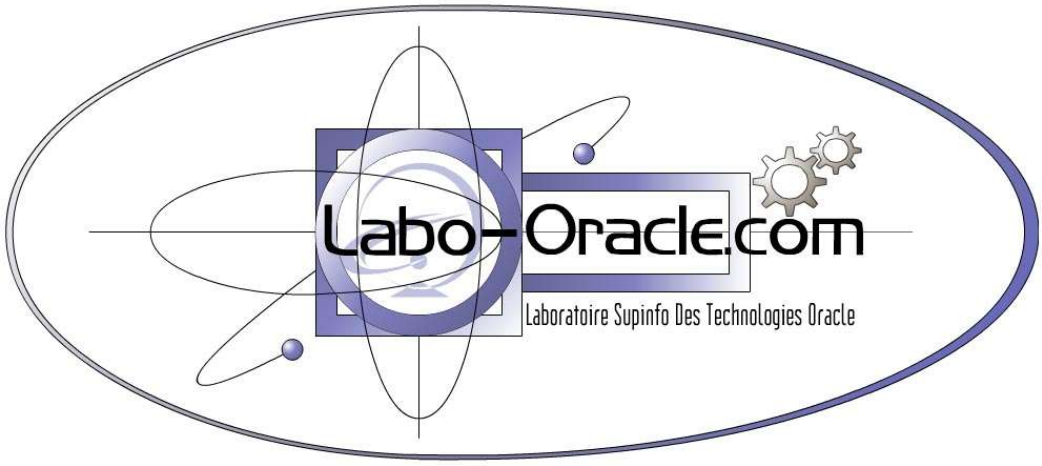


Module n°2

Retrieving date

1Z0-007

Auteur : Reda Benkirane

Retrieving date – d September yyyy

Nombre de pages : 33

Paris Academy of Computer Science
SUPINFO Paris
Ecole Supérieure d'Informatique

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

Summary

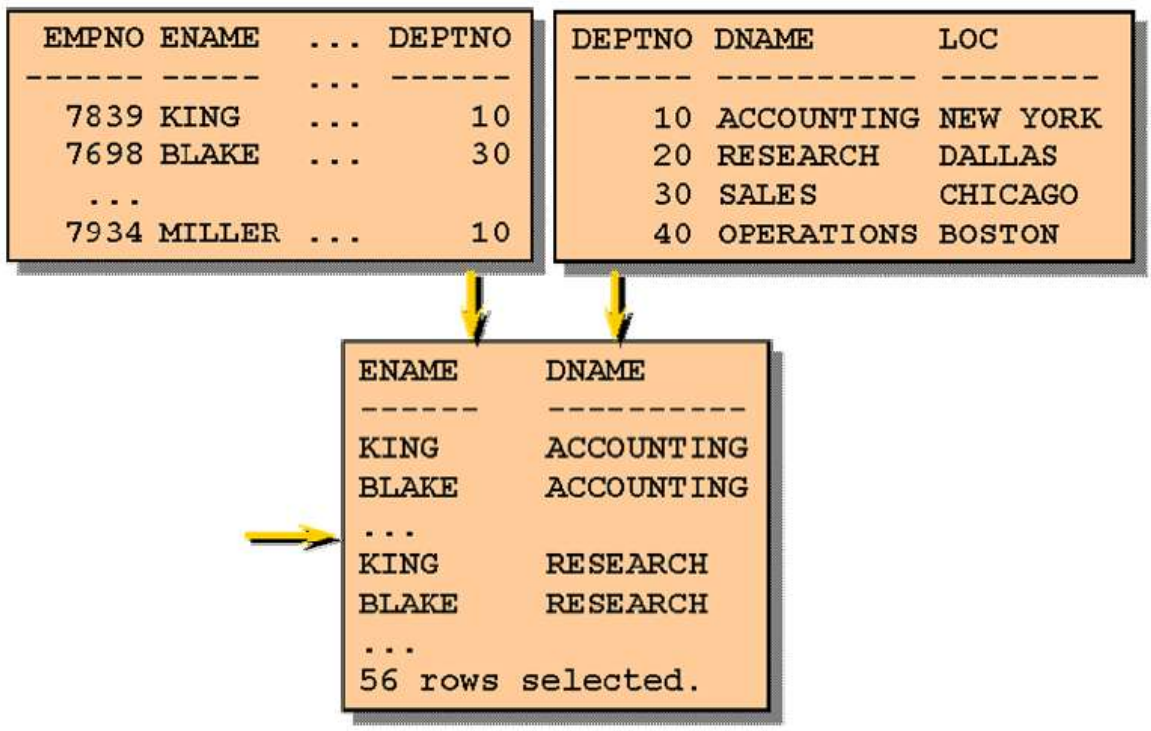
1.Displaying data from multiple tables

1.1.Obtaining data from multiple tables

1.1.1.Generating Cartesian product

A Cartesian product generates a large number of rows, and the result is not useful.
When a join condition is invalid or omitted, the statement generates a Cartesian product; in which all combinations of rows are displayed.

So, if you don't want to have any Cartesian products, you must include valid join condition in the **WHERE** clause.



The Cartesian product, between the EMP and DEPT table generates 56 rows.

1.1.2.Types of joins

To retrieve information from more than one table in a single SQL statement, you MUST join tables using a join condition.
A join condition specifies an existing relation between the columns of these different tables.

Oracle Proprietary Joins (8i and prior)	SQL : 1999 compliant joins
Equijoin	Cross joins
Non-Equijoin	Natural joins
Outer join	Using clause
Self join	Full or two sided outer joins
	Arbitrary join conditions for outer joins

1.1.3.Joining tables using Oracle syntax

To display data from two or more related tables, you just have to write a simple join condition in the **WHERE** clause.

Syntax:

SELECT *table1.column, table2.column*
FROM *table1, table2*
WHERE *table1.column1=table2.column2*

Table1.column Denotes the table and column from which data is retrieved.
Table1.columnl = Is the condition that joins (or relates) the tables together.
Table2.column2

When joining tables in a SQL statement:

- Precede the columns name with the table name for clarity and performance.
- Must prefix the column name with the table name if the same columns name appears in different tables.
- To join *n* tables, you MUST use *n-1* join conditions.

This statements displays employee's department names thanks to a join condition between the EMP and DEPT table.
Values of deptno in the DEPT table are equal to values of deptno on EMP table.

```
SQL> SELECT       emp.ename,dept.dname
  2  FROM         emp,dept
  3  WHERE              emp.deptno=dept.deptno;

     ENAME      DNAME
     ---------- --------------
     SMITH      RESEARCH
     ALLEN      SALES
     WARD       SALES
     JONES      RESEARCH
     MARTIN     SALES
     BLAKE      SALES
     CLARK      ACCOUNTING
     SCOTT      RESEARCH
     KING       ACCOUNTING
     ...
     15 row(s) selected.
```

You should prefix the name of the columns in the **WHERE** clause with the table name to avoid ambiguity.

