Module n°4

Advanced SQL

1Z0-007

Auteur : Reda Benkirane

Advanced SQL – d September yyyy

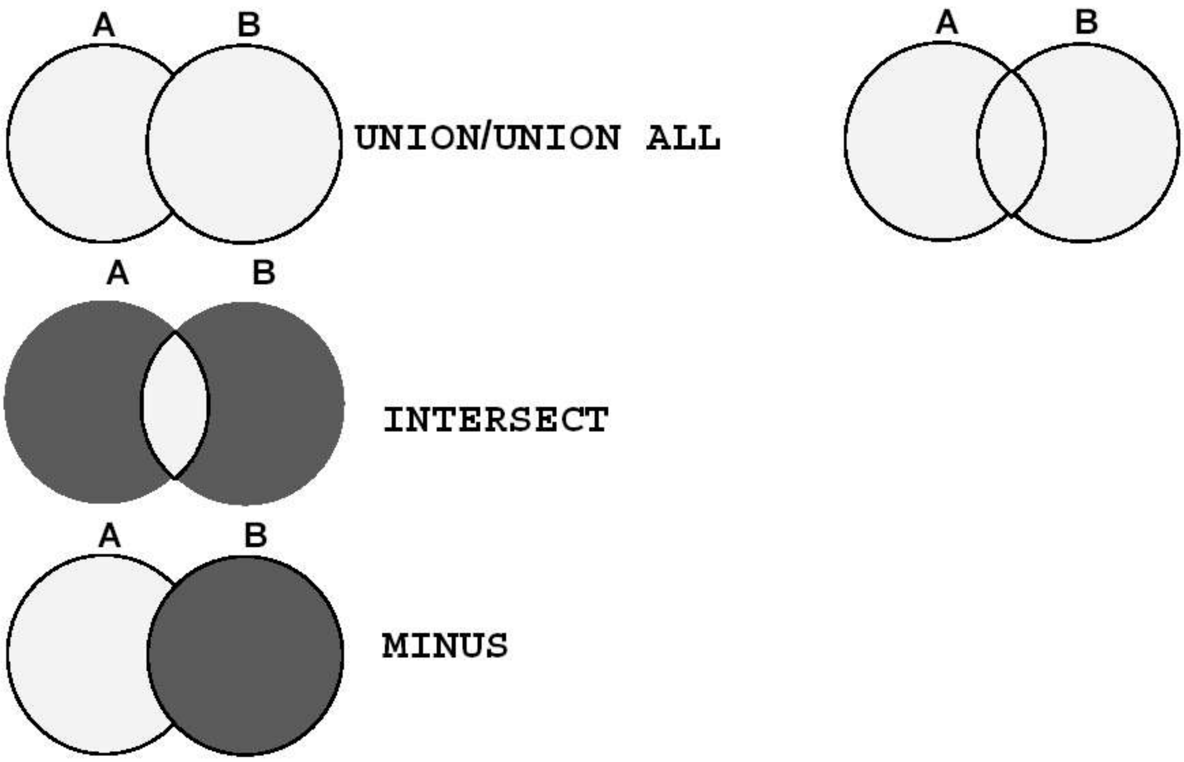
Nombre de pages : 47

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com

Summary

1.Using SET operators

1.1.The SET operators



The set operators combine the results of two or more component queries into one result. Queries containing set operators are called *compound queries*.

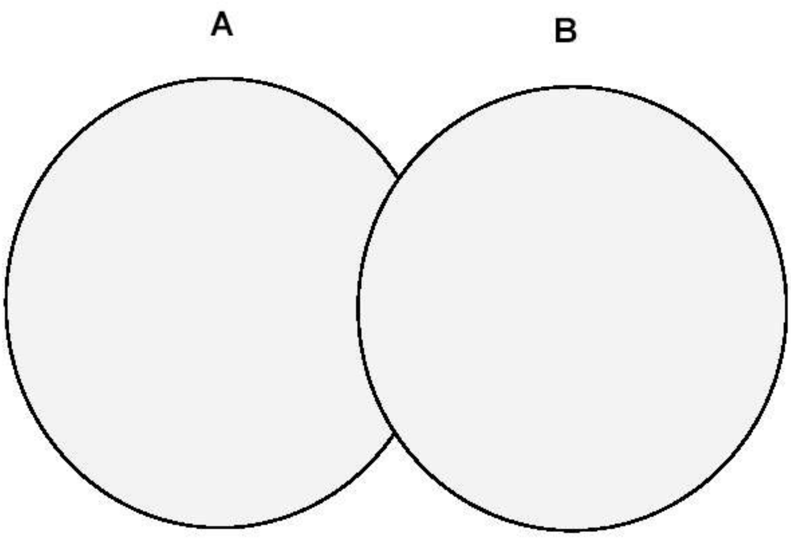
Operator	**Result**
INTERSECT	All distinct rows selected by both queries (**INTERSECT** combines two queries and returns only those rows from the first **SELECT** statements that are identical to at least one row from the second **SELECT** statement.)
UNION	All distinct rows selected by both query
UNION ALL	All rows selected by both query, including all duplicates
MINUS	All distinct rows selected by the first **SELECT** statement that are not produced in the second **SELECT** statement

```
SELECT     column1, column2, ...
FROM       table1
SET OPERATOR
SELECT     column1, column2, ...
FROM table2 ;
```

All set operators have equal precedence. If a SQL statement contains multiple set operators, the database evaluates them from left (top) to right (bottom) if no parentheses explicitly specify another order.
INTERSECT and **MINUS** operators are specific to Oracle.

1.2.UNION and UNION ALL

1.2.1.The UNION operator



The **UNION** operator combines results from multiple queries, but eliminates duplicate rows
Use the **UNION** operator to return all rows from multiple tables and eliminate any duplicate rows.

The number of columns and the data types of those columns must be identical in the two **SELECT** statements. The names of the columns need not to be identical.
UNION operates over all of the columns being selected.
NULL values are not ignored during duplicate checking.
Queries that use **UNION** in the **WHERE** clause must have the same number and type of columns in their **SELECT** list.
By default, the output is sorted in ascending order of the first column of the **SELECT** clause.

1.2.2.Using the UNION operator

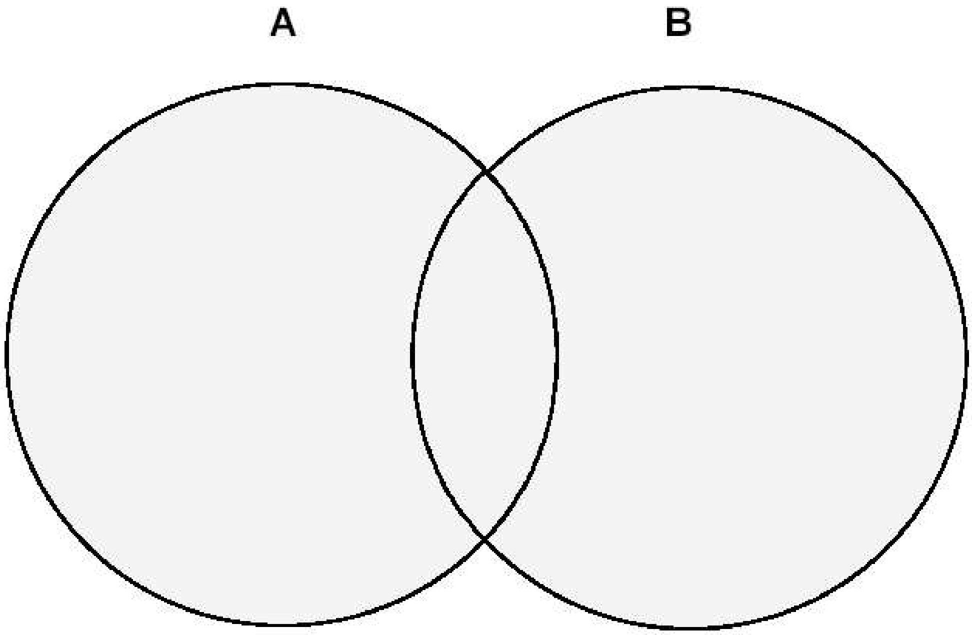
The **UNION** operator eliminates all duplicate rows.

Example:

```
SQL> SELECT       ename, job, sal
  2  FROM   emp
  3  UNION
  4  SELECT       name, title, 0
  5  FROM   emp_history;

ENAME      JOB              SAL
---------- --------- ----------
ADAMS      CLERK           1100
ALLEN      SALESMAN           0
ALLEN      SALESMAN        1600
BALFORD    CLERK              0
BLAKE      MANAGER         2850
…
23 row(s) selected
```

→ This statement displays all the results from the two queries, but common lines are not displayed. Also it displays a 0 when the line comes from the table that doesn't have a column for the salaries.

1.2.3.The UNION ALL operator



Use the **UNION ALL** operator to return all rows from multiple queries.
Unlike **UNION**, duplicate rows are not eliminated and the output is not sorted by default.
The **DISTINCT** keyword cannot be used

1.2.4.Using the UNION ALL operator

The **UNION ALL** operator returns results from both queries, including duplicate rows.

