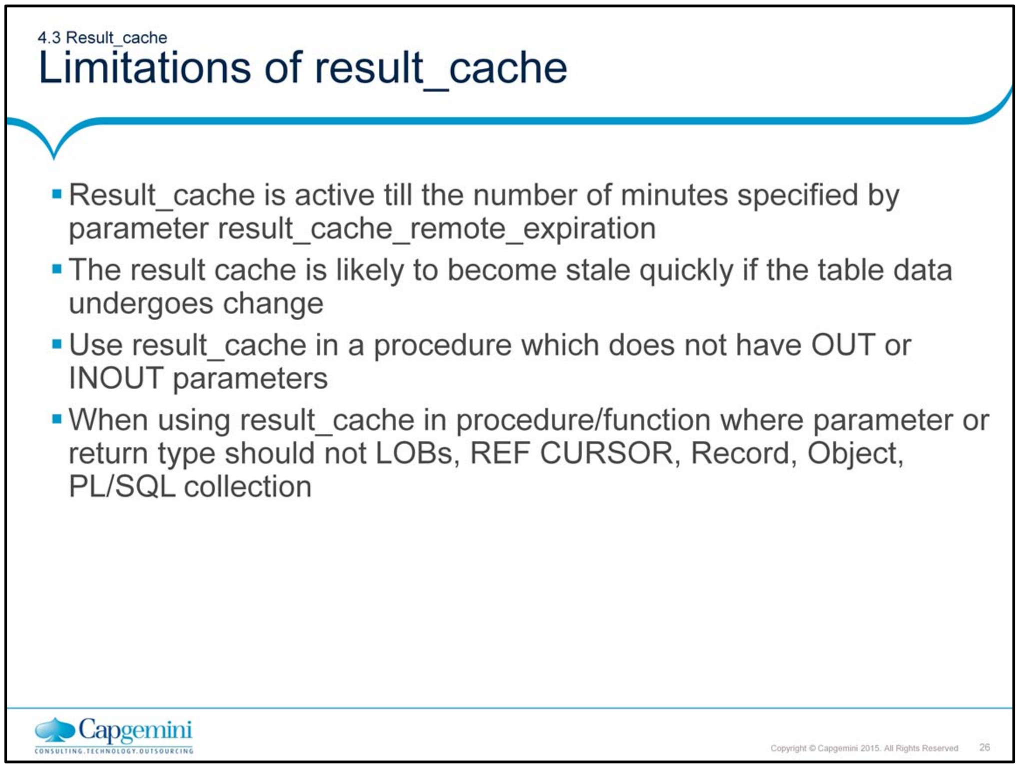
- Result_cache_max_size: specifies the maximum area in bytes to be allocated for result_cache storage within the SGA
- Result_cache_max_result: specifies maximum percentage of the result_cache are that can be used by a single resultset
- Result_cache_mode: specified when ResultCache operator is used 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.