For example, without the table prefixes, in the previously example, the *deptno* column could be from the EMP table or the DEPT table.

1.2. Equijoins

1.2.1.Retrieving records with equijoins

An EQUIJOIN is used to retrieve information from two or more tables, when there is similarity between values from the different joined tables.

In this example:

- The **SELECT** clause specifies the column names to retrieve:
  - Employee name, employee number and employee department number.
  - Department number and department name.
- The **FROM** clause specifies the two tables that the database must access:
  - EMP table
  - DEPT table
- The **WHERE** clause specifies how the tables are to be joined:
  - EMP.deptno=DEPT.deptno

We use an EQUIJOIN because the deptno column is common to the two tables.

```
SQL> SELECT       emp.ename,dept.dname
  2  FROM         emp,dept
  3  WHERE              emp.deptno=dept.deptno;

     ENAME      DNAME
     ---------- --------------
     SMITH      RESEARCH
     ALLEN      SALES
     WARD       SALES
     JONES      RESEARCH
     MARTIN     SALES
     BLAKE      SALES
     CLARK      ACCOUNTING
     SCOTT      RESEARCH
     KING       ACCOUNTING
     ...
     15 row(s) selected.
```

1.2.2.Additional search conditions using the AND operator

In addition to the join, you can restrict the rows returned using the **AND** operator.
For example, if you want display only the department name of the employee SMITH, you can use the **AND** operator in the **WHERE** clause.

```
SQL> SELECT       emp.ename,dept.dname
  2  FROM              emp,dept
  3  WHERE             emp.deptno=dept.deptno;
  4  AND               emp.ename='SMITH'

     ENAME      DNAME
     ---------- --------------
     SMITH      RESEARCH
```

1.2.3.Using table aliases

You can use table aliases instead of table names, if table names are lengthy.
Aliases help to keep SQL code smaller, therefore using less memory.

Guidelines:

- Table aliases can be up than 30 characters in length.
- If a table alias is used for a table name in the **FROM** clause, that alias MUST be substituted for the table name in the **SELECT** statement.
- Tables' aliases should be meaningful.
- The table alias is valid only for the current **SELECT** statement.
- The aliases are specified in the **FROM** clause after the column name, separated by a space.

Here is an SQL statement joining the DEPT and EMP table and using table's aliases:
We use "e" for the EMP table and "d" for the DEPT table.

```
SQL> SELECT       e.ename,d.dname
  2  FROM         emp e,dept d
  3  WHERE             e.deptno=d.deptno;

     ENAME      DNAME
     ---------- --------------
     SMITH      RESEARCH
     ALLEN      SALES
     WARD       SALES
     JONES      RESEARCH
     MARTIN     SALES
     BLAKE      SALES
     CLARK      ACCOUNTING
     SCOTT      RESEARCH
     KING       ACCOUNTING
     ...
     15 row(s) selected.
```

1.2.4.Joining more than two tables

In some cases, you will have to join more than two tables.
To create a correct join, you MUST create at least *n-1* join conditions for *n* tables.

For example, if you want to display the employee name, the department name and the department location, you have to join the EMP table with the DEPT table, and the DEPT table with the LOCATION table.

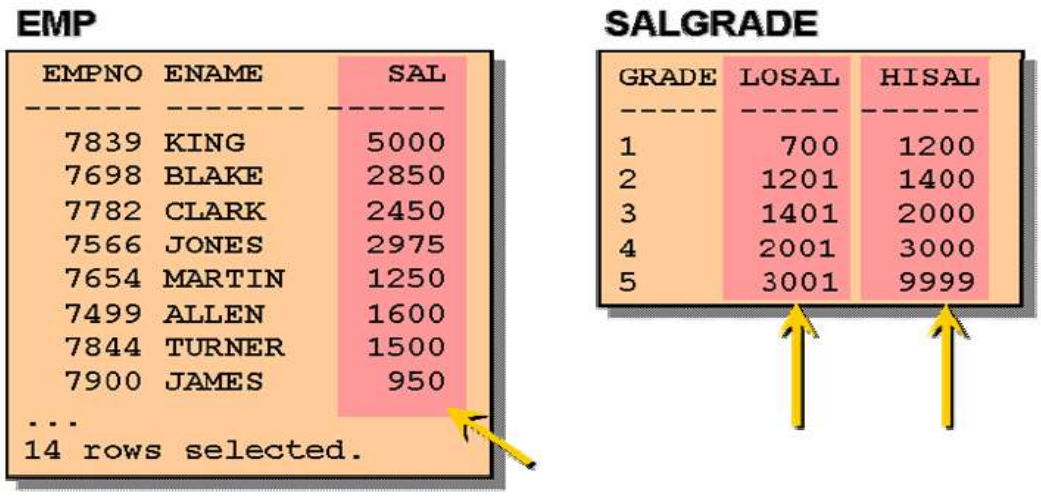
```
SQL> SELECT       c.name, o.ordid, i.itemid, i.itemtot, o.total
  2  FROM         customer c, ord o, item i
  3  WHERE        c.custid = o.custid
  4  AND          o.ordid = i.ordid
  5  AND          c.name = 'TKB SPORT SHOP';

NAME                 ORDID      ITEMID ITEMTOT      TOTAL
---------------- --------- --------- --------- ---------
TKB SPORT SHOP         610         3        58     101.4
TKB SPORT SHOP         610         1        35     101.4
TKB SPORT SHOP         610         2       8.4     101.4
```

This statement displays orders, item ids, items total and the order total for the customer TBK SPORT SHOP.

1.3. Other joins

1.3.1. Retrieving records with Non-Equijoins

A Non-Equijoin is a join condition containing something other than an equality operator.



The salary in the SAL table is enclosed in the LOSAL and HISAL values in the SALGRADE table.

This statement creates a non-equijoin to evaluate an employee's salary grade.
Note that all employees appear once when the statement is executed.

```
SQL> SELECT       e.ename, e.sal, s.grade
  2  FROM         emp e, salgrade s
  3  WHERE              e.sal BETWEEN s.losal AND s.hisal;

     ENAME SAL    GRADE
     ----- ----- --------- ---------
     JAMES 950    1
     SMITH 800    1
     ADAMS 1100   1
     ...
     14 rows selected.
```

You can use other conditions, such as >= and <=; but **BETWEEN** is still the easiest way.

1.3.2. Outer joins

An outer join is used to see the rows that do not meet the join condition in addition to the rows meeting the condition.

The operator is a plus sign enclosed in parentheses (+), and placed on the side of the join that is deficient in information.