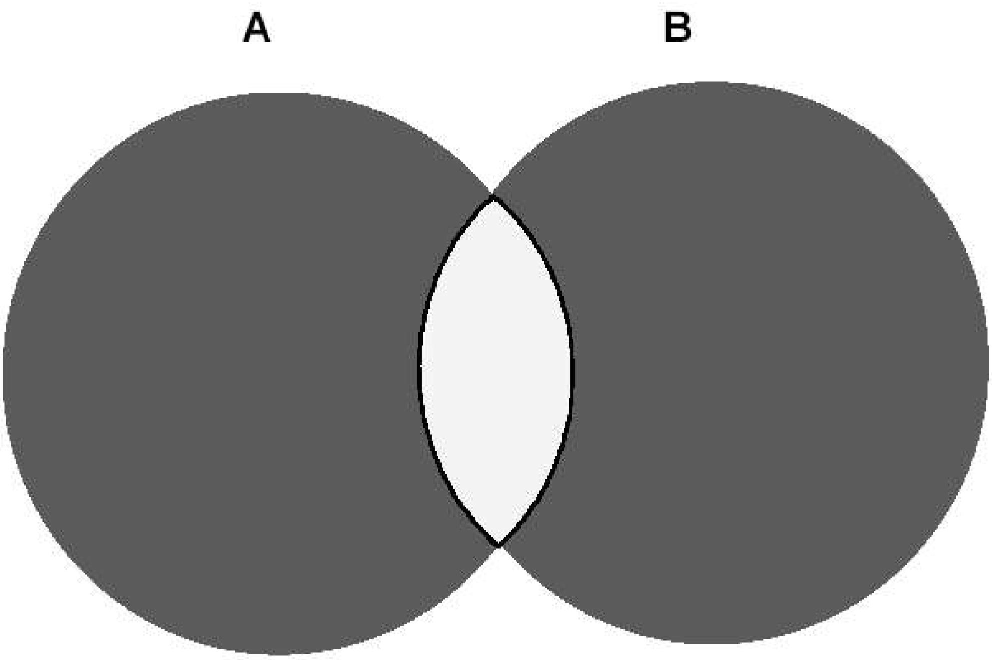
Example:

```
SQL> SELECT       ename, empno, job
  2  FROM         emp
  3  UNION ALL
  4  SELECT       name, empid, title
  5  FROM         emp_history
  6  ORDER BY     ename;

ENAME             EMPNO JOB
---------- ---------- ---------
ADAMS              7876 CLERK
ALLEN              7499 SALESMAN
ALLEN              7499 SALESMAN
BALFORD            6235 CLERK
BLAKE              7698 MANAGER
BRIGGS             7225 PAY CLERK
…
23 row(s) selected.
```

→ This statement displays all the results from the two queries, including the ones that are on the two tables.

1.3.INTERSECT

1.3.1.The INTERSECT operator



Use the **INTERSECT** operator to return all rows common to both queries.

The number of columns and the data types of those columns must be identical in the two **SELECT** statements. The names of the columns need not be identical.

Reversing the order of the intersected tables does not alter the result.
INTERSECT, like **UNION**, does not ignore **NULL** values.
Queries that use **INTERSECT** in the **WHERE** clause must have the same number and type of columns in their **SELECT** list.

1.3.2.Using the INTERSECT operator

Example:

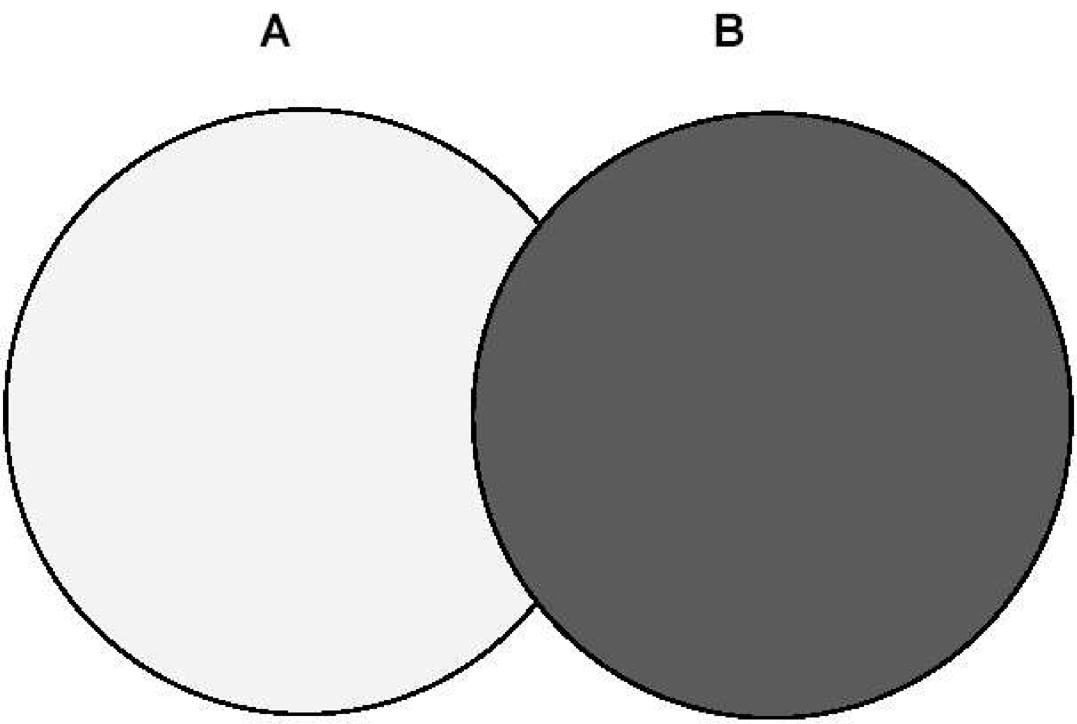
```
SQL>  SELECT      ename, empno, job
  2   FROM               emp
  3   INTERSECT
  4   SELECT      name, empid, title
  5   FROM               emp_history;

ENAME           EMPNO  JOB
---------- ---------- ---------
ALLEN            7499 SALESMAN
CLARK            7782 MANAGER
SCOTT            7788 ANALYST
```

→ This statement returns only the records that have the same values in the selected columns in both tables.

If you add a column in the **SELECT** list, the result may be different.

1.4.MINUS

1.4.1.The MINUS operator



Use the **MINUS** operator to return rows returned by the first query but not the second query (the first **SELECT** statement **MINUS** the second **SELECT** statement).
Like other operators, you must pay attention to the number of columns and their data type. Beware of the **WHERE** clause too.

1.4.2.Using the MINUS operator

Example:

```
SQL> SELECT               name, empid, title
  2  FROM          emp_history
  3  MINUS
  4  SELECT               ename, empno, job
  5  FROM          emp;

NAME            EMPID TITLE
---------- ---------- ---------
BALFORD          6235 CLERK
BRIGGS           7225 PAY CLERK
JEWELL           7001 ANALYST
SPENCER          6087 OPERATOR
VANDYKE          6185 MANAGER
WILD             7356 DIRECTOR

6 row(s) selected.
```

→ This statement returns the name, the employee id and their function that have quit the company (the employees from the EMP_HISTORY table minus the employees from the EMP table).

1.5.SET operator guidelines

1.5.1.SET operators rules

The expressions in the **SELECT** lists must match in number and data type. The names displayed in the result are the ones from the first **SELECT** list.
Duplicate rows are automatically eliminated except in **UNION ALL**.

The output is sorted in ascending order by default except in **UNION ALL**. The **ORDER BY** clause can be used, but only in the last query. The argument of the **ORDER BY** clause must match one of the names in the first **SELECT** list.
Column names from the first query appear in the result.

Parentheses can be used to alter the sequence of execution.
Queries that use set operator in their **WHERE** clause must have the same number and type of columns in their **SELECT** list.

1.5.2.Matching the SELECT statement

Because the expressions in the **SELECT** lists of compound queries must match in number and data type, you can use dummy columns and the data type conversion functions to comply with this rule. This can be done with the DUAL table and with various functions of data converting.

Example:

```
SQL> SELECT       deptno, TO_CHAR(null) AS location, hiredate
  2  FROM         emp
  3  UNION
  4  SELECT       deptno, loc, TO_DATE(null)
  5  FROM         dept;

    DEPTNO LOCATION      HIREDATE
---------- ------------- --------
        10 NEW YORK
        10               09/06/81
        10               17/11/81
        10               23/01/82
        20 DALLAS
        20               17/12/80
…
        30               28/09/81
        30               03/12/81

    DEPTNO LOCATION      HIREDATE
---------- ------------- --------
        40 BOSTON

18 row(s) selected.
```

→ This statement displays deptno, location and hiredate of all employees.

1.6.The Oracle Server and SET operators

When using **SET** operators, the Oracle Server eliminates automatically duplicate rows, except when using the **UNION ALL** operator.

By default, the result is sorted in ascending order of the first column of the **SELECT** statement.

The corresponding expressions in the **SELECT** lists of the component queries of a compound query must match in number and data type.

1.7.Controlling the order of rows

By default, the output is sorted in ascending order on the first column. You can use the **ORDER BY** clause to change this. The **ORDER BY** clause can be used only once in a compound query. If used, the **ORDER BY** clause must be placed at the end of the query. The **ORDER BY** clause accepts the column name, an alias, or the positional notation.

Example:

```
SQL> COLUMN       a_dummy NOPRINT
SQL> SELECT       'sing' AS "My dream", 3 a_dummy
  2  FROM         dual
  3  UNION
  4  SELECT       'I'd like to teach', 1
  5  FROM         dual
  6  UNION
  7  SELECT       'the world to', 2
  8  FROM         dual
  9  ORDER BY     2;

My dream
---------------------
I'd like to teach
the world to
sing
```

→ The column A_DUMMY is useful to sort the result to display a correct sentence. The **NOPRINT** operator is used to not display the column with the sort criteria.

2.Oracle Server datetime functions

2.1.Datetime functions

2.1.1.TZ_OFFSET

The **TZ_OFFSET** function returns the time zone offset corresponding to the value entered.
The value returned by this function is dependent on the date when the statement is executed.

For example, if the function returns -08:00, the value can be interpreted as the time zone from where the command was executed which is eight hours after UTC.

It's possible to enter a valid time zone name, a time zone offset from UTC, or the keyword SESSIONTIMEZONE or DBTIMEZONE.

Syntax:

TZ_OFFSET *([**'**time_zone_name'] '[+ | -] hh:mm']*
[SESSIONTIMEZONE] [DBTIMEZONE]);

Example:

- The time zone 'US/Eastern' is four hours behind UTC.

```
SQL> SELECT TZ_OFFSET('US/Eastern') FROM DUAL;

TZ_OFFS
-------
-04:00
```

- The time zone 'Canada/Yukon' is seven hours behind UTC.

```
SQL> SELECT TZ_OFFSET('Canada/Yukon') FROM DUAL;

TZ_OFFS
-------
-07:00
```

You can query the V$TIMEZONE-NAMES dynamic performance view, to list the valid time zone.

```
SQL> DESCRIBE v$timezone_names;

 Nom                                        NULL ?   Type
          -----------------------------------------        --------
----------------

 TZNAME                                              VARCHAR2(64)
 TZABBREV                                            VARCHAR2(64)
```

2.1.2.CURRENT_DATE

The **CURRENT_DATE** function returns the current date in the session's time zone.
Notice that the returned value is a date in the Gregorian calendar.