4.3 Result_cache

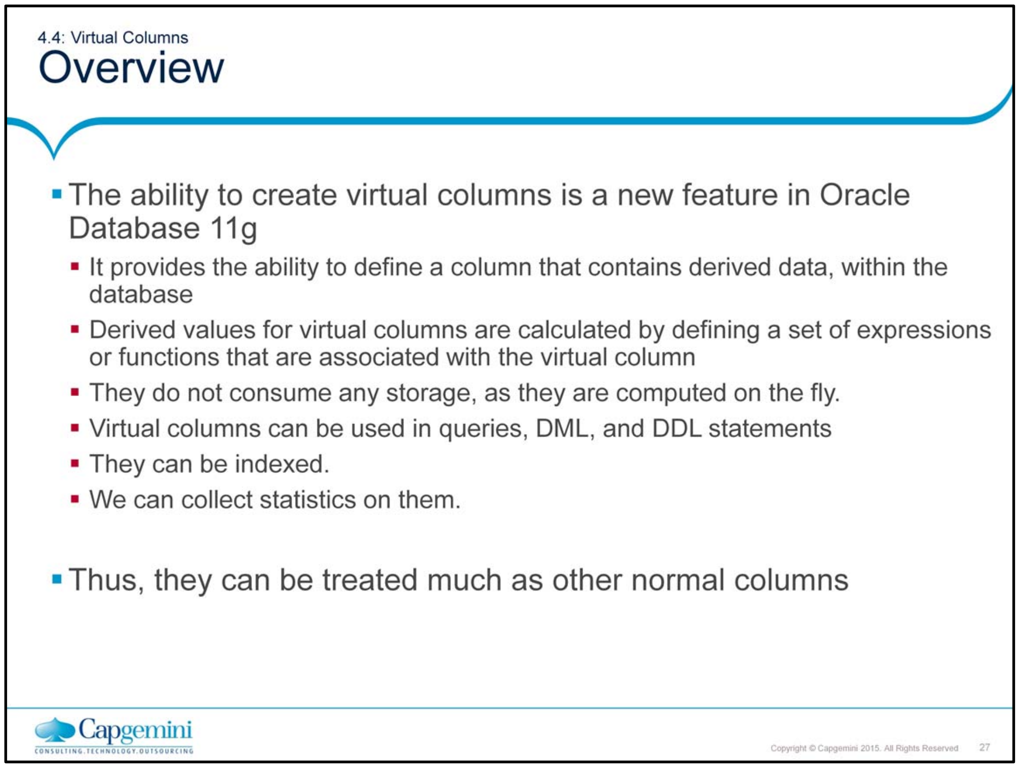
Limitations of result_cache

- 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

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

26

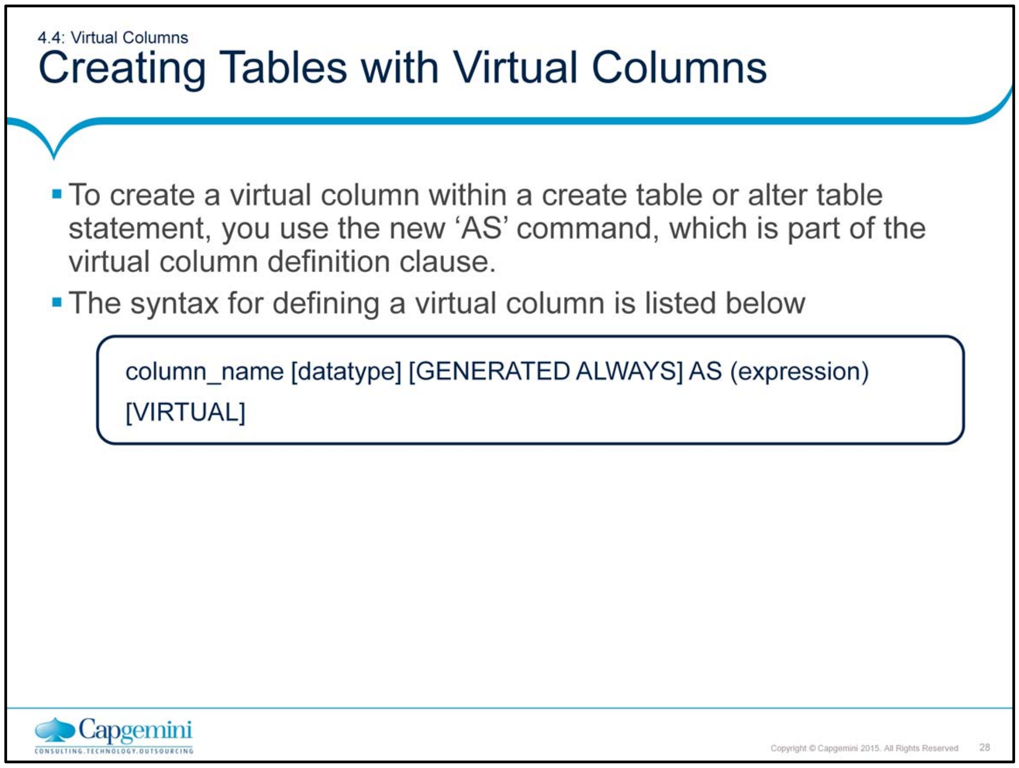
Tips on using result_cache be used:
Legacy databases: The databases which are highly normalized and frequent joins are required.
Mostly read-only tables in the database: Since the rows in result_cache get invalidated with change to the table data, it is best to be used with databases which mostly have read-only data in the tables.