Syntax:

SELECT *table1.column, table2.column*
FROM *table1, table2*
WHERE *table1.column (+) =table2.column;*

SELECT *table1.column, table2.column*
FROM *table1, table2*
WHERE *table1.column=table2.column (+);*

Table1.column =	Is the condition that joins the tables together.
Table2.column (+)	Is the outer join symbol, which can be placed on either side of the **WHERE** clause.

This statement displays the employees' name, with their id and department name.
The OPERATIONS department appears; in spite of the lack of employee in this department.

```
SQL> SELECT       e.ename, d.deptno, d.dname
  2  FROM         emp e, dept d
  3  WHERE              e.deptno(+) = d.deptno
  4  ORDER BY     e.deptno;

ENAME          DEPTNO DNAME
---------- ---------- --------------
CLARK              10 ACCOUNTING
KING               10 ACCOUNTING
MILLER             10 ACCOUNTING
...
TURNER             30 SALES
WARD               30 SALES
                   40 OPERATIONS
                   50
...
```

Note that the outer join operator can appear on only ONE side of the expression, and a condition involving an outer join cannot use the **IN** operator or be linked to another condition by the **OR** operator.

1.3.3.Self joins

A self join is used to join a table to itself.
For example, to find the name of each employee's manager, you must join the EMP table to itself, using a self join.

Syntax:

SELECT *alias1.column, alias2.column*
FROM *table1 alias1, table2 alias2*
WHERE *alias1.column=alias2.column*

For example, if you want to display the name of each employee's manager, you can execute the following SQL statement:

```
SQL> SELECT       w.ename,m.ename
  2  FROM         emp w,emp m
  3  WHERE              w.mgr=m.empno;

     ENAME      ENAME
     ---------- ----------
     SCOTT      JONES
     FORD       JONES
     ALLEN      BLAKE
     WARD       BLAKE
     JAMES      BLAKE
     TURNER     BLAKE
     …
     14 row(s) selected.
```

1.3.4.Cross join

The **CROSS JOIN** returns a Cartesian product from the joined tables.
The result is the same as the Cartesian product between two table.
Cartesian product without cross join:

```
SQL> SELECT       ename,dname
  2  FROM         emp,dept;

     ENAME      DNAME
     ---------- --------------
     SMITH      ACCOUNTING
     ALLEN      ACCOUNTING
     WARD       ACCOUNTING
     JONES      ACCOUNTING
     MARTIN     ACCOUNTING
     BLAKE      ACCOUNTING
     CLARK      ACCOUNTING
     SCOTT      ACCOUNTING
     KING       ACCOUNTING
     TURNER     ACCOUNTING
     ADAMS      ACCOUNTING
     ...
     136 row(s) selected.
```

Cartesian product using cross join:

```
SQL> SELECT       ename,dname
  2  FROM         emp
  3  CROSS JOIN   dept;

     ENAME      DNAME
     ---------- --------------
     SMITH      ACCOUNTING
     ALLEN      ACCOUNTING
     WARD       ACCOUNTING
     JONES      ACCOUNTING
     MARTIN     ACCOUNTING
     BLAKE      ACCOUNTING
     CLARK      ACCOUNTING
     SCOTT      ACCOUNTING
     KING       ACCOUNTING
     TURNER     ACCOUNTING
     ADAMS      ACCOUNTING
     ...
     136 row(s) selected.
```

1.3.5.Natural join

The **NATURAL JOIN** is used to let the join be completed automatically by Oracle9i.
The join can happen only on columns having the same names and data types in the two tables.
If the columns have the same name, but different data types, the **NATURAL JOIN** syntax causes an error.

The **NATURAL JOIN** will produce the same result as an equijoin.

Equijoin using the **WHERE** clause:

```
SQL> SELECT       emp.ename,dept.dname
  2  FROM         emp,dept
  3  WHERE              emp.deptno=dept.deptno;

     ENAME      DNAME
     ---------- --------------
     SMITH      RESEARCH
     ALLEN      SALES
     WARD       SALES
     JONES      RESEARCH
     MARTIN     SALES
     BLAKE      SALES
     CLARK      ACCOUNTING
     SCOTT      RESEARCH
     KING       ACCOUNTING
     ...
     15 row(s) selected.
```

Equijoin using the **NATURAL JOIN** clause:

```
SQL> SELECT       ename, dname
  2  FROM         emp
  3  NATURAL JOIN dept;

     ENAME      DNAME
     ---------- --------------
     SMITH      RESEARCH
     ALLEN      SALES
     WARD       SALES
     JONES      RESEARCH
     MARTIN     SALES
     BLAKE      SALES
     CLARK      ACCOUNTING
     SCOTT      RESEARCH
     KING       ACCOUNTING
     ...
     15 row(s) selected.
```

1.3.6. Creating joins with the USING clause

If columns have the same name but the data types do not match, the **NATURAL JOIN** clause fails, but it can be modified with the **USING** clause to specify the columns that should be used for an equijoin.

It's important not to use table name or alias in the referenced columns.

```
SQL> SELECT       e.empno,e.ename,d.loc
  2  FROM         emp e JOIN dept d
  3  USING              (deptno);

           EMPNO ENAME      LOC
     ---------- ---------- -------------
           7369 SMITH      DALLAS
           7499 ALLEN      CHICAGO
           7521 WARD       CHICAGO
           7566 JONES      DALLAS
           7654 MARTIN     CHICAGO
           7698 BLAKE      CHICAGO
           7782 CLARK      NEW YORK
           7788 SCOTT      DALLAS
           ...
           15 row(s) selected.
```

1.3.7. Creating joins with the ON clause

The join condition for the NATURAL join is basically an equijoin of all columns with the same name, but you can specify arbitrary conditions or specify columns to join using the **ON** clause.

```
SQL> SELECT       e.empno,e.ename,d.loc,d.dname
  2  FROM         emp e JOIN dept d
  3  ON           (e.deptno=d.deptno);

            EMPNO ENAME      LOC            DNAME
      ---------- ---------- -------------- --------------
            7369 SMITH      DALLAS         RESEARCH
            7499 ALLEN      CHICAGO        SALES
            7521 WARD       CHICAGO        SALES
            7566 JONES      DALLAS         RESEARCH
            7654 MARTIN     CHICAGO        SALES
            7698 BLAKE      CHICAGO        SALES
            7782 CLARK      NEW YORK       ACCOUNTING
            7788 SCOTT      DALLAS         RESEARCH
            ...
            15 row(s) selected.
```

Note that the **ON** clause can also be used as follows to join columns that have different names.

```
SQL> SELECT       e.ename emp, r.mgr mgr
  2  FROM         emp e JOIN emp r
  3  ON           (e.mgr = r.empno);

     EMP                 MGR
     ---------- ----------
     SCOTT             7839
     FORD              7839
     ALLEN             7839
     WARD              7839
     JAMES             7839
     ...
     14 row(s) selected.
```