The following examples illustrate that **CURRENT_DATE** is sensitive to the session time zone.
In the first statement, the session is altered to set the **TIME_ZONE** parameter to -5:0.
The **TIME_ZONE** is a session parameter only, not an initialization parameter, which is set as follows:

TIME_ZONE = *'[+ | -] **hh:mm';***

The format mask indicates the hours and minutes before or after UTC (Greenwich Mean Time)
The **CURRENT_DATE** value changes when the **TIME_ZONE** parameter value is changed.

The **ALTER SESSION** command sets the date format of the session to 'DD-MON-YYY HH24: MI: SS'.

```
SQL> ALTER SESSION
  2  SET NLS_DATE_FORMAT = 'DD-MON-YYY HH24:MI:SS';

Session altered.

SQL> ALTER SESSION
  2  SET TIME_ZONE ='-5:0';

Session altered.

SQL> SELECT SESSIONTIMEZONE, CURRENT_DATE
  2  FROM DUAL;

SESSIONTIMEZONE          CURRENT_DATE
-------------------      ----------------------
-05:00                   31-AOU-004 04:29:20


SQL> ALTER SESSION SET TIME_ZONE = '-8:0';

Session altered.

SQL> SELECT SESSIONTIMEZONE, CURRENT_DATE
  2  FROM DUAL;

SESSIONTIMEZONE          CURRENT_DATE
---------------------    ----------------------
-08:00                   31-AOU-004 01:29:52
```

2.1.3.CURRENT_TIMESTAMP

This function returns the current date and time in the session time zone, as a value of the data type **TIMESTAMP WITH TIME ZONE.**

Syntax:

CURRENT_TIMESTAMP (*precision***)***;*

Precision: Is an optimal argument that specifies the fractional second precision of the time value returned. Default precision is 6.

The following examples illustrate that **CURRENT_TIMESTAMP** is sensitive to the session time zone.
First, the session is altered to set the **TIME_ZONE** parameter to -5:0.
Note that the **CURRENT_TIMESTAMP** value changes when the **TIME_ZONE** parameter is changed.

```
SQL> ALTER SESSION
  2  SET TIME_ZONE='-5:0';

Session altered.

SQL> SELECT SESSIONTIMEZONE,CURRENT_TIMESTAMP
  2  FROM DUAL;

SESSIONTIMEZONE         CURRENT_TIMESTAMP
--------------------    -------------------------------
-05:00                  31/08/04 04:39:23,817000 -05:00

SQL> ALTER SESSION
  2  SET TIME_ZONE='-8:0';

Session altered.

SQL> SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP
  2  FROM DUAL;

SESSIONTIMEZONE         CURRENT_TIMESTAMP
------------------      --------------------------------
-08:00                  31/08/04 01:40:28,319000 -08:00
```

2.1.4.LOCALTIMESTAMP

The **LOCALTIMESTAMP** function returns the current date and time in the session time zone as a value of data type **TIMESTAMP.**

The difference between this function and the **CURRENT_TIMESTAMP** function is that it returns a **TIMESTAMP** value.

Syntax:

LOCAL_TIMESTAMP (*TIMESTAMP_precision)***;**

TIMESTAMP_precision: Is an optimal argument that specifies the fractional second precision of the **TIMESTAMP** value returned by the function.

These examples illustrate the difference between **LOCALTIMESTAMP** and **CURRENT_TIMESTAMP.**

Note that the **LOCALTIMESTAMP** does not display the time zone value.

```
SQL> ALTER SESSION
  2  SET TIME_ZONE='-5:0';

Session altered.

SQL> SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP
  2  FROM DUAL;

CURRENT_TIMESTAMP                      LOCALTIMESTAMP
------------------------------         ------------------------------
31/08/04 04:47:23,217000 -05:00        31/08/04 04:47:23,217000


SQL> ALTER SESSION
  2  SET TIME_ZONE='-8:0';

Session altered.

SQL> SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP
  2  FROM DUAL;

CURRENT_TIMESTAMP                      LOCALTIMESTAMP
------------------------------         ------------------------------
31/08/04 01:48:00,304000 -08:00        31/08/04 01:48:00,304000
```

2.1.5.DBTIMEZONE and SESSIONTIMEZONE

The default database time zone is the same as the one of the operating system.
This time zone is set by specifying the **SET TIME_ZONE** clause in the **CREATE DATABASE** statement.

The **DBTIMEZONE** function returns the value of the database time zone.
The value returned is time zone offset, or a time zone region name, depending on how the user specified the database time zone value in the most recent **CREATE DATABASE** or **ALTER DATABSE** statement.

```
SQL> SELECT DBTIMEZONE FROM DUAL;

     DBTIME
     ------
     +00:00
```

The **SESSIONTIMEZONE** function returns the value of the current session's time zone.
The value returned in a time zone offset, or a time zone region name, depends on how the user specified the database time zone value in the most recent **CREATE DATABASE** or **ALTER DATABASE** statement.

Note that the database time zone is different from the current session's time zone.

```
SQL> SELECT          SESSIONTIMEZONE
  2  FROM            DUAL;

SESSIONTIMEZONE
---------------
-08:00
```

2.1.6.EXTRACT

The **EXTRACT** expression extracts and returns the value of a specified datetime field from a datetime or interval value expression.
The user can extract any of components mentioned in the following syntax.

Syntax:

SELECT EXTRACT (*[YEAR] [MONTH] [DAY] [HOUR] [MINUTE]*
[SECOND] [TIMEZONE_HOUR] [TIMEZONE_MINUTE] [TIMEZONE_REGION] [TIMEZONE_ABBR]

FROM *[datetime_value_expression] [interval_value_expression]***);**

When extracting a **TIMEZONE_REGION** or **TIMEZONE_ABBR**, the value returned is a string containing the appropriate time zone name or abbreviation.
When extracting any other values the value returned is in the UTC format.

In the following statement, the **EXTRACT** function is used to extract the YEAR from **SYSDATE.**

```
SQL> SELECT        EXTRACT (YEAR FROM SYSDATE)
     FROM          DUAL;

EXTRACT(YEARFROMSYSDATE)
------------------------
                    2004
```

In this example, the **EXTRACT** function is used to extract the MONTH from HIREDATE column of the EMP table, for all the employees.

```
SQL> SELECT     ename, hiredate,
  2             EXTRACT(MONTH FROM hiredate)
  3  FROM       emp;

ENAME      HIREDATE            EXTRACT(MONTHFROMHIREDATE)
---------- ------------------- --------------------------
SMITH      17-DEC-980 00:00:00                         12
ALLEN      20-FEV-981 00:00:00                          2
WARD       22-FEV-981 00:00:00                          2
JONES      02-AVR-981 00:00:00                          4
MARTIN     28-SEP-981 00:00:00                          9
BLAKE      01-MAI-981 00:00:00                          5
CLARK      09-JUN-981 00:00:00                          6
SCOTT      09-DEC-982 00:00:00                         12
KING       17-NOV-981 00:00:00                         11
TURNER     08-SEP-981 00:00:00                          9
ADAMS      12-JAN-983 00:00:00                          1

ENAME      HIREDATE            EXTRACT(MONTHFROMHIREDATE)
---------- ------------------- --------------------------
JAMES      03-DEC-981 00:00:00                         12
FORD       03-DEC-981 00:00:00                         12
MILLER     23-JAN-982 00:00:00                          1
GEROGES    26-AOU-004 11:24:58                          8

15 row(s) selected.
```

2.2.Conversions

2.2.1.TIMESTAMP conversion using FROM_TZ

The **FROM_TZ** function is used to convert a **TIMESTAMP** value to a **TIMESTAMP WITH TIME ZONE** value.

Syntax:

FROM_TZ (TIMESTAMP *timestamp_value, time_zone_value);*

Time_zone_value: Is a character string in format 'TZH: TZM' or a character expression that returns a string **TZR** with optional **TZD** format.

TZR represents the time zone region in a datetime input strings.

The following example converts a **TIMESTAMP** value to a **TIMESTAMP WTH TIME ZONE VALUE.**

```
SQL> SELECT FROM_TZ(TIMESTAMP
  2                     '2000-03-28 08:00:00',
  3                     'Australia/North')
  4  FROM   DUAL;


FROM_TZ(TIMESTAMP'2000-03-2808:00:00','AUSTRALIA/NORTH')
-------------------------------------------------------------------
-
28/03/00 08:00:00,000000000 AUSTRALIA/NORTH
```

2.2.2.STRING to TIMESTAMP using TO_TIMESTAMP and TO_TIMESTAMP_TZ

The **TO_TIMESTAMP** function converts a string of CHAR, VARCHAR, NCHAR, or NVARCHAR2 data type to a value of TIMESTAMP data type.

Syntax:

TO_TIMESTAMP (*CHAR, [fmt], ['nlsparam']***);**

Fmt: Specifies the format of CHAR. If you omit it, the string must be in default format on the TIMESTAMP data type.

Nlsparam: Specifies the language in which the month and day names and abbreviations are returned. It can have this form: 'NLS_DATE_LANGUAGE= language'.
If omitted, the function uses the default date language for your session.

```
SQL> SELECT     TO_TIMESTAMP('2000-12-01 11:00:00',
  2                       'YYYY-MM-DD HH:MI:SS') AS tmp_conversion
  3  FROM       DUAL;

TMP_CONVERSION
-----------------------------------------------------------------
01/12/00 11:00:00,000000000
```

The **TO_TIMESTAMP_TZ** function converts a string of CHAR, VARCHAR, NCHAR, or NVARCHAR2 data type to a value of TIMESTAMP WITH TIME ZONE data type.