Volatile databases: Do not use result_cache if the database is volatile or there are many DML operations being performed. In this case Materialized Views is preferable option.



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


4.4: Virtual Columns

Creating Tables with Virtual Columns

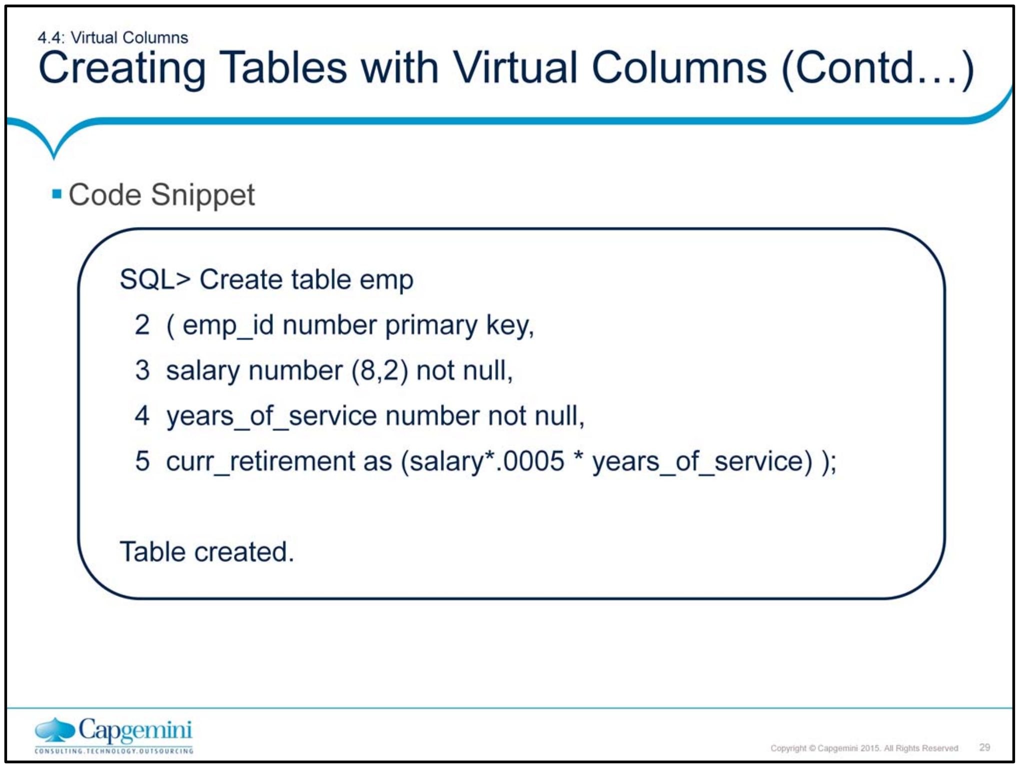
- To create a virtual column within a create table or alter table statement, you use the new ‘AS’ command, which is part of the virtual column definition clause.
- The syntax for defining a virtual column is listed below