1.3.8. Creating three-way joins with the ON clause

A three-way join is a join of three tables. Joins are performed from left to right.
Using the **ON** clause with the **JOIN** clause, you can join three tables

```
SQL> SELECT       c.name,o.ordid,i.itemid,i.itemtot,o.total
  2  FROM         customer c
  3  JOIN         ord o
  4  ON           (c.custid=o.custid)
  5  JOIN         item t
  6  ON           (o.ordid=i.ordid);

       NAME        ORDID     ITEMID   ITEMTOT    TOTAL
---------------- --------- --------- --------- ---------
TKB SPORT SHOP         610         3        58     101.4
TKB SPORT SHOP         610         1        35     101.4
TKB SPORT SHOP         610         2       8.4     101.4
3 rows selected.
```

1.3.9. Left Outer Join and Right Outer Join

This join returns the same results as the return of the classical outer join, the only difference is just that instead of putting the plus sign on the left of the **WHERE** clause, you have to specify the **RIGHT** keyword.

This SQL statement displays all rows in the EMP table, which is the right table even if there is no match in the DEPT table.

```
SQL> SELECT       e.ename, d.deptno, d.dname
  2  FROM         emp e
  3  RIGTH OUTER JOIN dept d
  4  on           e.deptno=d.deptno);

ENAME          DEPTNO DNAME
---------- ---------- --------------
CLARK              10 ACCOUNTING
KING               10 ACCOUNTING
MILLER             10 ACCOUNTING
...
TURNER             30 SALES
WARD               30 SALES
                   40 OPERATIONS
                   50
...
```

1.3.10.Full Outer Join

The **FULL OUTER JOIN** gives all rows from the tables even if there is not match between the joined tables. You can apply additional conditions in the **WHERE** clause using the **AND** operator.

Syntax:

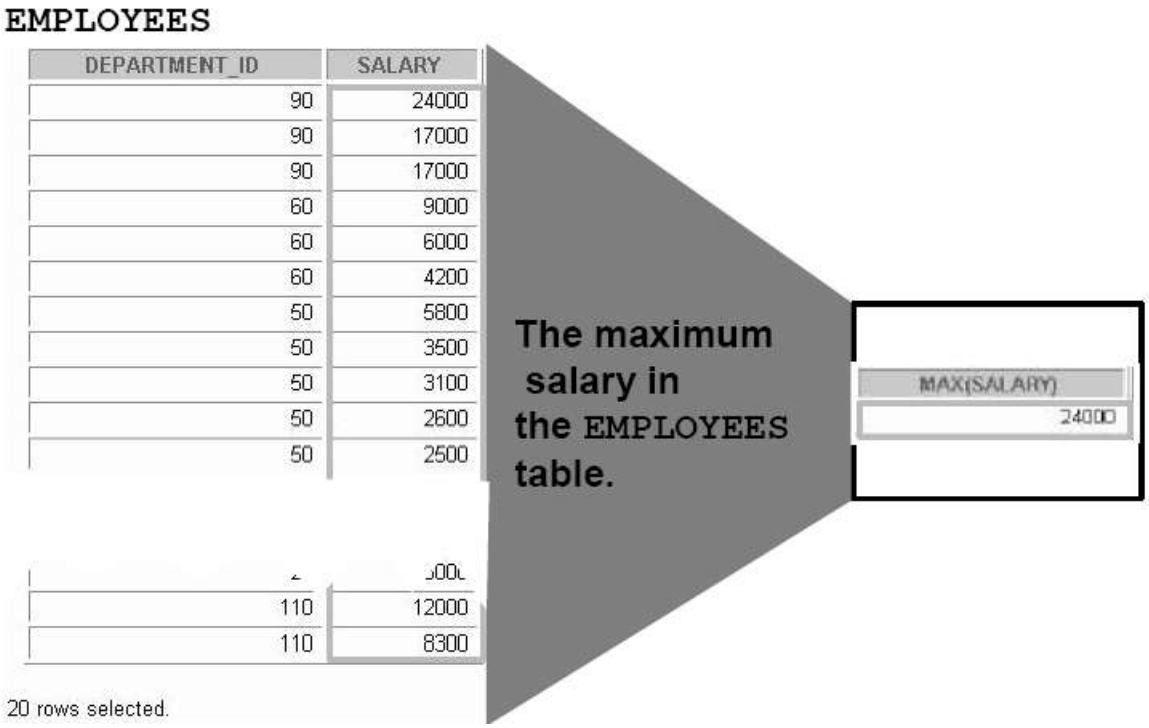
SELECT *table1.column1, table2.column*
FROM *table1*
FULL OUTER JOIN *dept*
ON(*table1.column2=table2.column2).*

```
SQL> SELECT       e.ename,e.deptno,d.dname
  2  FROM         emp e
  3  FULL OUTER JOIN dept d
  4  ON           (e.deptno=d.deptno);


    ENAME          DEPTNO DNAME
    ---------- ---------- --------------
    MILLER             10 ACCOUNTING
    KING               10 ACCOUNTING
    CLARK              10 ACCOUNTING
    ...
    PAPIER             30 SALES
    JAMES              30 SALES
    TURNER             30 SALES
    ...
    WARD               30 SALES
    ALLEN              30 SALES
    THG
    THG


                          store
                          STORE
                          OPERATIONS
                          STORE

22 row(s) selected.
```

2.Aggregating Data using Group Functions

2.1. Group functions

2.1.1. What are Group functions



Group functions operate on sets of rows to give one result per group.
These sets of rows may be the whole table or the table split into smaller groups.

Syntax:

SELECT *[column,] group_function (column) ,* **...**
FROM *table*
[WHERE *condition***]**
[GROUP BY *column***]**
[ORDER BY *column***]**

2.1.2.Types of Group Functions

Here are group functions whose accept arguments:

Function	Description
AVG([DISCTINT | ALL]*n*)	Average value of n, ignoring null values.
COUNT({* | [DISTINCT | ALL] *expr*})	Numbers of rows, where expr evaluates to something other than null.
MAX([DISTINCT | ALL] *expr*})	Maximum value of expr, ignoring null values.
MIN([DISTINCT | ALL] *expr*})	Minimum value of expr, ignoring null values.
STDDEV([DISTINCT | ALL] *x*)	Standard deviation of x, ignoring null values
SUM([DISTINCT | ALL] *x*)	Sum values of x, ignoring null values.
VARIANCE([DISTINCT | ALL] *x*)	Variance of x, ignoring null values.

2.2. Using Group Functions

2.2.1.Using the AVG and SUM functions

You can use **AVG, SUM, MIN** and **MAX** functions with columns storing numeric data.

