Module n°3
DML & DDL statements

1Z0-007

Auteur : Reda Benkirane

DML & DDL statements – d September yyyy
Nombre de pages : 55

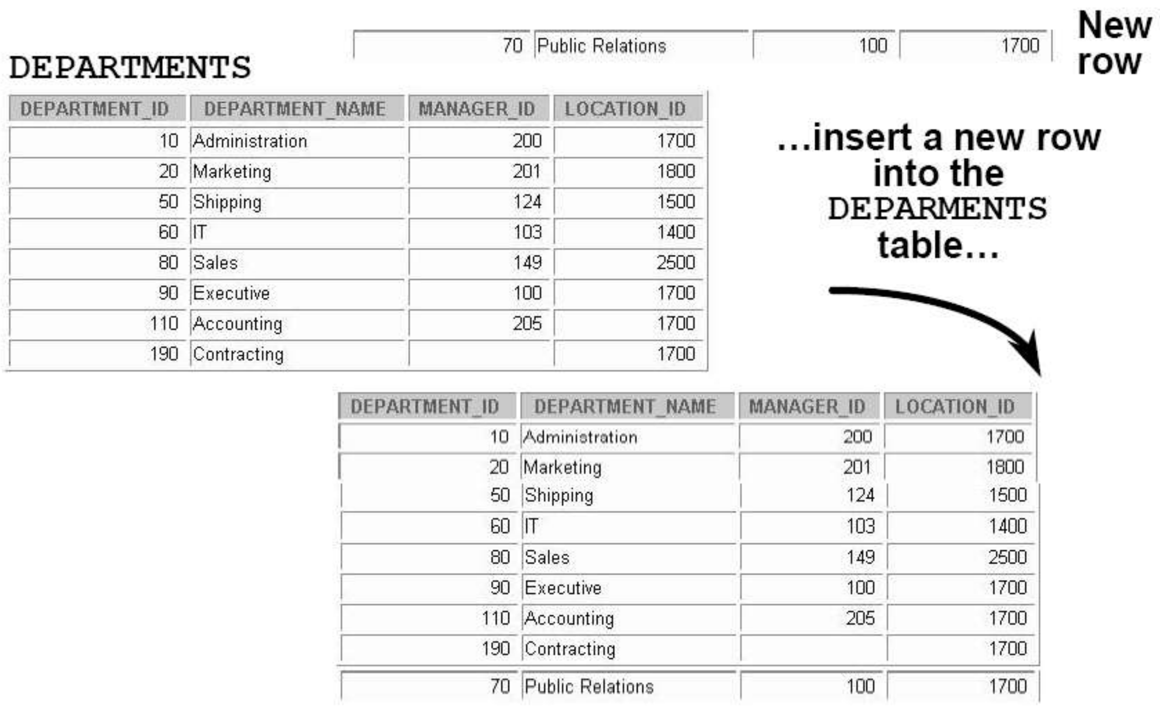
Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com

1.Manipulating data (DML)

1.1. Adding a new row to a table

1.1.1.The INSERT statement

The INSERT statement allows users to add new rows to a table.



70	Public Relations	100	1700

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700
70	Public Relations	100	1700

Syntax:

INSERT INTO *table [(column[, column...])]*
VALUES *(value [, value...]);*

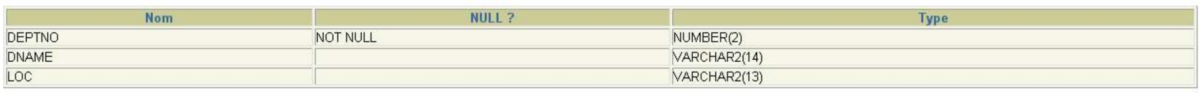
Table	Is the name of the table
Column	Is the name of the column in the table to populate
Value	Is the corresponding value for the column

When used with the **VALUES** clause, the **INSERT** statement adds only one row at a time to a table.

If you insert a new row that contains values for each column, the column list is not required in the **INSERT** clause. However, if you do not use the column list, the values must be listed according to the default order of the columns in the table.

You must enclose character and date values in single quotation marks.

Here is the DEPT table's structure:



Nom	NULL ?	Type
DEPTNO	NOT NULL	NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

This **INSERT** statement inserts a new row into the *DEPT* table:

```
SQL> INSERT INTO  dept(deptno,dname,loc)
  2  VALUES              (90,'Sciences','Paris');

1 row created.
```

1.1.2.Inserting new rows with null values

There are two methods for inserting null values into a table:

Method	Description
Implicit	Omit the column from the column list.
Explicit	Specify the NULL keyword in the values list, Specify the empty string (' ') in the **VALUES** list for character strings and dates.

You must be sure that the column in which you want to insert null values don't have the NOT NULL constraint, by issuing a **DESCRIBE** command and verifying the *NULL?* status of the column.