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

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 28

If the datatype is omitted, it is determined based on the result of the expression. The GENERATED ALWAYS and VIRTUAL keywords are provided for clarity only.



4.4: Virtual Columns

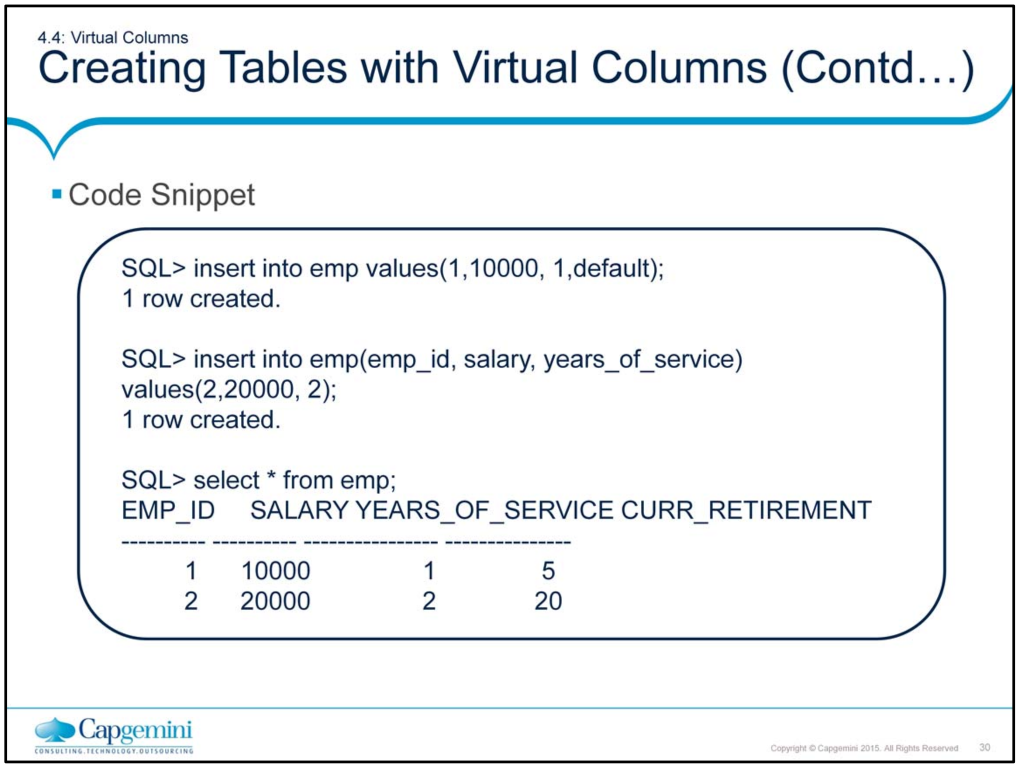
Creating Tables with Virtual Columns (Contd…)

- 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.
```

In this case, we are going to create an employee table that derives the current value of the employees’ retirement benefit, which is a formula based on the employee’s years of service, the total salary, and a multiple.



4.4: Virtual Columns

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
```