This SQL statement displays, the average, highest, lowest, and sum of salaries of all the employees.

```
SQL> SELECT       AVG(sal),MAX(sal),MIN(sal),SUM(sal)
  2  FROM         emp;

    AVG(SAL)   MAX(SAL)   MIN(SAL)   SUM(SAL)
    ---------- ---------- ---------- ----------
    2001,66667       5000        800      30025
```

2.2.2.Using the MIN and MAX functions

The **MIN** and **MAX** functions can be used with any datatype.

This statement displays the highest salary and the employee name that is last in the alphabetized list of all employees.

```
SQL> SELECT       MAX(SAL),MAX(ename)
  2  FROM         emp;

    MAX(SAL)   MAX(ENAME)
    ---------- ----------
          5000 WARD
```

Notice that the **AVG, SUM, VARIANCE** and **STDDEV** functions can be used only with numeric datatypes.

2.2.3.Using the COUNT function

The **COUNT (*)** returns the number of rows in a table that satisfy the criteria of the **SELECT** statement, including duplicate rows and null values.
If a **WHERE** clause is included in the **SELECT** statement, the **COUNT (*)** returns number of rows that satisfy the condition.

COUNT(*expr***)** returns the number of non-null values in the *expr* column.
COUNT(DISTINCT *expr***)** returns the number of unique, non-null values in the *expr* column.

This SQL statement displays the number of all rows in the EMP table:

```
SQL> SELECT        COUNT(*)
  2  FROM          emp;

      COUNT(*)
     ----------
             17
```

This statement displays, the number of employees working in the department 30:

```
SQL> SELECT        COUNT(ename)
  2  FROM          emp
  3  WHERE                deptno=30;

     COUNT(ENAME)
     ------------
                       7
```

2.2.4.Using the DISTINCT keyword

You can use the **DISTINCT** keyword to suppress the counting of any duplicate values within a column.

This example displays the number of distinct department values in the EMP table:

```
SQL> SELECT        COUNT(DISTINCT deptno)
  2  FROM          emp;

     COUNT(DISTINCTDEPTNO)
     ---------------------
                         3
```

2.2.5.Using the NVL function with Group Functions

By default, group functions ignore null values, but using the **NVL** function, it's possible to force group functions to include null values.

If you need to calculate the annual salary of all employees, you have to include their commission, but some employees don't have commission.
Using the **NVL** function, the average is calculated even if the commission is null.

```
SQL> SELECT       SAL*12+NVL(comm,0) as "SALARY"
  2  FROM         emp;

    SALARY
    ----------
     12600
     19700
     15500
     35700
     16400
     34200
     29400
```

2.3. Creating groups of data

2.3.1. Using the GROUP BY clause

You can divide the table of information into smaller groups, using the **GROUP BY** clause.

Syntax:

SELECT *column, group_function(column)*
FROM *table*
[WHERE *condition]*
[GROUP BY *expression]*
[ORDER BY *column];*

Guideline:

- When including a group function in a **SELECT** statement, you can't select individual results as well, UNLESS the individual column appears in the **GROUP BY** clause.
- You can exclude rows before dividing them into groups, using a **WHERE** clause.
- You must include the columns that are not used with the group functions in the **SELECT** clause into the **GROUP BY** clause.
- You can't use column alias in the **GROUP BY** clause.
- By default, rows are sorted by ascending order.

When using the **GROUP BY** clause, you have to make sure that all columns in the **SELECT** list, which are not used in the group functions, are included in the GROUP BY clause.

This SQL statement displays the number of employees working in the different departments:

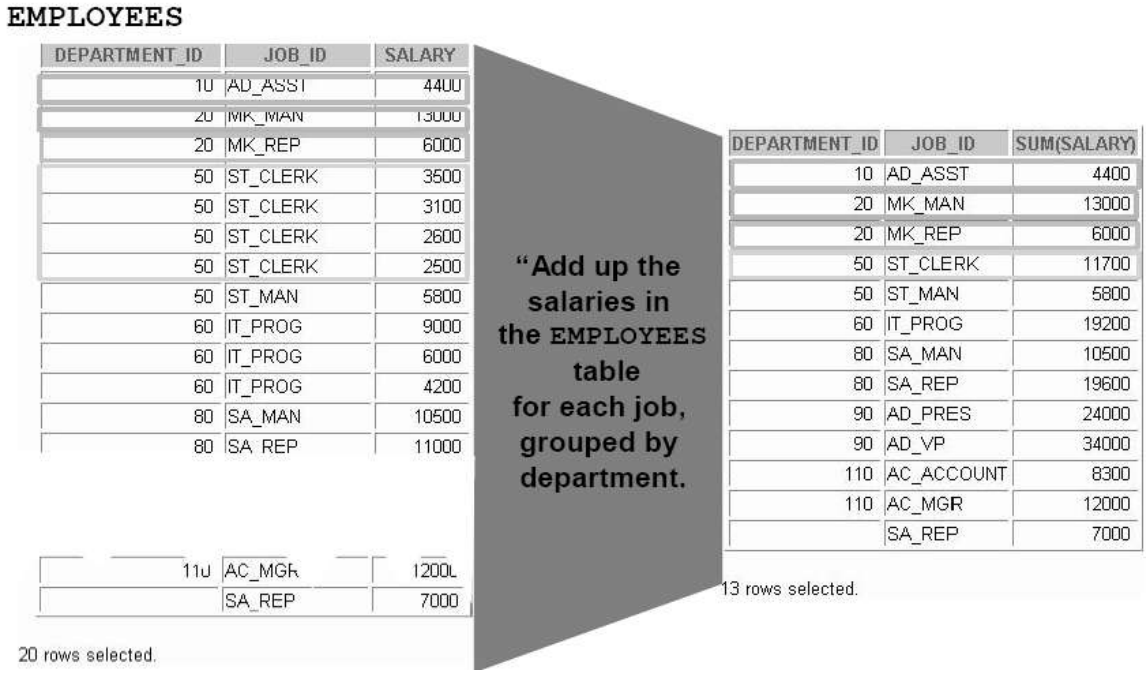
```
SQL> SELECT        COUNT(ename),deptno
  2  FROM          emp
  3  GROUP BY      deptno;

COUNT(ENAME)      DEPTNO
------------ ----------
           3          10
           5          20
           7          30
```

Note that columns do not have to be in the SELECT list to be used is the GROUP BY clause.

```
SQL> SELECT        COUNT(ename)
  2  FROM          emp
  3  GROUP BY      deptno;

COUNT(ENAME)
------------
           3
           5
           7
```

2.3.2.Using the GROUP BY clause on multiple columns

Sometimes, you may need to see results for groups within groups.
You can return summary results for groups and subgroups by listing more than one column in the GROUP BY clause.