Syntax:

TO_TIMESTAMP_TZ (*CHAR, [fmt], ['nlsparam']***);**

Fmt: Specifies the format of CHAR. If you omit it, the string must be in default format on the TIMESTAMP WITH TIME ZONE data type.

Nlsparam: Specifies the language in which the month and day names and abbreviations are returned. It can have this form: 'NLS_DATE_LANGUAGE= language'.
If omitted, the function uses the default date language for your session.

```
SQL> SELECT       TO_TIMESTAMP_TZ('1999-12-01 11:00:00 -8:00',
  2             'YYYY-MM-DD HH:MI:SS TZH:TZM') AS tz_conversion
  3  FROM         DUAL;

TZ_CONVERSION
-------------------------------------------------------------------
-
01/12/99 11:00:00,000000000 -08:00
```

2.2.3.Time interval conversion with TO_YMINTERVAL

The **TO_YMINTERVAL** function is used to convert a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to an INTERVAL YEAR TO MONTH data type.
This data type stores a period of time using the YEAR and MONTH datetime fields.

Syntax:

TO_YMINTERVAL (*char*);

Char: Is the character string to be converted.

The following statement calculates a date that is one year two months after the hire date for the employees working in the department 20 of the EMP table.

```
SQL> SELECT     hiredate,
  2             hiredate + TO_YMINTERVAL('01-02') AS
  3             hire_date_YMIN
  4  FROM       emp
  5  WHERE      deptno=20;

HIREDATE                 HIRE_DATE_YMIN
-------------------      -------------------
17-DEC-1980 00:00:00     17-FEV-1982 00:00:00
02-AVR-1981 00:00:00     02-JUN-1982 00:00:00
09-DEC-1982 00:00:00     09-FEV-1984 00:00:00
12-JAN-1983 00:00:00     12-MAR-1984 00:00:00
03-DEC-1981 00:00:00     03-FEV-1983 00:00:00
```

3.GROUP BY with ROLLUP and CUBE operators

3.1.Reviews of GROUP BY and HAVING

3.1.1.Review of group functions

See. SQLP course “Module 2: Techniques of data retrieval”.

3.1.2.Review of the GROUP BY clause

See. SQLP course “Module 2: Techniques of data retrieval”.

3.1.3.Review of the HAVING clause

See. SQLP course “Module 2: Techniques of data retrieval”.

3.2.ROLLUP

3.2.1.The ROLLUP operator

The **ROLLUP** operator is an extension of the **GROUP BY** clause that can produce cumulative aggregates such as sub-totals.

```
SELECT      column, group_function
FROM        table
[WHERE      condition]
[GROUP BY   [ROLLUP] (group_by_expression)];
```

The **ROLLUP** operator creates groups by moving in one direction, from right to left, along the list of columns specified in the **GROUP BY** clause. It then applies the aggregate function to these groups.

3.2.2.Example

The **ROLLUP** operator creates subtotals that roll up from the most detailed level to a grand total, following the grouping list specified in the **GROUP BY** clause. First it computes the standard aggregate values for the groups specified in the **GROUP BY** clause. Then it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns.

Given two expressions in the **ROLLUP** operator of the **GROUP BY** clause, the operation results in n+1=2+1=3 groupings.

Rows based on the values of the first n expressions are called rows or regular rows and the others are called super aggregate rows.

Example:

```
SQL> SELECT       deptno,job, SUM(sal)
  2  FROM         emp
  3  GROUP BY     ROLLUP(deptno,job);

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        10                 8750
        20 ANALYST         6000
        20 CLERK           1900
        20 MANAGER         2975
        20                10875
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        5600

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        30                 9400
                          29025

13 row(s) selected.
```

-> This statement displays the sum of all salaries for each job in each department, as well as sum of all salaries for each department and for all of them.

3.3.CUBE

3.3.1.The CUBE operator

The **CUBE** operator is an additional switch in the **GROUP BY** clause in the **SELECT** statement. The **CUBE** operator can be applied to all aggregate functions.

```
SELECT      column, group_function
FROM        table
[WHERE      condition]
[GROUP BY   [CUBE] (group_by_expression)];
```

While **ROLLUP** produces only a fraction of possible subtotal combinations, **CUBE** produces subtotals for all possible combinations of groupings specified in the **GROUP BY** clause, and a grand total.
All the returned values are calculated from the specified aggregate group in the **SELECT** list.

3.3.2.Example

CUBE operator returns the same result as **ROLLUP** operator, but its group function can be applied to subgroup also.
The number of additional groups in the result is determined by the number of columns in the **GROUP BY** clause, because each columns combination is used to produce supersets. Then if there is *n* column (s) or expressions in the **GROUP BY** clause, there will be 2^n possible combinations supersets.

Example:

```
SQL> SELECT       deptno, job, SUM(sal)
  2  FROM         emp
  3  GROUP BY CUBE(deptno, job);

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        10                 8750
        20 ANALYST         6000
        20 CLERK           1900
        20 MANAGER         2975
        20                10875
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        5600
        30                 9400
           ANALYST         6000
           CLERK           4150
           MANAGER         8275
           PRESIDENT       5000
           SALESMAN        5600
                          29025
```

➔ This statement returns the same result as the **ROLLUP** operator. The sum of the salary has been added.

3.4. Analytical functions

3.4.1.Describe analytical functions

To ease advanced SQL programming, analytical functions were introduced since Oracle8*i* Release 2, to simplify calculations such as average variables and classifications.
Groups defined by the **GROUP BY** clause of a **SELECT** are called partitions. The result of a statement can be composed by a partition with all the lines, some big partitions, or many small partitions with few lines each. Analytical functions are applied to each line of each partition.

3.4.2.RANK function

The **RANK** function creates a classification starting with 1.

```
SELECT      column,group_function(argument),
        RANK() OVER([PARTITION BY column] ORDER BY
        group_function(argument [DESC])
FROM table
GROUP BY    column
```

Example:

```
SQL> SELECT             deptno, job, SUM(sal),
  2              RANK() OVER(PARTITION BY deptno
  3              ORDER BY SUM(sal) DESC)
  4              AS rank_of_job_per_dep,
  5              RANK() OVER(ORDER BY SUM(sal) DESC)
  6              AS rank_of_sumsal
  7   FROM       emp
  8   GROUP BY          deptno, job
  9   ORDER BY          deptno;


    DEPTNO JOB         SUM(SAL) RANK_OF_JOB_PER_DEP RANK_OF_SUMSAL
---------- --------- ---------- ------------------- --------------
        10 ANALYST         5000                   1              3
        10 MANAGER         2450                   2              6
        10 CLERK           1300                   3              8
        20 ANALYST         6000                   1              1
        20 MANAGER         2975                   2              4
        20 CLERK           1900                   3              7
        30 SALESMAN        5600                   1              2
        30 MANAGER         2850                   2              5
        30 CLERK            950                   3              9

9 row(s) selected.
```

-> This statement displays the sum of salaries classification by department and for each one of these.

The classification is applied on all the columns specified in the **ORDER BY**, if none is specified the classification will be on all columns. **RANK** gives a rank from 1 to the smallest value, except when **DESC** is used.

3.4.3.CUME_DIST function

The **CUME_DIST** function calculates the relative position of a value compared to the other values of the partition.

```
SELECT            column, group_function(argument),
           CUME_DIST() OVER([PARTITION BY column]
           ORDER BY group_function(argument [DESC])
FROM       table
GROUP BY          column
```

The **CUME_DIST** function determines the part of the lines of the partition which are lower or equal to the current value. The result is a decimal value between zero and one. By default, the order is ascending, i.e. the smallest value of the partition corresponds to the smallest **CUME_DIST**.

Example:

```
SQL> SELECT              deptno, job, SUM(sal),
  2               CUME_DIST() OVER(PARTITION BY deptno
  3         ORDER BY SUM(sal) DESC)
  4               AS cume_dist_per_dep
  5  FROM         emp
  6  GROUP BY            deptno, job
  7  ORDER BY            deptno, SUM(sal);

    DEPTNO JOB         SUM(SAL) CUME_DIST_PER_DEP
---------- --------- ---------- -----------------
        10 CLERK           1300                 1
        10 MANAGER         2450        ,666666667
        10 PRESIDENT       5000        ,333333333
        20 CLERK           1900                 1
        20 MANAGER         2975        ,666666667
        20 ANALYST         6000        ,333333333
        30 CLERK            950                 1
        30 MANAGER         2850        ,666666667
        30 SALESMAN        5600        ,333333333

9 row(s) selected.
```

-> This statement displays **CUME_DIST** of the sum of the salaries for each job in each department.

3.5.GROUPING

3.5.1.The GROUPING function

When using **ROLLUP** and **CUBE** operator, blanks appear in the result. The **GROUPING** function exists to differentiate these blanks from NULL values. It can determine the sub-total level, i.e. the groups from which sub-totals are calculated.

```
SELECT      column, group_function, GROUPING(expr)
FROM        table
[WHERE      condition]
[GROUP BY   [ROLLUP][CUBE] (group_by_expression)];
```

The **GROUPING** function can take only one argument. This one must be the same as one of the expressions in the **GROUP BY** clause.

3.5.2.Example

The GROUPING function acts as a Boolean function.
A value of 0 is returned if:

- The expression has been used to calculate the aggregate value.
- The NULL value in the expression column is a stored NULL value.

A value of 1 is returned if:

- The expression has not been used to calculate the aggregate value.
- The NULL value in the expression column is created by **ROLLUP/CUBE** as a result of grouping.