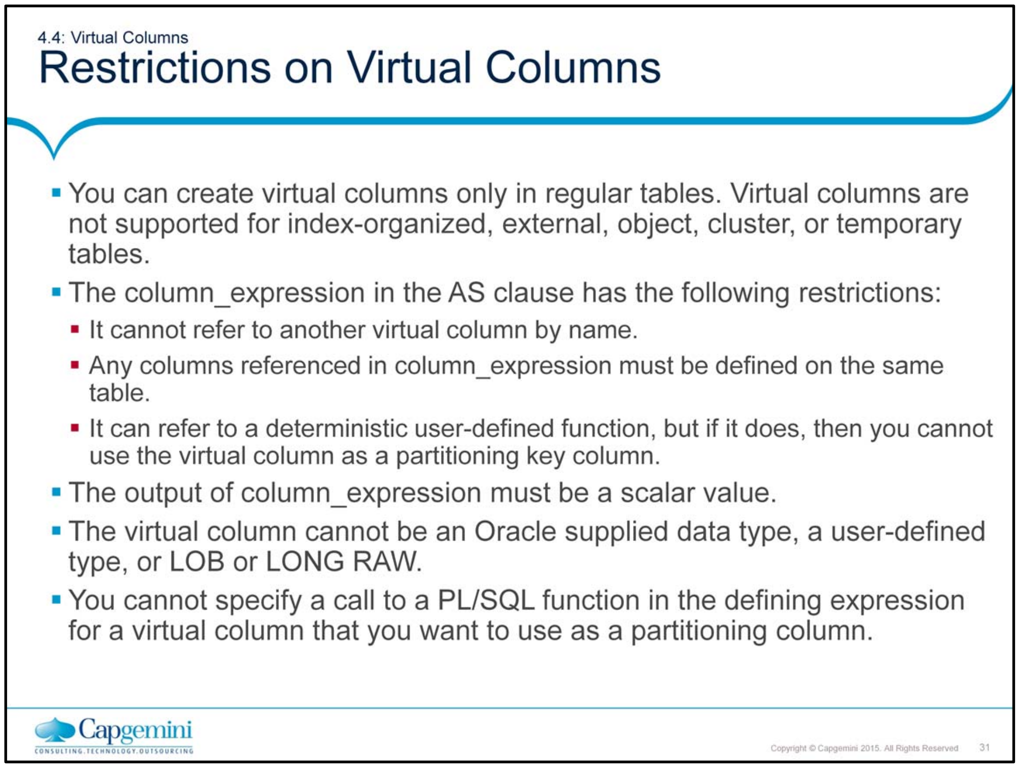
Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

You can also add a virtual column using the alter table command as seen in this example:

```
SQL>Alter table emp add
(curr_retirement as (salary*.0005 * years_of_service) );
```

The expression used to generate the virtual column is listed in the DATA_DEFAULT column of the [DBA|ALL|USER]_TAB_COLUMNS views.

```
select table_name, column_name, data_default from user_tab_columns where
table_name='EMP' and column_name='CURR_RETIREMENT';
TABLE_NAME COLUMN_NAME DATA_DEFAULT
---------------- ------------------ ------------------------------
EMP CURR_RETIREMENT "SALARY"*.0005*"YEARS_OF_SERVICE"
```

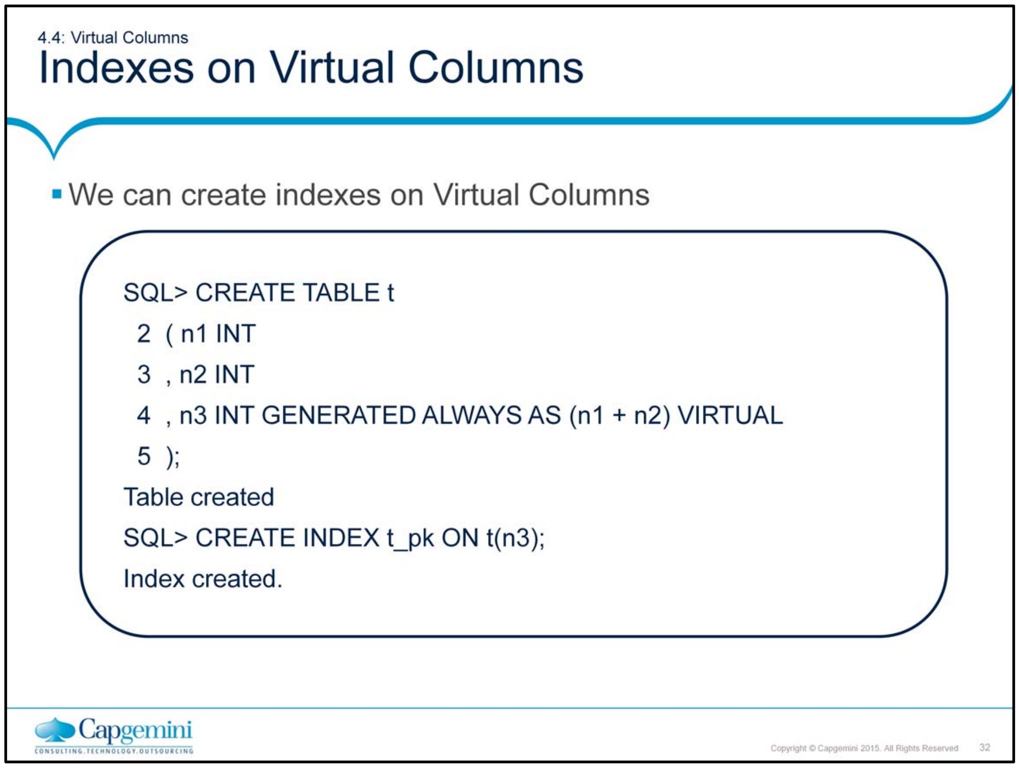


4.4: Virtual Columns

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.