In this SQL statement:

- The SELECT clause specifies the columns to retrieve.
- The FROM clause specifies tables that database must access.
- The GROUP BY clause specifies how results must be grouped:
    - First, the rows are grouped by department number.

- Second, within the department number groups, the rows are grouped by job.

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

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        20 ANALYST         6000
        20 CLERK           1900
        20 MANAGER         2975
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        6600

           10 row(s) selected.
```

2.3.3.Excluding group results using the HAVING clause

If you want to restrict groups of rows that you select, you have to use the HAVING clause.

Syntax:

SELECT *column, group_function*
FROM *table*
[WHERE *condition]*
[GROUP BY *group_by_expression]*
[HAVING *group_condition]*
[ORDER BY *column];*

When using the **HAVING** clause, the Oracle Server performs the following steps:

1. Rows are grouped.
2. The group function is applied to the group.
3. The groups that match the criteria in the **HAVING** clause are displayed.

It's recommended to place the **HAVI NG** clause after the **GROUP BY** clause.

This SQL statement displays department id and maximum salaries for those departments whose maximum salary is greater or equal to 3000$.

```
SQL> SELECT       MAX(sal),deptno
  2  FROM         emp
  3  GROUP BY     deptno
  4  HAVING       MAX(sal)>=3000;

        MAX(SAL)     DEPTNO
        ---------- ----------
        5000         10
        3000         20
```

2.3.4.Nesting group functions

If needed, group functions can be nested to a depth of two.
The following statement displays the maximum average salary.

```
SQL> SELECT       MAX(AVG(sal))
  2  FROM         emp
  3  GROUP BY     deptno;

     MAX(AVG(SAL))
     -------------
     2916,66667
```

2.3.5.Illegal queries using group functions

Any column or expression in the **SELECT** list that is not an aggregate function must be in the **GROUP BY** clause.

```
SQL> SELECT       deptno,COUNT(ename)
  2  FROM         emp;
SELECT deptno,COUNT(ename)
       *
ERROR at line 1 :
ORA-00937: Not a single-group group function
```

You cannot use the **WHERE** to restrict groups, and group functions cannot be used in the **WHERE** clause.

```
SQL> SELECT       deptno,AVG(sal)
  2  FROM         emp
  3  WHERE              AVG(sal)=1000;
WHERE AVG(sal)=1000
      *
ERROR at line 1 :
ORA-00934: Group function is not allowed here
```

Column alias cannot be used in the **GROUP BY** clause.

```
SQL> SELECT       deptno department,AVG(SAL)
  2  FROM         emp
  3  GROUP BY     department;
GROUP BY department
         *
ERROR at line 1 :
ORA-00904: Invalid column name
```

3.Subqueries

3.1.Basic Subqueries

3.1.1.Guidelines for using Subqueries

A subquery is a **SELECT** statement that is embedded in a clause of another **SELECT** statement. Using subqueries, you can by build powerful statements out of simple ones.

Subqueries can be placed in the **WHERE** clause, the **HAVING** clause and the **FROM** clause. You can include to your statements operators to add comparison conditions such as '**=**' or **IN**.

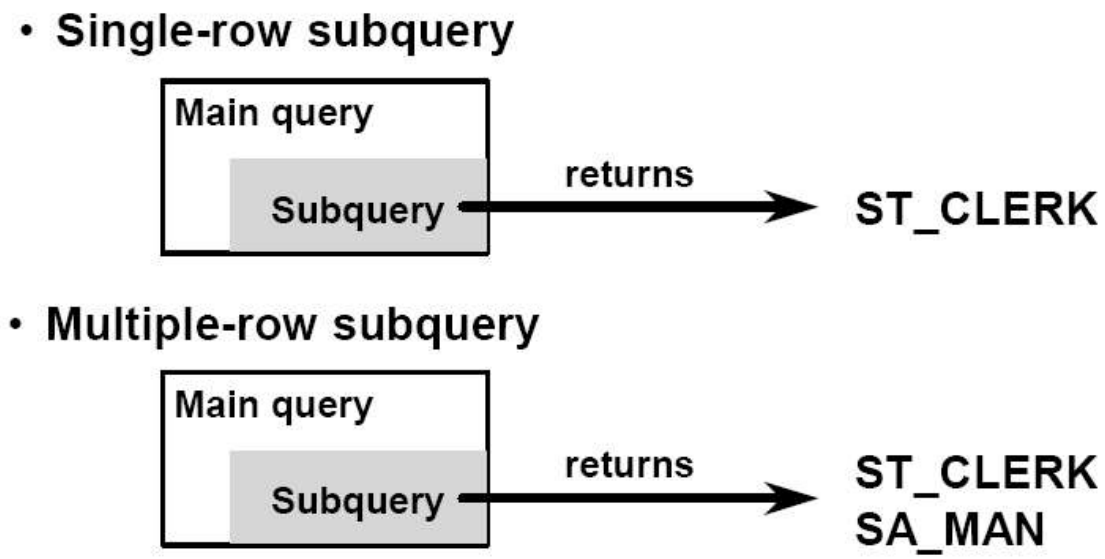
In the following statement, the inner query determines the salary of SMITH, and the outer query takes the result of the inner one and uses this result to display all employee that earn more than this amount.

```
SQL> SELECT       ename
  2  FROM         emp
  3  WHERE              sal>(SELECT sal
  4  FROM         emp
  5  WHERE              ename='SMITH');

     ENAME
     ----------
     ALLEN
     WARD
     JONES
     MARTIN
     ...
     14 row(s) selected.
```

Guidelines:

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison condition.
- The **ORDER BY** clause in the subquery is not needed unless you are performing Top-N analysis.
- Use single-row operator with single-row subqueries and use multiple-row operators with multiple-row subqueries.

3.1.2.Types of subqueries



Type	Description
Single-row	Queries that return only one row from the inner SELECT statement.
Multiple-row	Queries that return more than one row from the inner SELECT statement.

3.2.Single-Row subqueries

3.2.1.Executing a Single-Row subquerie

A single row subquery is one that returns one row from the inner **SELECT** statement.
This type of subquery uses single-row operators, and are placed in the **WHERE** clause.

This statement displays employees that work in the same department as the employee number 7369.

```
SQL> SELECT       ename
  2  FROM         emp
  3  WHERE              deptno=(SELECT deptno
  4                        FROM emp
  5                        WHERE empno=7369);

     ENAME
     ----------
     SMITH
     JONES
     SCOTT
     ADAMS
     FORD
```

Many subqueries can be included into a statement using operators like **AND** and **OR**.

This following statement displays employees working in the same department as the employee SMITH, and has a salary greater than the salary of the employee number 7369.