Example:

```
SQL> SELECT             deptno, job, SUM(sal), GROUPING(deptno),
  2                     GROUPING(job)
  3  FROM               emp
  4  GROUP BY ROLLUP    (deptno, job);

    DEPTNO JOB         SUM(SAL) GROUPING(DEPTNO) GROUPING(JOB)
---------- --------- ---------- ---------------- -------------
        10 CLERK           1300                0             0
        10 MANAGER         2450                0             0
        10 PRESIDENT       5000                0             0
        10                 8750                0             1
        20 ANALYST         6000                0             0
        20 CLERK           1900                0             0
        20 MANAGER         2975                0             0
        20                10875                0             1
        30 CLERK            950                0             0
        30 MANAGER         2850                0             0
        30 SALESMAN        5600                0             0

    DEPTNO JOB         SUM(SAL) GROUPING(DEPTNO) GROUPING(JOB)
---------- --------- ---------- ---------------- -------------
        30                 9400                0             1
                          29025                1             1

13 row(s) selected.
```

-> This statement displays two GROUPING columns that indicate if deptno and job have been calculated with the **ROLLUP** operator. When the value is 1, it shows that the column has been used and thus it is not a NULL value.

When GROUPING function is used with CUBE operator, the column GROUPING(deptno) will have a “1” and the column GROUPING(job) a “0”, because deptno is not used to calculate the sum of the sub-group.

3.5.3.GROUPING SETS

You can use the **GROUPING SETS** function to define multiple groupings in the same query. **GROUPING SETS** are a further extension of the **GROUP BY** clause that let you specify multiple groupings of data.

A single **SELECT** statement can now be written with the **GROUPING SETS** function to specify various groupings, rather than multiple **SELECT** statements combined by **UNION ALL** operators.

3.5.4.Using GROUPING SETS

```
SQL> SELECT     deptno,job,mgr,AVG(sal)
  2  FROM       emp
  3  GROUP BY   GROUPING SETS
  4      ((deptno,job), (job,mgr));

    DEPTNO JOB              MGR   AVG(SAL)
---------- --------- ---------- ----------
        10 CLERK                      1300
        10 MANAGER                    2450
        10 PRESIDENT                  5000
        20 ANALYST                    3000
        20 CLERK                       950
        20 MANAGER                    2975
        30 CLERK                       950
        30 MANAGER                    2850
        30 SALESMAN                   1400
        60
           ANALYST         7566       3000

    DEPTNO JOB              MGR   AVG(SAL)
---------- --------- ---------- ----------
           CLERK           7698        950
           CLERK           7782       1300
           CLERK           7788       1100
           CLERK           7902        800
           MANAGER         7839 2758,33333
           PRESIDENT                  5000
           SALESMAN        7698       1400


19 row(s) selected.
```

-> This query computes aggregates over two groupings.
The table is divided into two groups: deptno, job and job, mgr.
The average salary of each of these groups is calculated. The result set displays average salary for each of the two groups.

3.6.Composite columns

3.6.1.The composite columns

A composite column is a collection of columns treated as a unit during the computation of groupings. To use composite column, you have to specify the columns in parentheses as in the following example:

ROLLUP (column A, (column B, column C))

Here, (column B, column C) are treated as a unit and **ROLLUP** will not be applied across (column B, column C).

Composite columns are usually used in **ROLLUP, CUBE,** and **GROUPING SETS.**

GROUPIN SETS statements	Equivalent GROUP BY statements
GROUP BY GROUPING SETS (a, b, c)	**GROUP BY** a **UNION ALL** **GROUP BY** b **UNION ALL** **GROUP BY** c
GROUP BY GROUPING SETS (a, b, (b, c)) (The **GROUPIN SET** has a composite column)	**GROUP BY** a **UNION ALL** **GROUP BY** b **UNION ALL** **GROUP BY** b, c
GROUP BY GROUPING SETS ((a, b, c))	**GROUP BY** a, b, c
GROUP BY GROUPING SETS (a, (b), ())	**GROUP BY** a **UNION ALL** **GROUP BY** b **UNION ALL** **GROUP BY** ()

3.6.2. Example

```
SQL> SELECT     deptno,job,mgr,SUM(sal)
  2  FROM       emp
  3  GROUP BY   ROLLUP(deptno, (job,mgr));

    DEPTNO JOB              MGR   SUM(SAL)
---------- --------- ---------- ----------
        10 CLERK           7782       1300
        10 MANAGER         7839       2450
        10 PRESIDENT                  5000
        10                            8750
        20 ANALYST         7566       6000
        20 CLERK           7788       1100
        20 CLERK           7902        800
        20 MANAGER         7839       2975
        20                           10875
        30 CLERK           7698        950
        30 MANAGER         7839       2850

    DEPTNO JOB              MGR   SUM(SAL)
---------- --------- ---------- ----------
        30 SALESMAN        7698       5600
        30                            9400
        60
        60
                                     29025

16 row(s) selected.
```

-> This statement computes the following groupings:

- (deptno, job, mgr)
- (deptno)
- ()

And displays:

- Total salary for every department.
- Total salary for every department , job, and manager
- Grand total.

3.7.Concatenated groupings

3.7.1.The concatenated groupings

Concatenated groupings offer a concise way to generate useful combinations of groupings.
They are specified by listing multiple **GROUPING SETS, CUBE,** and **ROLLUP,** and separating them with commas.

GROUP BY GROUPING SETS (a, b), **GROUPING SETS** (c, d)

This preceding statement defines the following groupings:

(a, c), (a, d), (b, c), (b, d).

3.7.2.Example

```
SQL> SELECT     deptno,job,mgr,AVG(sal)
  2  FROM       emp
  3  GROUP BY   deptno,
  4             ROLLUP(job),
  5             CUBE(mgr);

    DEPTNO JOB              MGR   AVG(SAL)
---------- --------- ---------- ----------
        10 CLERK           7782       1300
        10 MANAGER         7839       2450
        10 PRESIDENT                  5000
        20 ANALYST         7566       3000
        20 CLERK           7788       1100
        20 MANAGER         7839       2975
        20 CLERK           7902        800
        30 CLERK           7698        950
        30 SALESMAN        7698       1400
        30 MANAGER         7839       2850
        60
           ...

35 row(s) selected(s).
```

-> This statement results in the following groupings:

- (deptno, mgr, job)
- (deptno, mgr)
- (deptno, job)
- (deptno).

The total salary for each of these groups is calculated.
This example displays the following:

- Total salary for every department, manager, and job.
- Total salary for every department and manager.
- Total salary for every department and job.
- Total salary for every department.

4.Advanced subqueries

4.1.Subqueries

4.1.1.What is a subquery

See. SQLP course “Module 2: Techniques of data retrieval”

4.1.2.Multiple-column subqueries

See. SQLP course “Module 2: Techniques of data retrieval”

4.1.3.Using subqueries

See. SQLP course “Module 2: Techniques of data retrieval”

4.2.Column comparisons

4.2.1.Pairwise comparisons

This SQL statement contains a multiple-column subquery because the subquery returns more than one column.

It compares that value in the MGR column and the DEPTNO column of each row in the EMP table with the values in the MGR column and the DEPTNO column for the employees with EMPNO 7369 or 7499.

First, the subquery retrieves the MGR and DEPTNO values for the employees with EMPNO 7369 or 7499.
Then the values are compared with the MGR column and the DEPTNO column for each row in the EMP table.
If the values match, the row is displayed; the employee number 7369 and 7499 will not be displayed.

```
SQL> SELECT      empno, mgr, deptno
     FROM        emp
     WHERE       (mgr, deptno) IN
                                (SELECT mgr, deptno
                                 FROM    emp
                                 WHERE empno IN (7369,7499))
     AND         empno NOT IN (7369,7499);


     EMPNO        MGR     DEPTNO
---------- ---------- ----------
      7521       7698         30
      7844       7698         30
      7900       7698         30
      7654       7698         30
```

4.2.2.Nonpairwise comparisons

The following example shows a nonpairwise comparison.
It displays the EMPNO, MGR, and DEPTNO of any employee whose manager ID matches any of the manager IDs of employees whose employee IDs are either 7639 or 7499 and DEPTNO matches any of the department IDs of employees whose employee IDs are either 7369 or 7499.

First, the subquery retrieves the MGR values for the employees with the EMPNO 7360 or 7499.
The second subquery retrieves the DEPTNO values for the same employees.

The retrieved values of the MGR and DEPTNO columns are compared with the MGR and DEPTNO columns for each row in the EMP table.
If the MGR column of the row in the EMP table matches with any of the values of the MGR retrieved by the inner subquery and if the DEPTNO column of the row in the EMP table matches with any of the values of the DEPTNO retrieved by the second subquery, the record is displayed.

```
SQL> SELECT     empno, mgr, deptno
  2  FROM       emp
  3  WHERE      mgr IN
  4                     (SELECT mgr
  5                     FROM    emp
  6                     WHERE empno IN (7369,7499))
  7  AND        deptno IN
  8                     (SELECT deptno
  9                     FROM    emp
 10                     WHERE   empno IN (7369,7499))
 11  AND        empno NOT IN (7369,7499);

     EMPNO        MGR     DEPTNO
---------- ---------- ----------
      7521       7698         30
      7844       7698         30
      7900       7698         30
      7654       7698         30
```

4.3.Scalar subqueries

4.3.1.The scalar subqueries

We qualify as a scalar subquery, all subqueries that return exactly one column value from one row. Multiple-column subqueries written to compare two or more columns, using a compound **WHERE** clause and logical operators do not qualify as scalar subqueries.

The value of the scalar subquery expression is the value of the select list item of the subquery.
If the subquery returns zero rows, the value of the scalar subquery expression is NULL.
If it returns more than one row, the Oracle Server returns an error.

You can use scalar subqueries in:

- Condition and expression part of **DECODE** and **CASE.**
- All clauses of **SELECT** except **GROUP BY.**
- In the left-hand side of the operator in the **SET** clause and **WHERE** clause of **UPDATE.**

Scalar subqueries are not valid in the following places:

- As default values for columns and hash expressions for clusters.
- In the RETURNING clause of DML statements.
- As the basis of a function-based index.