31



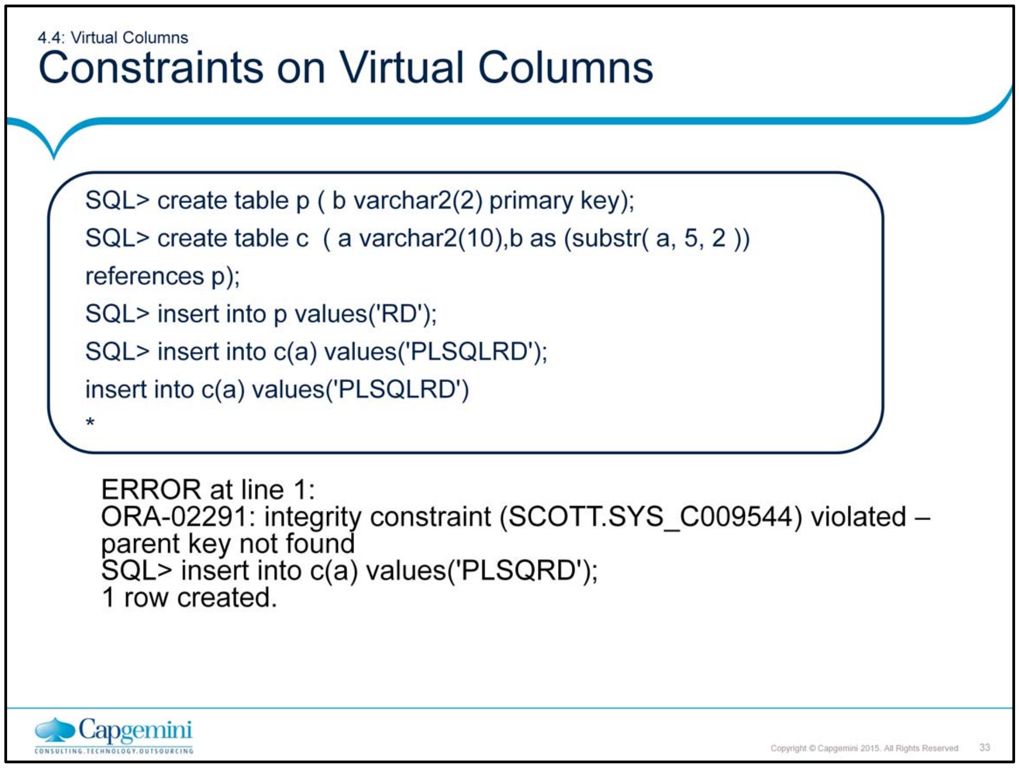
4.4: Virtual Columns

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.
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING



4.4: Virtual Columns

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.
```

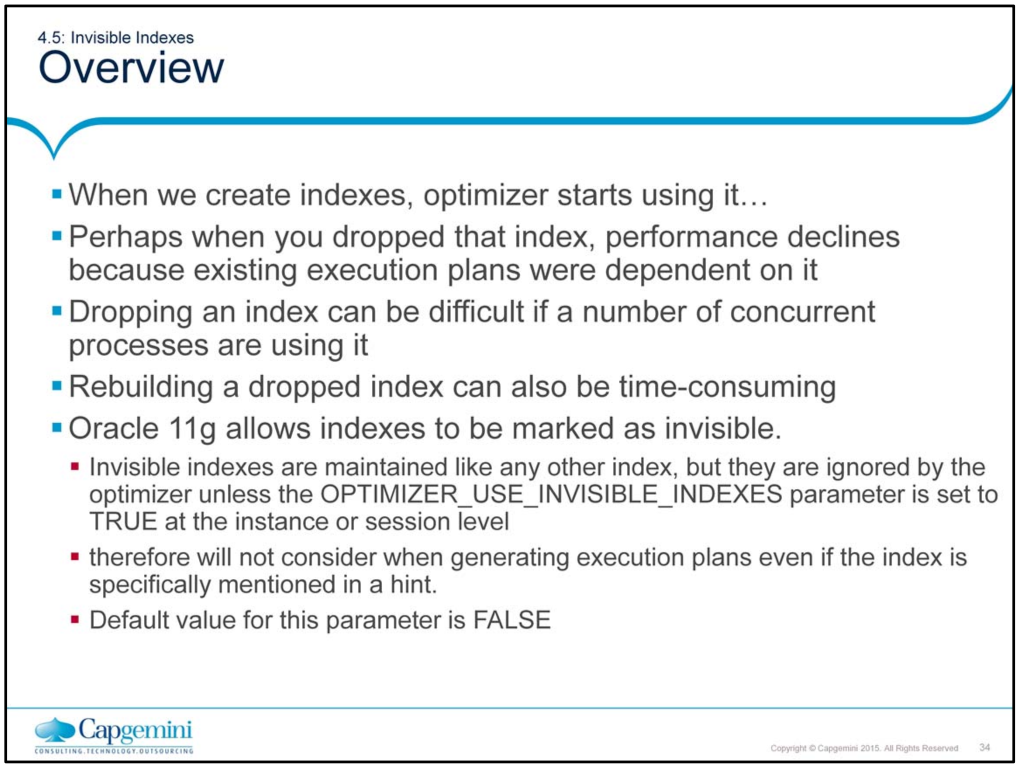
Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 33

```
SQL1> alter table c add constraint b_unique unique(b);
Table altered.