```
SQL> SELECT       ename
  2  FROM         emp
  3  WHERE              deptno=(SELECT deptno
  4                        FROM emp
  5                        WHERE ename='SMITH')
  6  AND          sal>
  7                      (SELECT sal
  8                        FROM emp
  9                        WHERE empno=7369);

     ENAME
     ----------
     JONES
     SCOTT
     ADAMS
     FORD
```

3.2.2.Using group functions in a subquery

Group functions can be used in single-row subqueries.

The following SQL statement displays the name, the job, and the salary of employees that have a salary equal to the lowest salary.

```
SQL> SELECT       ename,job,sal
  2  FROM         emp
  3  WHERE              sal=(SELECT MIN(sal)
  4                       FROM emp);

     ENAME      JOB              SAL
     ---------- --------- ----------
     SMITH      CLERK            800
```

3.2.3.The HAVING clause with subqueries

As the **WHERE** clause, the **HAVING** clause can contain subqueries.

This statement displays departments that have lowest salary greater that the lowest salary of the department 20.

```
SQL> SELECT       deptno,MIN(sal)
  2  FROM         emp
  3  GROUP BY     deptno
  4  HAVING       MIN(SAL)>
  5                      (SELECT MIN(SAL)
  6                       FROM emp
  7                       WHERE deptno=20);

         DEPTNO   MIN(SAL)
     ---------- ----------
             10       1300
             30        950
```

This SQL statement displays the job which has the lowest average salary.

```
SQL> SELECT       job,AVG(SAL)
  2  FROM         emp
  3  GROUP BY     job
  4  HAVING       AVG(SAL)=    (SELECT MIN(AVG(sal))
  5                             FROM emp
  6                             GROUP BY job);

     JOB         AVG(SAL)
     --------- ----------
     CLERK         1037,5
```

3.3.Multi-Row subqueries

3.3.1.Using the ANY operator in Multi-Row subqueries

The **ANY** operator compares a specified value to each values returned by the Multiple-row subquery.

Comparison operator	Meaning
>ANY	More than the minimum
>ALL	More than the maximum
<ANY	Less than the maximum
<ALL	Less than the minimum

This SQL statement displays employees who are not CLERKS and whose salary is less than the salary of any CLERKS.

```
SQL> SELECT       empno,ename,job,sal
  2  FROM         emp
  3  WHERE              sal< ANY(SELECT sal
  4                          FROM emp
  5                          WHERE job='CLERK')
  6  AND          job <> 'CLERK';

     EMPNO ENAME      JOB              SAL
---------- ---------- --------- ----------
      7521 WARD       SALESMAN        1250
      7654 MARTIN     SALESMAN        1250
      7952 PAPIER     SALESMAN        1000
```

3.3.2.Using the ALL operator in Multi-Row subqueries

The **ALL** operator compares a value to every value returned by a subquery.
Notice that the **NOT** operator can be used with **IN, ANY**, and **ALL** operators.

The following statement displays all employees whose salary is less than the salary of employees in the department 30.

```
SQL> SELECT       ename
  2  FROM         emp
  3  WHERE              sal<   ALL(SELECT   sal   FROM   emp   WHERE
deptno=30);

     ENAME
     ----------
     SMITH
```

3.3.3.Null values in a subquery

If the subquery returns a null value, the outer query will not return any rows.

If you want, the statement to return results, you have to use the **NVL** function.

3.3.4.Using subqueries in the FROM clause

Subqueries can be written in the **FROM** clause.
The subquery's result is a virtual table, which must have an alias.

Example:

```
SQL>        SELECT       e.ename, e.sal, e.deptno, b.salavg
  2         FROM         emp e, (SELECT deptno, AVG(sal) salavg
  3                                FROM emp
  4                                GROUP BY deptno) b
  5         WHERE        e.deptno = b.deptno
  6         AND          e.sal < b.salavg;

ENAME              SAL     DEPTNO     SALAVG
---------- ---------- ---------- ----------
CLARK            2450         10 2916,66667
MILLER           1300         10 2916,66667
SMITH             800         20       2175
ADAMS            1100         20       2175
PAPIER           1000         30 1485,71429
MARTIN           1250         30 1485,71429
JAMES             950         30 1485,71429
WARD             1250         30 1485,71429

8 row(s) selected.
```

This SQL statement displays the name, the salary, the department number and the department salary average for every employee earning a salary less than the average salary of his department.

The subquery returns department numbers and average salary of every department, and these results are stored in the virtual table.
The EMP table and the virtual table *b* are joined.

4.Producing readable output with iSQL*PLUS

4.1.Substitution variables

4.1.1.Using the & substitution variable

In iSQL*PLUS, you can use a single ampersand (&) substitution variable to temporarily store values. You can predefine variables in iSQL*PLUS by using the **DEFINE** command, which creates and assigns a value to a variable.

Use an ampersand to identify variables in your SQL statement.
You don't need to specify a value for each variable.

This statement displays the department name, and the department location of the department number given by the user. (10)

```
SQL> SELECT       dname,loc
  2  FROM         dept
  3  WHERE              deptno= &deptid;
Enter a value for deptid : 10
old   3 : WHERE deptno= &deptid
new   3 : WHERE deptno= 10

DNAME          LOC
-------------- -------------
ACCOUNTING     NEW YORK
```

4.1.2.Character and date values with substitution variables

When using substitution variables with dates or characters, you must enclose values within single quotation marks so that when the user enters the value, he doesn't have to use simple quotation marks.

This SQL statement displays employees whose job is the same as the one specified by the user

```
SQL> SELECT       ename
  2  FROM         emp
  3  WHERE              job='&job';
Enter a value for job : CLERK
old   3 : WHERE job='&job'
new   3 : WHERE job='CLERK'

ENAME
----------
SMITH
ADAMS
...
4 row(s) selected.
```

Note that all values are case sensitive.

4.1.3.Using the && substitution variable

If a variable is preceded by a double ampersand (&&), iSQL*PLUS will prompt the user for its value only one time.

The user has to enter only one time a value for the *column_name* variable, which appears twice in the statement.
The value entered by the user, is used twice.

```
SQL> SELECT       ename,job,&&column_n
  2  FROM         emp
  3  ORDER BY     &&column_name
  4  ;

Enter a column for column_na

old   1 : SELECT ename,job,&&
new   1 : SELECT ename,job,s
old   3 : ORDER BY &&column_n
new   3 : ORDER BY sal

ENAME      JOB              SAL
---------- --------- ----------
SMITH      CLERK            800
JAMES      CLERK            950
PAPIER     SALESMAN        1000
ADAMS      CLERK           1100
...
17 row(s) selected.
```

4.2.Defining substitution variables

4.2.1.Using the DEFINE command

Command	Description
DEFINE variable=variable	Creates a user variable with the CHAR data type and assigns it a value.
DEFINE variable	Displays the variable, its value, and its data type
DEFINE	Displays all user variables with their values and data types.

The first **DEFINE** command defines the *deptname* variable and attributes it a value.
The second **DEFINE** command displays the value of the deptname variable.

The SQL statement displays information about the department which is specified by the deptname variable.