- In **GROUP BY** clauses, **CHECK** constraints, **WHEN** condition.
- **HAVING** clauses.
- In **START WITH** and **CONNECT BY** clauses.
- In statements that are unrelated to queries, such as **CREATE PROFILE.**

4.3.2.Example

In the following statement, the inner query returns the value 20, which is the department ID of the department located in DALLAS.
The **CASE** expression in the outer query uses the result of the inner query to display the employee ID, name, and a value of Canada or USA, depending on whether the department ID of the record retrieved by the outer query is 20 or not.

```
SQL> SELECT     empno,ename,
  2             (CASE
  3             WHEN deptno=
  4                     (SELECT deptno FROM dept
  5                     WHERE LOC='DALLAS')
  6             THEN ' USA' ELSE 'Canada ' END) location
  7  FROM       emp;

     EMPNO ENAME      LOCATI
---------- ---------- ------
      7369 SMITH      Canada
      7499 ALLEN      USA
      7521 WARD       USA
      7566 JONES      Canada
      7654 MARTIN     USA
      7698 BLAKE      USA
      7782 CLARK      USA
      7788 SCOTT      Canada
      7839 KING       USA
      7844 TURNER     USA
      7876 ADAMS      Canada

     EMPNO ENAME      LOCATI
---------- ---------- ------
      7900 JAMES      USA
      7902 FORD       Canada
      7934 MILLER     USA
      7777 GEROGES    USA

15 row(s) selected.
```

4.4.Correlated subqueries

4.4.1.The correlated subqueries

A correlated subquery is a nested subquery that is evaluated once for each row processed by the main query and which on execution uses a value from a column in the outer query. The correlation can be obtained by using an element from the main query in the subquery.
Stages of execution of a correlated query:

- Get a candidate row (fetched by the outer query).
- Execute the inner query using the value of the candidate row.
- Use the values resulting from the inner query to qualify or disqualify the candidate.
- Repeat until no candidate row remains.

Although this discussion focuses on correlated subqueries in SELECT statements, it also applies to correlated UPDATE and DELETE statements.

4.4.2.Example

A correlated subquery is the mean to read every row in a table and compare values in each row with related data.

```
SELECT      outer1, outer2, ...
FROM        table1 alias1
WHERE       outer1 operator
            (SELECT       inner1
             FROM         table2 alias2
             WHERE        alias1.expr1 =
                          alias2.expr2);
```

It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. In other words, you use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.
The Oracle server performs a correlated subquery when the subquery references a column from a table in the parent query.

Example:

```
SQL> SELECT         empno, sal, deptno
  2  FROM           emp outer
  3  WHERE                 sal > (SELECT       AVG(sal)
  4                        FROM          emp inner
  5                        WHERE         outer.deptno = inner.deptno);

     EMPNO        SAL     DEPTNO
---------- ---------- ----------
      7499       1600         30
      7566       2975         20
      7698       2850         30
      7788       3000         20
      7839       5000         10
      7902       3000         20

6 row(s) selected.
```

→ This statement displays the entire list of employees that have a salary superior to the average salary of their department.

When the main and the correlated query refer to the same table, you have to use a table alias to avoid false results.

4.5.EXITS and NOT EXISTS operators

4.5.1.The EXISTS operator

With nesting SELECT statements, all logical operators are valid. In addition, you can use the EXISTS operator. This operator is frequently used with correlated subqueries to test whether a value is there.
If the subquery returns a line:

- The search doesn't continue in the subquery.
- The condition is labelled TRUE.

If the subquery doesn't return a value for a line:

- The condition is labelled FALSE.
- The search continues with the following line in the subquery.

4.5.2.Using the EXISTS operator

Example:

```
SQL> SELECT       empno, ename, job, deptno
  2  FROM         emp outer
  3  WHERE        EXISTS (SELECT      'X'
  4                       FROM   emp inner
  5                       WHERE  inner.mgr = outer.empno);

     EMPNO ENAME      JOB           DEPTNO
---------- ---------- --------- ----------
      7566 JONES      MANAGER           20
      7698 BLAKE      MANAGER           30
      7782 CLARK      MANAGER           10
      7788 SCOTT      ANALYST           20
      7839 KING       PRESIDENT         10
      7902 FORD       ANALYST           20

6 row(s) selected
```

→ This statement displays the entire list of employees that have at least one person under their orders.

Note that the inner SELECT query does not need to return a specific value, so a literal can be selected. From a performance standpoint, it is faster to select a constant than a column.

4.5.3.The NOT EXISTS operator

Example:

```
SQL> SELECT       deptno, dname
  2  FROM         dept d
  3  WHERE        NOT EXISTS (SELECT  'X'
  4                          FROM      emp e
  5                          WHERE     d.deptno = e.deptno);

    DEPTNO DNAME
---------- --------------
        40 OPERATIONS
```

→ This statement displays all the departments in which there is no employee.

A NOT IN construct can be used as an alternative for a NOT EXISTS operator. However, use it with caution. NOT IN evaluates to FALSE if any member of the set is a NULL value. Therefore, your query will not return any rows.

Example:

```
SQL> SELECT       o.ename
  2  FROM   emp o
  3  WHERE        NOT EXISTS (SELECT  'X'
  4                          FROM     emp i
  5                          WHERE    i.mgr = o.empno);

ENAME
----------
SMITH
ALLEN
WARD
MARTIN
TURNER
ADAMS
JAMES
MILLER

8 row(s) selected.
```

→ This statement displays all the employees that have anybody to give orders to.

The statement from the example can also be written with a NOT IN clause but the result will differ because NULL values will be used.

Example:

```
SQL> SELECT       o.ename
  2  FROM   emp o
  3  WHERE        o.empno NOT IN (SELECT    i.mgr
  4                          FROM     emp i);

No lines selected.
```

→ This statement doesn't return anything because the subquery returns a NULL value and all comparison with a NULL value returns a NULL value; the result is totally different.

When the subquery returns NULL values, it is better not to use NOT IN as an alternative to NOT EXISTS.

4.6.Correlated UPDATE and DELETE

4.6.1.The correlated UPDATE

In the case of the UPDATE statement, you can use a correlated subquery to update rows in one table based on rows from another table.

```
UPDATE     table1 alias1
SET  column = (SELECT expression
              FROM table2 alias2
              WHERE alias1.column = alias2.column);
```

Example:

```
SQL> ALTER TABLE emp
  2  ADD    (dname VARCHAR2(14));

SQL> UPDATE       emp e
  2  SET    dname = (SELECT     dname
  3                  FROM       dept d
  4                  WHERE      e.deptno = d.deptno);

14 row(s) updated.
```

4.6.2.The correlated DELETE

In the case of a DELETE statement, you can use a correlated subquery to delete only those rows that also exist in another table.

```
DELETE FROM        table1 alias1
WHERE       column operator
            (SELECT expression
            FROM table2 alias2
            WHERE alias1.column = alias2.column);
```

Example:

```
SQL> DELETE FROM emp E
  2  WHERE        empno =
  3        (SELECT      empno
  4        FROM         emp_history EH
  5        WHERE        E.empno = EH.empno);
```

→ This statement deletes from EMP table the lines that are already on the EMP_HISTORY table.

4.7.WITH clause

4.7.1.The WITH clause

Using the **WITH** clause, you can define a query block before using it in a query.
With this clause, you can reuse the same query in a **SELECT** statement when it is used more than once within a complex query.

The Oracle Server retrieves the results of a query block and stores it in the user's temporary tablespace.
This improves performance.

4.7.2.Example

The following example represents a situation in which it's possible to improve the performance and write SQL more simply by using the **WITH** clause.
The query creates the query names DEPT_COST and AVG_COST and then uses them in the body of the main query.
Internally, the **WITH** clause is resolved either as an inline view or a temporary table.

```
SQL> WITH
     dept_cost  AS(
        SELECT    d.dname, SUM(e.sal) AS dept_total
        FROM      emp e, dept d
        WHERE     e.deptno=d.deptno
        GROUP BY  d.dname),
     avg_cost   AS(
        SELECT    SUM(dept_total)/COUNT(*) AS dept_avg
        FROM      dept_cost)
     SELECT       *
     FROM         dept_cost
     WHERE        dept_total >
                        (SELECT dept_avg
                        FROM    avg_cost)
     ORDER BY   dname;

DNAME          DEPT_TOTAL
-------------- ----------
ACCOUNTING           8750
RESEARCH            10875
SALES                9400
```

5.Hierarchical retrieval

5.1. Hierarchical queries overview

5.1.1.When is a hierarchical query possible?

Hierarchical queries enable you to retrieve data based on a natural hierarchical relationship between rows in a table.
A relational database does not store records in a hierarchical way. However, where a hierarchical relationship exists between the rows of a single table, a process called tree walking enables the hierarchy to be constructed. A hierarchical query is a method of reporting, in order, the branches of a tree.

5.1.2.Natural structure tree

A tree structure is composed by a parent node that is divided into child branch, which can be divided into branches and so on. For example, the EMP table has a tree structure representing the management reporting line. The hierarchy can be created by looking at the relationship between equivalent values in the EMPNO and MGR columns. This relationship has been exploited by joining the table to itself. An employee's MGR number is the employee number of his or her manager.
The parent-child relationship of a tree structure enables you to control:

- The direction in which the hierarchy is walked
- The starting point inside the hierarchy

5.1.3.Hierarchical queries

Hierarchical queries are queries that contain CONNECT BY and START WITH clauses.

```
SELECT             [LEVEL], column, expr...
FROM               table
[WHERE             condition(s)]
[START WITH        condition(s)]
[CONNECT BY PRIOR condition(s)];
```

LEVEL is a pseudo column, which returns 1 for the root node, 2 for a child of the root and so on. **LEVEL** counts how far down a hierarchical tree you have travelled.
START WITH specifies the root rows of the hierarchy. This clause is required for a true hierarchical query.
CONNECT BY PRIOR specifies the columns in which the relationship between parent and child rows exists. This clause is required for a hierarchical query.
The **SELECT** statement cannot contain a join or a query from a view that contains a join.