Implicit way:

```
SQL> INSERT INTO  dept(deptno,dname)
  2  VALUES              (5,'Implicit');

1 row created.
```

Explicit way:

```
SQL> INSERT INTO  dept(deptno,dname,loc)
  2  VALUES              (6,'Explicit',NULL);

1 row created.
```

1.1.3.Inserting special values

You can use functions to insert special values in a table.

The following **INSERT** statement, adds a row in the *EMP* table, supplying the current date and time in the *hiredate* column, using the **SYSDATE** function for current date and time.

```
SQL> INSERT INTO  emp(empno,ename,hiredate)
  2  VALUES              (1,'Robert',SYSDATE);

1 row(s) created.
```

You can issue this SQL statement to ensure that the row has been added:

```
SQL> SELECT       ename,hiredate
  2  FROM         emp
  3  WHERE              empno=1;

     ENAME      HIREDATE
     ---------- --------
     Robert     10/08/04
```

1.1.4.Inserting specific date values

If you want to enter a date in a format other than the default one, for example with another century, you have to use the **TO_DATE** function.

This INSERT statement, adds a new employee into the *emp* table. It sets the *hiredate* column to be “February 3, 1999”.

```
 SQL> INSERT INTO  emp(empno,ename,hiredate)
   2  VALUES             (1,'MIKE',TO_DATE('FEV  3,  1999','MON  DD,
YYYY'));

 1 row(s) created.
```

If you issue the following **INSERT** statement, the year of the *hiredate* is interptreted as 2099:

```
 SQL> INSERT INTO  emp(empno,ename,hiredate)
   2  VALUES             (1,'MIKE',’03-FEB-99’);

 1 row(s) created.
```

When using the RR format, the system automatically provides the correct century.

1.1.5.Using subqueries in an INSERT statement

It’s possible to use a subquery in place of the table name in the **INTO** clause of the **INSERT** statement.

The select list in the subquery must have the same number of columns as the column list on the **VALUES** clause, and rules on the columns of the base table must be respected in order for the **INSERT** statement to work successfully.

Example:

```
 SQL> INSERT INTO
   2            (SELECT  empno,ename,hiredate,job,sal,deptno
   3            FROM     emp
   4            WHERE    deptno=30)
   5  VALUES       (3,'Taylor',
                   TO_DATE('07-JUL-99', 'DD-MON-RR'),
                   'ST_CLERK', 5000,30);

 1 row(s) created.
```

Verification:

```
 SQL> SELECT       empno,ename,hiredate,sal,job,deptno
   2  FROM         emp
   3  WHERE               deptno=30;

      EMPNO ENAME      HIREDATE         SAL JOB           DEPTNO
 ---------- ---------- -------- ---------- --------- ----------
       7499 ALLEN      20/02/81       1600 SALESMAN          30
       7521 WARD       22/02/81       1250 SALESMAN          30
       7654 MARTIN     28/09/81       1250 SALESMAN          30
       7698 BLAKE      01/05/81       2850 MANAGER           30
       7844 TURNER     08/09/81       1500 SALESMAN          30
          3 Taylor     07/06/99       5000 ST_CLERK          30
```

1.1.6.Using the DEFAULT value

You can specify **DEFAULT** in the **INSERT** statement to set the column to the value previously specified as the default value for the column.
If no default value has been specified, Oracle sets the column to NULL.

The following statement uses a default value for the MGR column.
If there is no default value defined, a null value will be inserted.

```
SQL> INSERT INTO         emp(empno, ename, mgr)
  2  VALUES                     (222, 'SCOTT', DEFAULT);

1 row created.
```

1.1.7.Creating a script

You can use script files to issue **INSERT** statements. You just have to save commands with substitutions variables to a file and then execute the commands in the file.
When you run the scripts, you are prompted to input a value for the & substitution variables.
The values are then used into the statement.

This statement adds information into the *DEPT table*, after prompting the user for values for the substitution variables.

```
SQL> INSERT INTO  dept(deptno,dname,loc)
  2  VALUES              (&deptno,'&dname','&loc');

Enter a value for deptno : 3

Enter a value for : Education

Enter a value for : New-York

old   2 : VALUES(&deptno,'&dname','&loc')

new   2 : VALUES(3,'Education','New-York')

1 row created.
```

1.1.8.Copying rows from another table

You can add rows to a table using values derived from existing tables with an **INSERT** statement.
You just have to use a subquery in place of the **VALUES** clause.

Syntax:

INSERT INTO *table [column (, column)] subquery;*

Table:	Is the table name.
Column:	Is the name of the column in the table to populate.
Subquery	Is the subquery that returns rows into the table.

The number of columns and their data types in the column list of the **INSERT** clause must match the number of columns and their data types in the subquery.

```
SQL> INSERT INTO  emp(empno,sal,job)
  2               SELECT       empno,sal,job
  3               FROM         emp
  4               WHERE        job='CLERK';

4 row(s) created.
```

1.2. Changing data in a table

1.2.1.The UPDATE statement

You can modify existing rows by using the **UPDATE** statement.