```
SQL> DEFINE deptname=sales
SQL> DEFINE deptname
DEFINE DEPTNAME        = "sales" (CHAR)
SQL> SELECT       *
  2  FROM         dept
  3  WHERE              dname=UPPER('&deptname');

old   3 : WHERE dname=UPPER('&deptname')
new   3 : WHERE dname=UPPER('sales')

    DEPTNO DNAME          LOC
---------- -------------- -------------
        30 SALES          CHICAGO
```

4.2.2.Using then UNDEFINE command

The **UNDIFINE** command is used to delete a user variable.

```
SQL> UNDEFINE deptname
SQL> DEFINE deptname
SP2-0135: the symbol deptname is UNDIFINED
```

4.2.3.Using the VERIFY command

To confirm the changes in the SQL statement, you can use the iSQL*PLUS **VERIFY** command. Setting **VERIFY ON**, forces iSQL*PLUS to display the text of a command before and after substitution variables are replaced with values.

4.3.Customizing the iSQL*PLUS environment

4.3.1.Using the SET command

SET command variables

SET variable and values	Description
ARRAY [SIZE] {20| n}	Sets the database data fetch size.
FEED [BACK] {6| n | OFF | ON}	Displays the number of records returned by a query when the query selects at least *n* records.
HEA[DING] {OFF | ON}	Determines whether column headings are displayed in reports.
LONG {80 | n}	Sets the maximum width for displaying LONG values.

The **SET** commands are used to control current session, so the values are available only for the current session and will not be saved.

Syntax:

SET *system_variable value*

In the syntax:

System_variable Is a variable that controls one aspect of the session environment.

Value Is a value for the system variable.

It is possible to verify what has been set, using the SHOW command.
To see all **SET** variable values, use the **SHOW ALL** command.

4.3.2.iSQL*PLUS format commands

If you want to control the report features, you have to use the following commands:

Command	Description
COL[UMN] [column option]	Controls column format
TTI[TLE] [text | OFF | ON]	Specifies a header to appear at the top of each pages of the report.
BTI[TLE] [text | OFF | ON]	Specifies a footer to appear at the bottom of each pages of the report.
BRE[AK] [ON *report_element*]	Suppresses duplicate values and divides rows of data into sections by using line breaks.

Format commands remain effective until the end of the iSQL*PLUS session or until the format setting is overwritten or cleared.

4.3.3.Using the COLUMN command

This command controls the display of column.

COLUMN command options:

Option	Description
CLE[AR]	Clears any column formats
HEA[DING] *text*	Sets the column heading (a vertical line | forces a line feed in the heading if you do not use justification)
FOR[MAT] *format*	Changes the display of the column data
NOPRI[NT]	Hides the column
NUL[L] *text*	Specifies the *text* to be displayed for null values
PRI[NT]	Shows the column

COLUMN format models:

Element	Description	Example	Result
9	Single Zero-suppression digit	999999	1234
0	Enforces leading zero	099999	001234
$	Floating dollar sign	$9999	$1234
L	Local currency	L9999	L1234
.	Position of decimal point	9999.99	1234.00
,	Thousand separator	9,999	1,234

Example:

- Create column headings:

```
o  SQL> COLUMN ename HEADING 'Employee|Name'
o  SQL> COLUMN sal JUSTIFY LEFT FORMAT $99,990.00
o  SQL> COLUMN mgr FORMAT 999999999 NULL 'No manager'
```

- Displays the current setting for the ENAME column:

```
o  SQL> COLUMN ename
o  COLUMN   ename ON
o  HEADING  'Employee|Name' headsep '|'
```

- Clear settings for the ENAME column:

```
 SQL> COLUMN ename CLEAR
```

4.3.4.Using then BREAK command

You can use the **BREAK** command to divide rows into sections and suppress duplicate values.
Use the **ORDER BY** clause that you are breaking to ensure that the **BREAK** command works.

Syntax:

BREAK *on column[| alias | row]*

column [| alias | row Suppresses the display of duplicate values for a given column

You can clear all **BREAK** settings by using the **CLEAR** command:

CLEAR BREAK

4.3.5.Using the TTITLE and BTITLE commands

TTITLE is used to format page headers and **BTITLE** for footers, which appear at the bottom of the page.
The **BTITLE** and **TTITLE** commands have the same syntax.

Syntax:

TTI[TLE] | BTI[TLE] [*text* **| OFF | ON]**

text: Represents the title text.
OFF | ON: Toggles the title either off or on. It's not visible when turned off.

Example:

* Set the report header:

```
SQL> TTITLE 'Salary|Report'
```

* Set the report footer:

```
SQL> BTITLE 'Confidential'
```

4.3.6.Creating a Script File to run a Report

How to create a script file:

1. Create the SQL **SELECT** statement at the SQL prompt. You must ensure that the data required for the report is accurate before you save the statement to a file and apply formatting commands.

2. Save the **SELECT** statement to a script file.

3. Edit the script file to enter iSQPL*PLUS commands.

4. Add the required formatting commands before the **SELECT** statement. Be sure not to place iSQL*PLUS commands within the **SELECT** statement.

5. Verify that the **SELECT** statement is followed by a run character, either a semicolon (;) or a slash (/).

6. Add the format-creating iSQL*PLUS commands after the run character.

7. Save the script file with your changes.

8. Load the script file into the iSQL*PLUS text window, and click the Execute button.

This script displays the job, name, and salary for every employee whose salary is less than $3000.
The script adds a centred, two-line header that reads “Employee Report” and a centred footer that reads “Confidential”.
It renames the job title column to read “Job name” splits over two lines, renames the employee column name to read “Employee” and finally, renames the salary column to read “Salary” and formats it as $2,500.00.

Note that REM represents a remark or comment in iSQL*PLUS.

```
SET FEEDBACK OFF
TTITLE 'Employee|Report'
BTITLE 'Confidential'
BREAK ON job
COLUMN job HEADING 'Job|Name'
COLUMN ename HEADING 'Employee'
COLUMN sal HEADING 'Salary' FORMAT $99,999.99
REM ** Insert SELECT statement
SELECT job,ename,sal
FROM emp
WHERE sal < 3000
ORDER BY job,ename
/
REM clear all formatting commands ...
SET FEEDBACK ON
COLUMN job CLEAR
COLUMN ename CLEAR
COLUMN sal CLEAR
CLEAR BREAK
```