5.2. Walking the tree

5.2.1.Start point

The row or rows to be used as the root of the tree are determined by the **START WITH** clause. The **START WITH** clause can be used in conjunction with any valid condition and can contain a sub query.
If the **START WITH** clause is omitted, the tree walk is started with all the rows in the table as root rows. If a **WHERE** clause is used, the walk is started with all the rows that satisfy the **WHERE** condition. This no longer reflects a true hierarchy.
The **START WITH** and **CONNECT BY PRIOR** are not a part of the standard ANSI SQL.

5.2.2. Direction

The hierarchical direction of the query, whether it is from parent to child or from child to parent, is determined by the **CONNECT BY PRIOR** column placement. The **PRIOR** operator refers to the parent row. To find the children of a parent row, the Oracle server evaluates the **PRIOR** expression for the parent row and the other expressions for each row in the table. Rows for which the condition is true are the children of the parent. The Oracle server always selects children by evaluating the **CONNECT BY** condition with respect to a current parent row.
For example, to walk from the top down using the EMP table you have to define a hierarchical relationship in which the EMPNO value of the parent row is equal to the MGR value of the child row.

```
… CONNECT BY PRIOR       empno=mgr
```

The **CONNECT BY** clause cannot contain subqueries.

5.2.3. Example

Example:

```
SQL> SELECT       empno, ename, job, mgr
  2  FROM         emp
  3  START WITH   empno = 7698
  4  CONNECT BY PRIOR mgr = empno;

     EMPNO ENAME      JOB              MGR
---------- ---------- --------- ----------
      7698 BLAKE      MANAGER         7839
      7839 KING       PRESIDENT
```

-> This statement displays a tree walk from top to bottom of the EMP table, starting with the employee 7698.

Expressions in the conditions can represent more than a single column. In that case, **PRIOR** operator is only valid for the first one.

Example:

```
SQL> SELECT             empno, ename, job, mgr
  2  FROM         emp
  3  START WITH   empno = 7698
  4  CONNECT BY PRIOR empno = mgr AND sal > NVL(comm,0);

     EMPNO ENAME      JOB              MGR
---------- ---------- --------- ----------
      7698 BLAKE      MANAGER         7839
      7499 ALLEN      SALESMAN        7698
      7521 WARD       SALESMAN        7698
      7844 TURNER     SALESMAN        7698
      7900 JAMES      CLERK           7698
```

-> This statement returns the lines which manager number equal to employee number of the parent line, and which salary is superior from the commission.

5.3. Organizing data

5.3.1. Ranking rows with the LEVEL pseudo column

You can explicitly show the rank or level of a row in the hierarchy by using the **LEVEL** pseudo column. This will make your report more readable.
The point where one or more branches split away from a larger branch are called nodes, and the very end of a branch is called a leaf, or leaf node. The value of the **LEVEL** pseudo column depends on the

node on which the line is. The value of the **LEVEL** pseudo column will be 1 for the parent node, then 2 for a child node and so on.
The first node of a tree is called the root node, whereas others are called child node. A parent node is a node that has at least one child. A leaf node is a node without any child.

5.3.2.Formatting hierarchical report using LEVEL and LPAD

It is possible to graphically reflect the hierarchy: use the LPAD function in conjunction with the pseudo column **LEVEL** to display a hierarchical report as an indented tree (cf. SQLP course "Module 1: Basic **SELECT** statements").

Example:

```
SQL> COLUMN org_chart FORMAT A15
SQL> SET PAGESIZE 20
SQL> SELECT              LPAD(' ', 3 * LEVEL-3)||ename AS org_chart,
  2                      LEVEL, empno, mgr, deptno
  3  FROM          emp
  4  START WITH    mgr IS NULL
  5  CONNECT BY PRIOR empno = mgr;
SQL> CLEAR COLUMN

ORG_CHART            LEVEL      EMPNO        MGR     DEPTNO
--------------- ---------- ---------- ---------- ----------
KING                     1       7839                    10
   JONES                 2       7566       7839         20
      SCOTT              3       7788       7566         20
         ADAMS           4       7876       7788         20
      FORD               3       7902       7566         20
         SMITH           4       7369       7902         20
   BLAKE                 2       7698       7839         30
      ALLEN              3       7499       7698         30
      WARD               3       7521       7698         30
      MARTIN             3       7654       7698         30
      TURNER             3       7844       7698         30
      JAMES              3       7900       7698         30
   CLARK                 2       7782       7839         10
      MILLER             3       7934       7782         10

14 row(s) selected.
```

-> This statement adds spaces in front of the employee names according to their level in the hierarchy.

5.3.3.Pruning branches

In a hierarchical query, it is possible to delete a branch or a node of a tree.
To prune a branch, you must use the **WHERE** clause. Thus the concerned branch is not shown but child branches are.

Example:

```
SQL> SELECT       deptno, empno, ename, job, sal
  2  FROM         emp
  3  WHERE               ename != 'SCOTT'
  4  START WITH   mgr IS NULL
  5  CONNECT BY PRIOR empno = mgr;

    DEPTNO      EMPNO ENAME      JOB              SAL
---------- ---------- ---------- --------- ----------
        10       7839 KING       PRESIDENT       5000
        20       7566 JONES      MANAGER         2975
        20       7876 ADAMS      CLERK           1100
        20       7902 FORD       ANALYST         3000
        20       7369 SMITH      CLERK            800
        30       7698 BLAKE      MANAGER         2850
        30       7499 ALLEN      SALESMAN        1600
        30       7521 WARD       SALESMAN        1250
        30       7654 MARTIN     SALESMAN        1250
        30       7844 TURNER     SALESMAN        1500
        30       7900 JAMES      CLERK            950
        10       7782 CLARK      MANAGER         2450
        10       7934 MILLER     CLERK           1300
```

-> This statement walks the tree, from top to bottom, from the root node without showing SCOTT, but including child branch (ADAMS).

5.3.4.Ordering Data

It is recommended that you do not use the **ORDER BY** clause when creating hierarchical query reports because the implicit natural ordering of the tree may be destroyed.

5.3.5.ROW_NUMBER() function

Default, NULL values are displayed first, when the value are sorted in a decreasing order. Since Oracle 8*i* Release 2, it is possible to force NULL values to be displayed last, by adding **NULLS LAST** in the **ORDER BY** clause.
The **ROW_NUMBER** function gives to each line of the partition, a number based on the sequence defined by the **ORDER BY** clause.

Example:

```
SQL> SELECT       empno, job, comm,
  2             ROW_NUMBER() OVER(ORDER BY comm DESC NULLS LAST)
  3         AS rnum
  4  FROM         emp;

     EMPNO JOB              COMM       RNUM
---------- --------- ---------- ----------
      7654 SALESMAN        1400          1
      7521 SALESMAN         500          2
      7499 SALESMAN         300          3
      7844 SALESMAN           0          4
      7369 CLERK                         5
      7566 MANAGER                       6
      7900 CLERK                         7
      7934 CLERK                         8
      7902 ANALYST                       9
      7876 CLERK                        10
      7698 MANAGER                      11
      7782 MANAGER                      12
      7788 ANALYST                      13
      7839 PRESIDENT                    14

14 row(s) selected.
```

-> This statement numbers the lines according to the value of the commission and displays NULL values last

6.DML and DDL statements

6.1.Multitable INSERT statement

6.1.1.Types of multitable INSERT statement

You can insert a row into multiple tables as part of a single DML statement using the **INSERT… SELECT** statement.

Here are the different types of mutltitable **INSERT** statements:

- Unconditional **INSERT ALL.**
- Conditional **INSERT ALL.**
- Conditional **FIRST INSERT.**
- Pivoting **INSERT.**

Syntax:

INSERT [ALL] [Conditional_insert_clause]
[insert_into_clause Values_clause] (Subquery)

Condtional_insert_clause:

[ALL] [FIRST]
[WHEN condition THEN] [insert_into_clause_values_clause]
[ELSE] [Insert_into_clause Values_clause]

Syntax	Description
Unconditional **INSERT:ALL into_clause**	Specify **ALL** followed by multiple *insert_into_clauses* to perform an unconditional mutltitable insert. The Oracle Server executes each *insert_into_clause* once for each row returned by the subquery.
Conditional **INSERT: conditional_insert_clause**	Specify the *conditional_insert_clause* to perform a conditional mutltitable insert. The Oracle Server filters each *insert_into_clause* through the corresponding **WHEN** condition, which determines that *insert_into_clause* is executed. A single multitable insert statement can contain up to 127 **WHEN** clauses.
Conditional **FIRST: INSERT**	If you specify **FIRST,** the Oracle Server evaluates each **WHEN** clause in the order in which it appears in the statement. If the first **WHEN** clause is evaluated to true, the Oracle Server executes the corresponding **INTO** clause and skips subsequent **WHEN** clauses for the given row.

Conditional **INSERT: ELSE clause**	For a given row, if no **WHEN** clauses is evaluated to true, and if you have specified an **ELSE** clause, the Oracle Server executes the **INTO** clause list associated with the **ELSE** clause, but if you didn't specify an **ELSE** clause, the Oracle Server takes no action for that row.

6.1.2.Unconditional INSERT ALL

The following statement inserts rows into the SAL_HISTORY and the MGR_HISTORY table.
The **SELECT** statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 7400 from the EMP table.

The details of the employee ID, hire date, and salary are inserted into the SAL_HISTORY table.

The details of employee ID, manager ID, and salary are inserted into the MGR_HISTORY table.

The **INSERT** statement is referred to as an unconditional **INSERT,** as no further restriction is applied to the rows that are retrieved by the **SELECT** statement. All the rows retrieved by the **SELECT** statement are inserted into the two tables.
The **VALUES** clause in the **INSERT** statements specifies the columns from the **SELECT** statement that have to be inserted into each tables.