SQL> alter table c drop constraint b_unique;
Table altered.

SQL> alter table c add constraint c_pk primary key(b);
Table altered.
```

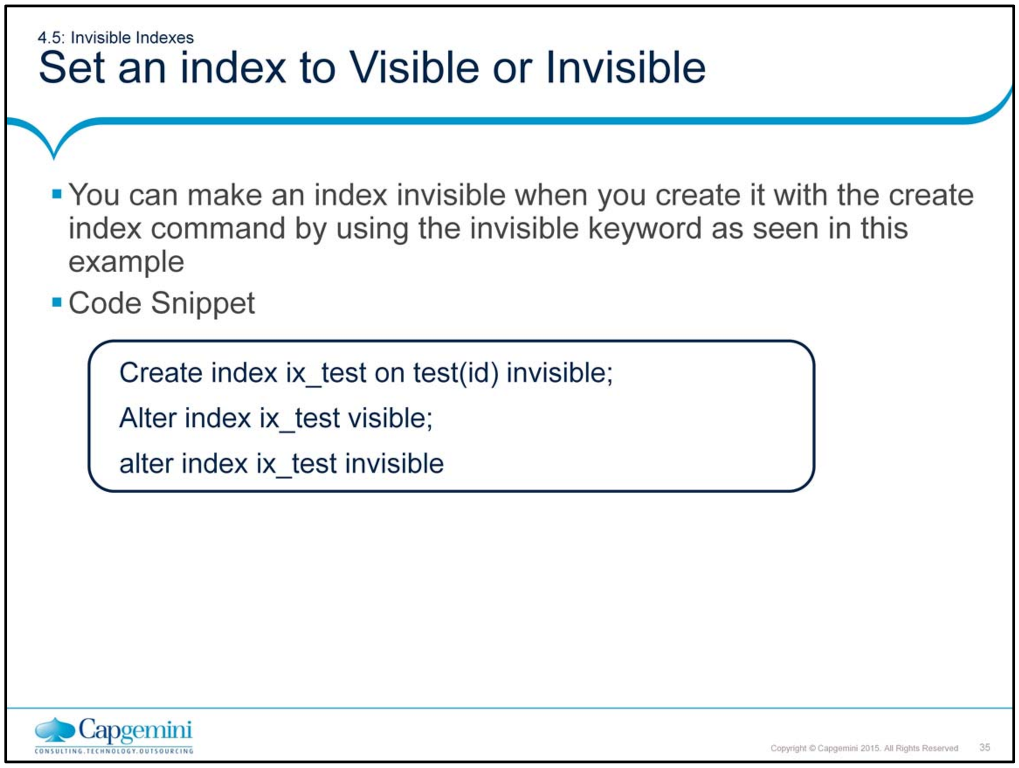


4.5: Invisible Indexes

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

You can make a new index invisible when you create it. You can then test it with representative SQL statements, including the new index in the hint of the SQL statement, and determine the impact of the SQL statements. Likewise, if you are planning on dropping an index, you can make it invisible. This will invalidate all shared SQL statements that have execution plans using that index, and the optimizer will stop considering that index for use.
Using invisible index, we can do the following:
Test the removal of an index before dropping it.
Use temporary index structure for certain operations or modules of an application before affecting the entire application.



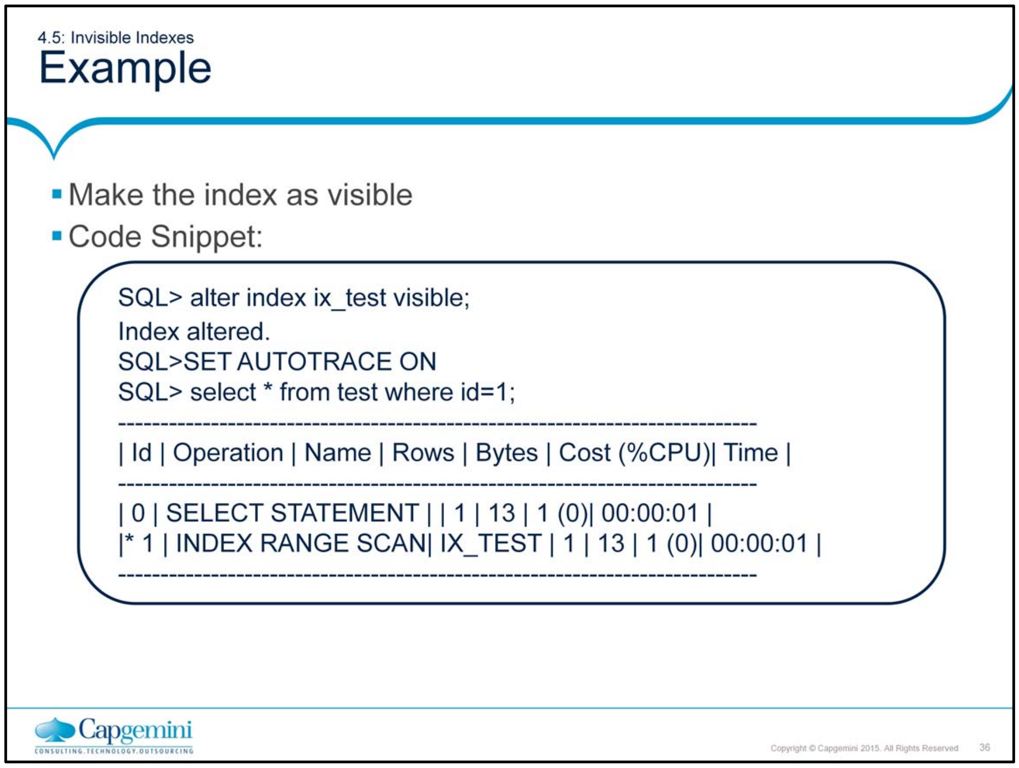
4.5: Invisible Indexes

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
```

You can also make an existing index visible or invisible using the invisible or visible keywords with the alter index command as seen in above examples.
You can override the invisible attribute of all indexes by setting the optimizer_ use_invisible_indexes parameter at the system or session level. Setting this parameter to TRUE will cause the optimizer to consider index usage regardless of the invisible setting.

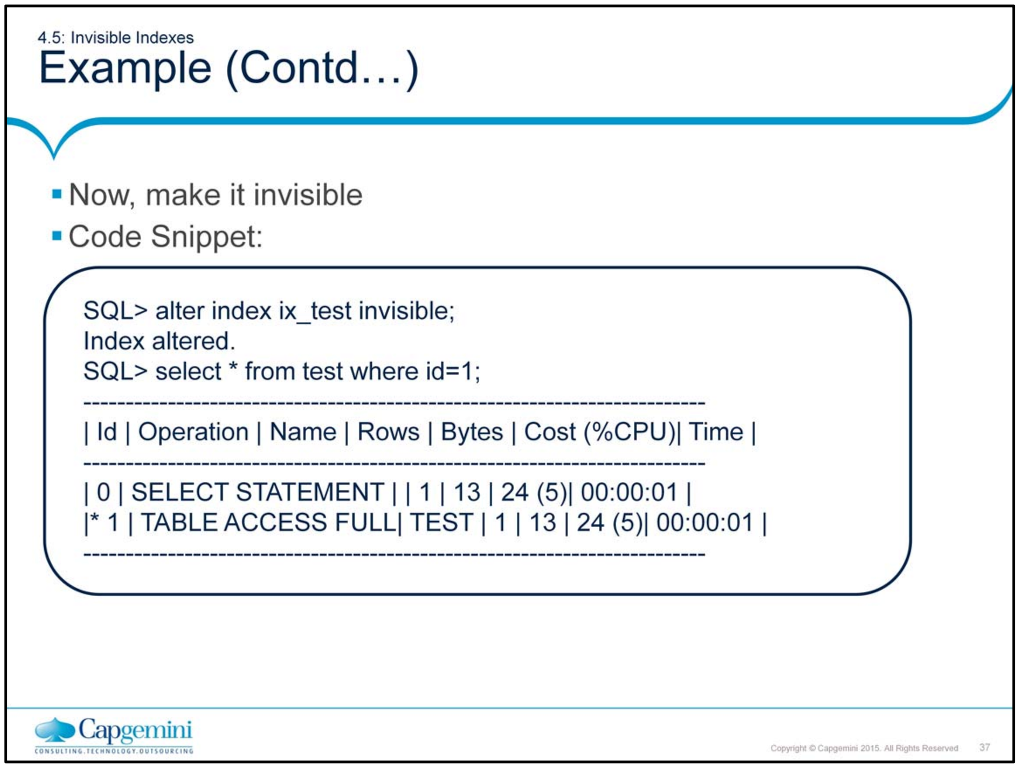


4.5: Invisible Indexes

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 |
-----------------------------------------------------------------------------
```



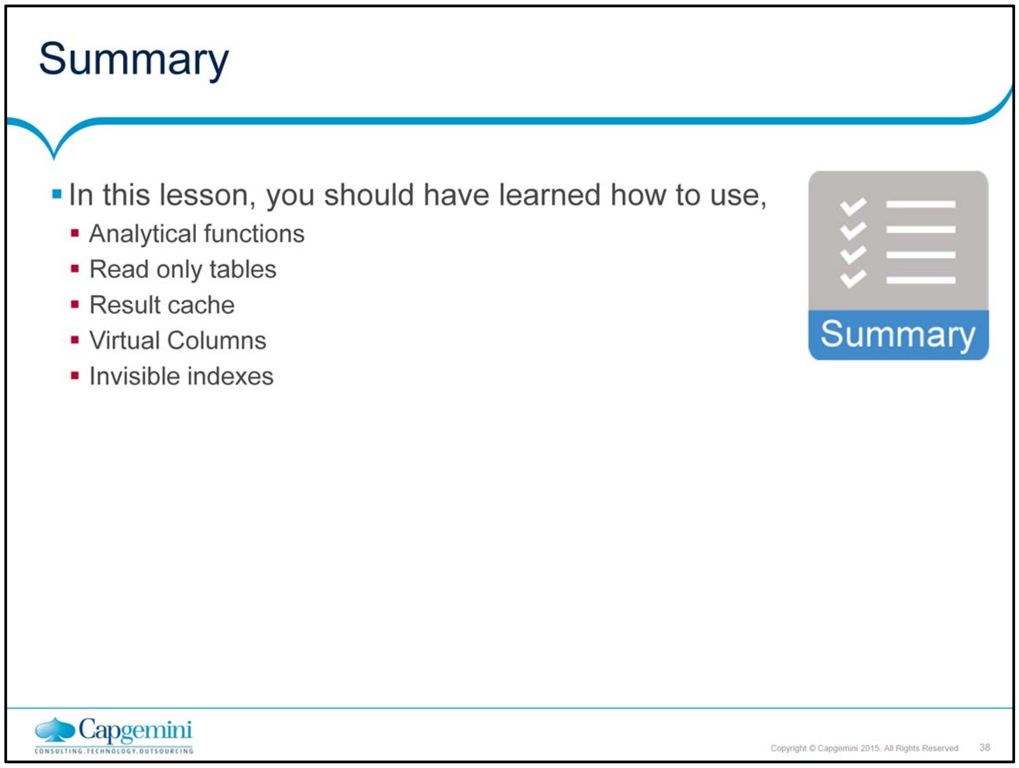
4.5: Invisible Indexes

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 |
---------------------------------------------------------------------------
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

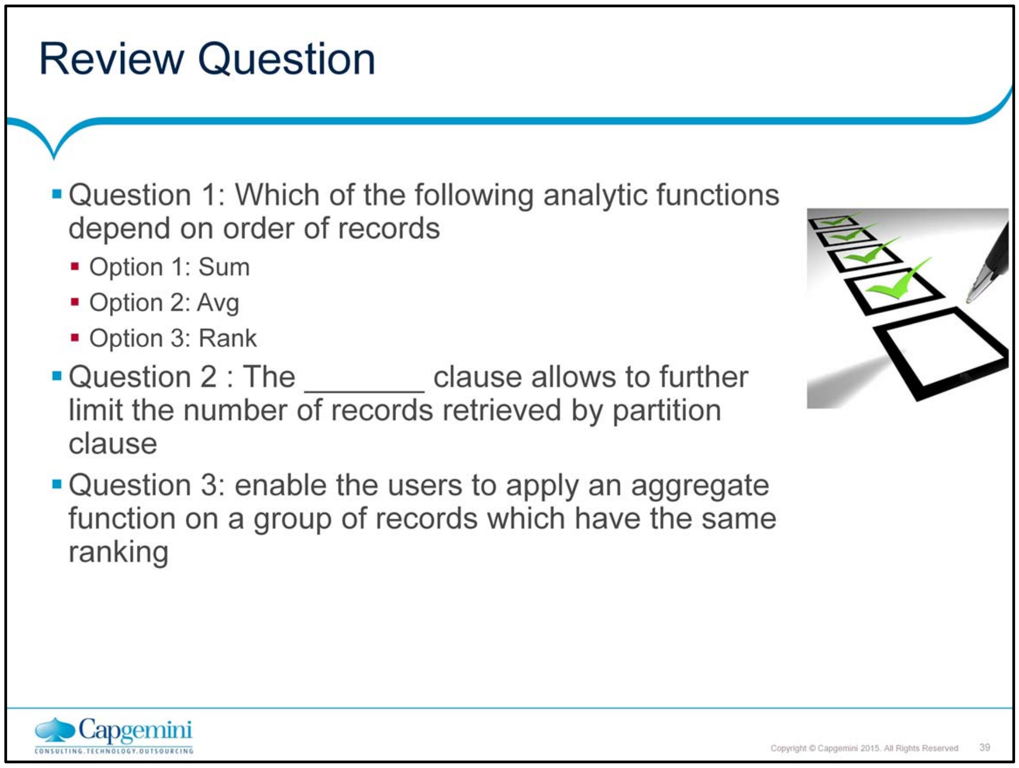


Summary

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

Summary

This lesson addressed some of the datetime functions available in the Oracle database.



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

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

39

Add the notes here.