Each row returned by the **SELECT** statement results in two insertions, one for the SAL_HISTORY table and one for the MGR_HISTORY table.

```
SQL> INSERT ALL
       INTO sal_history        VALUES(empno,hiredate,sal)
       INTO mgr_history        VALUES(empno,mgr,sal)
       SELECT
                 empno, hiredate, sal, mgr
       FROM      emp
       WHERE     empno > 7400;

28 row(s) created.
```

6.1.3.Conditional INSERT ALL

This example is similar to the previous one.
The **INSERT** statement is referred to as a conditional **ALL INSERT,** as further restriction is applied to the rows that are retrieved by the **SELECT** statement. From the rows that are retrieved by the **SELECT** statement, only those rows in which the value of SAL column is more than 2000 are inserted into the SAL_HISTORY table, and similarly only those rows where the value of the MGR column is more than 7400 are inserted in the MGR_HISTORY table.

```
INSERT ALL
     WHEN sal > 2000 THEN
          INTO sal_history VALUES(empno,hiredate,sal)
     WHEN mgr > 7400 THEN
          INTO        mgr_history VALUES(empno,mgr,sal)
          SELECT      empno, mgr, hiredate, sal
          FROM        emp
          WHERE       empno > 7400

18 row(s) created.
```

6.1.4.Conditional FIRST INSERT

The following example inserts rows into more than one table, using one single **INSERT** statement.

The **SELECT** statement retrieves the details of department ID, total salary, and maximum hire date for every department in the EMP table.

The **INSERT** statement is referred to as a conditional **FIRST INSERT,** as an exception is made for the departments whose total salary is more than 1000 $. The condition **WHEN SAL > 1000** is evaluated first. If the total salary for a department is more than 1000$, then the record is inserted into the SPECIAL_SAL table irrespective of the hire date.
If this first **WHEN** clause is evaluated to true, the Oracle Server executes the corresponding **INTO** clause and skips subsequent **WHEN** clauses for this row.

For the rows that do not satisfy the first **WHEN** condition, the rest of the conditions are evaluated. If a row does not satisfy any **WHEN** condition the record is inserted into the hiredate_history table.

```
 SQL> INSERT FIRST
   2            WHEN SAL > 1000         THEN
   3         INTO special_sal        VALUES (deptno,sal)
   4            WHEN hiredate LIKE('%00%') THEN
     5                 INTO  hiredate_history_00               VALUES
(deptno,hiredate)
   6            WHEN hiredate LIKE ('%99%') THEN
     7                 INTO  hiredate_history_99               VALUES
(deptno,hiredate)
   8            ELSE
   9         INTO hiredate_history VALUES(deptno,hiredate)
  10            SELECT         deptno,sum(sal)   sal,   MAX(hiredate)
hiredate
  11            FROM           emp
  12            GROUP BY       deptno;

 4 row(s) created.
```

6.1.5.Pivoting INSERT

Pivoting is an operation in which you need to build a transformation so that each record from any input stream, such as, a nonrelational database table, must be converted into multiple records for more relational database table environment.

In the following statement, the sales data is received from the nonrelational database table SALES_SOURCE_DATA, which is the details of the sales performed by a sales representative on each day of a week, for a week with a particular week ID.

```
 SQL> INSERT ALL
   2     INTO sales_info VALUES(employee_id,week_id,sales_MON)

   3     INTO sales_info VALUES(employee_id,week_id,sales_TUE)

   4     INTO sales_info VALUES(employee_id,week_id,sales_WED)

   5     INTO sales_info VALUES(employee_id,week_id,sales_THUR)

   6     INTO sales_info VALUES(employee_id,week_id,sales_FRI)

     7          SELECT  employee_id,  week_id,  sales_MON,  sales_TUE,
sales_WED
   8            ,sales_THUR, sales_FRI
   9     FROM   sales_source_data;
```

```
5 row(s) created.
```

6.2.External tables

6.2.1.Creating an external table

An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database.
You can use external data as a virtual table. This data can be queried and joined directly and in parallel without requiring the external data to be first loaded in the database.
It's possible to use SQL, PL/SQL, and java to query the data in an external table.

The main difference between external and regular tables is that externally organized tables are read-only.
So, no DML operations are possible on it.

The metadata for an external table is created using the **CREATE TABLE** DDL statement.

The Oracle Server provides two major access drivers for external table.
One, the loader access driver, or ORACLE_LOADER, is used to read data from external files using the Oracle loader technology.

The other Oracle provided access driver, the import/export access driver, or ORACLE_INTERNAL, can be used for both the importing and exporting of data using a platform independent format.

You can create an external table using the **ORGANIZATION EXTERNAL** clause with the **CREATE TABLE** statement.
When you use it, you create a metadata in the data dictionary that you can use to access external data.
If you specify **EXTERNAL** in the **ORGANIZATION** clause, you indicate that the table is read-only, and located outside the database.

TYPE access_driver_type indicates the access driver of the external table.

The **REJECT LIMIT** clause allows you to specify how many conversion errors can occur during a query of the external data before an Oracle error is returned and the query is aborted.

DEFAULT DIRECTORY lets you specify one or more default directory objects corresponding to directories on the file system where the external data sources may reside.

The **ACCESS PARAMETERS** clause allows you to assign values to parameters of the specific access drivers for this external table.

LOCATION is a clause that lets you specify one external locator for each external data source.
Usually, the **location_specifier** is a file, but it need not be.

6.2.2.Example of creating an external table

You have to use the **CREATE DIRECTORY** statement to create a directory object, which is an alias for a directory on the server's file system where an external data source resides.

Note that you must have the **CREATE ANY DIRECTORY** system privileges to create directories.
Once you created a directory, you automatically grant the **READ** object privilege and can grant **READ** privileges to other users and roles.

Syntax:

CREATE [OR REPLACE] DIRECTORY *directory* **AS '***path_name';*

OR REPLACE: Re-create the directory database object if it already exists.

Directory: Specify the name of the directory to create.

Path_name: Specify the full pathname of the operating system.

Assume that there is a flat file that has records in the following format:

10, Jones, 11-DEC-1934
20, Smith, 12-JUN-1972

Records are delimited by new lines, and the fields are terminated by a comma. The name of the file is: /flat_files/emp1.txt

To convert this file as the data source for an external table, whose metadata will reside in the database, you must:

1. Create a directory object emp_dir as follows:

```
CREATE DIRECTORY emp_dir AS '/flat_files';
```

2. Run the following **CREATE TABLE** statement.

```
SQL> CREATE TABLE       odlemp (
     empno NUMBER, ename CHAR (20), hiredate DATE)
     ORGANIZATION EXTERNAL
     (TYPE ORACLE_LOADER
     DEFAULT DIRECTORY emp_dir
     ACCESS PARAMETERS
     (RECORDS DELIMITED BY NEWLINE
     BADFILE            'bad_emp'
     LOGFILE            'log_emp'
     FIELDS TERMINATED BY ','
     (empno CHAR,
     ename  CHAR,
     hiredate CHAR date_format date mask "dd-mon-yyyy"))
     LOCATION           ('emp1.txt'))
     PARALLEL           5
     REJECT LIMIT       200;

Table created.
```

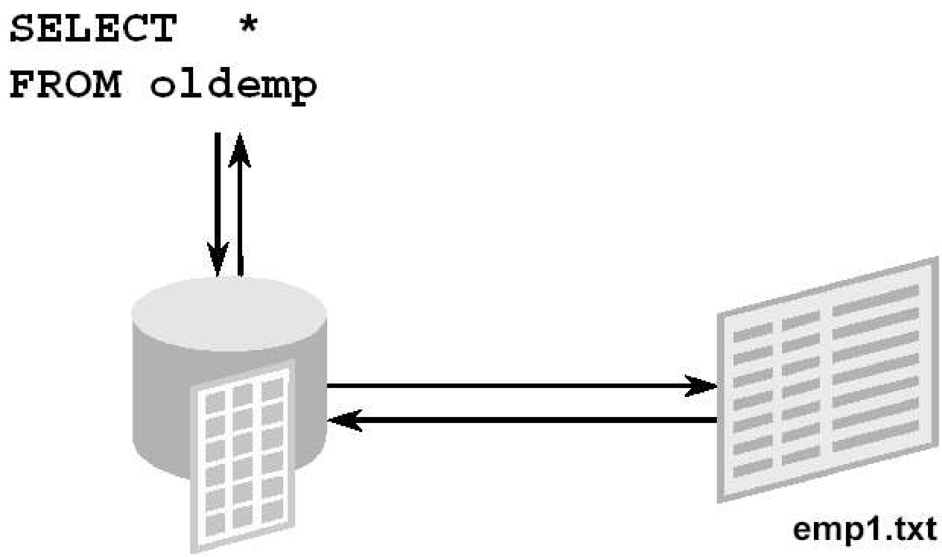
-> The **TYPE** specification is given to illustrate its use.

ORACLE_LOADER is the default access driver if not specified.

ACCESS PARAMETERS provides values to parameters of the specific access driver.

The **PARALLEL** clause enables five parallel execution servers to simultaneously scan the external table when executing **INSERT INTO TABLE** statement.

6.2.3.Querying external tables



An external table does not describe any data that is stored in the database.
It describes how the external table layer needs to present the data to the server.
When the database needs to access data in an external source, it calls the appropriate access driver to get the data from an external source in a form that the database server expects.

Note that the description of the data in the data source is separate from the definition of the external table.
The data types for the fields in the data source can be different from the columns in the table.
The access drivers ensure the data from the data source is processed so that it matches the definition of the external table.

6.3.CREATE INDEX with CREATE TABLE statement

It's possible for a user who has sufficient privileges to create an index while creating a table.

In the following example, the **CREATE INDEX** clause is used with the **CREATE TABLE** statement to create a primary key index explicitly.
You can name the index at the time of **PRIMARY KEY** creation.

```
SQL> CREATE TABLE new_emp
  2     (empno NUMBER(6)
  3                     PRIMARY KEY USING INDEX
  4                     (CREATE INDEX emp_id_idx ON
  5                     new_emp(empno)),
  6     ename VARCHAR2(50));

Table created.
```

You can ensure that the index has been created by querying the USER_INDEX data dictionary table.

```
SQL> SELECT index_name, table_name
  2  FROM user_indexes
  3  WHERE table_name='NEW_EMP';

INDEX_NAME                     TABLE_NAME
------------------------------ ------------------------------
EMP_ID_IDX                     NEW_EMP
```