Syntax:

UPDATE *table*
SET *column = value [, column = value ...]*
[WHERE *condition];*

Table: Is the table name.
Column: Is the name of the column in the table to populate.
Value: Is the corresponding value or subquery for the column.
Condition: Identifies the rows to be updated and is composed of column names expressions, constants, subqueries, and comparison operators.

This statement increases the salary of SMITH by 20 percent.

```
SQL> UPDATE        emp
  2  SET           sal=sal*1.20
  3  WHERE                ename='SMITH';

1 row updated.
```

You can issue a **SELECT** statement to ensure that the change has been done.

```
SQL> SELECT        sal,ename
  2  FROM          emp
  3  WHERE                ename='SMITH';

SAL        ENAME
---------- ----------
       960 SMITH
```

1.2.2.Updating two columns with a subquery

You can update multiple columns in the **SET** clause of an **UPDATE** statement by writing multiple subqueries.

Syntax:

```
UPDATE  table
SET     column1 =
                    (SELECT column
                     FROM   table
                     WHERE condition), column2 = value;
[WHERE  condition];
```

This statement updates the EMP table, increasing the SMITH salary by 20 percent and changing his job to SALESMAN.

```
SQL> UPDATE        emp
  2  SET           sal = sal * 1.20,
  3                job = 'SALESMAN'
  4  WHERE                ename = 'SMITH';

1 row updated.
```

1.2.3.Updating rows based on another table

Using subqueries in an UPDATE statement, it’s possible to update rows in a table based on values from another table.

This SQL statement updates the COPY_EMP table based on the values from the EMP table.

```
SQL> UPDATE        copy_emp
  2  SET           deptno =  ( SELECT deptno
  3                              FROM emp
  4                              WHERE empno=7369);

14 row(s) updated.
```

1.2.4.Using the DEFAULT value

You can specify **DEFAULT** in the **UPDATE** statement to set the column to the value previously specified as the default value for the column.
If no default value has been specified, Oracle sets the column to NULL.

The following statement uses a default value for the MGR column.
If there is no default value defined, a null value will be assigned to MGR.

```
SQL> UPDATE        emp
  2  SET           MGR =DEFAULT
  3  WHERE                ename='SCOTT';

1 row(s) updated.
```

1.2.5.Integrity constraint error

Integrity constraints ensure that the data adheres to a predefined set of rules.
So, if you attempt to update a record with a value that is tied to an integrity constraint, an error occurs.

```
 SQL>        UPDATE       emp
   2         SET          deptno= 55
   3         WHERE        deptno = 10;
 UPDATE emp
 *
 ERROR at line 1 :
 ORA-02291:  integrity  constraint  (SCOTT.EMP_FK)  violated  -  Parent
record not found.
```

1.3.Removing a row from a table

1.3.1.The DELETE statement

The **DELETE** statement is used to remove existing rows from a table.

Syntax:

DELETE [FROM] *table*
[WHERE *condition***];**

If you use the **DELETE** statement without the **WHERE** clause, all rows are deleted from the table.

```
 SQL> DELETE FROM copy_emp;

 14 row(s) deleted.
```

You can delete specific rows by using the **WHERE** clause.

```
 SQL> DELETE FROM  copy_emp
   2  WHERE              empno=7369;

 1 row(s) deleted.
```

This statement deletes the employee identified in the **WHERE** clause.

1.3.2.Deleting a rows based on another table

Using a subquery in the **WHERE** clause, you can delete rows from a table based on values from another table.

This SQL statement deletes all employees who are working in the RESEARCH department.
The subquery searches into the DEPT table to find the department number of the department named ‘RESEARCH’.

```
 SQL> DELETE FROM  emp
   2  WHERE              deptno=(SELECT     deptno
   3                     FROM         dept
   4                     WHERE        dname='RESEARCH');

 5 row(s) deleted.
```

1.3.3.Integrity constraint error

You cannot delete record with a value that is a primary key referenced by an existing foreign key. This statement returns an error, because it attempts to delete a parent record that has child records.

```
SQL> DELETE FROM dept;
DELETE from dept
*
ERROR on line 1 :
ORA-02292: integrity constraint (BENKIR_R.EMP_DEPTNO_FK) violated -
Child record found
```

1.4.Using the WITH CHECK OPTION keyword on DML statements

The **WITH CHECK OPTION** keyword prohibits user from changing rows that are not in the subquery used to identify the table and columns of the DML statement.

In this SQL statement, the subquery identifies rows that are in the department 30, but the department ID is not in the **SELECT** list and a value is not provided for it in the **VALUES** list.
So, inserting this row would result in a department ID of null, which is not in the subquery.

Example 1:

```
SQL> INSERT INTO  (SELECT empno,ename,hiredate,sal,deptno
  2                      FROM emp
  3                      WHERE deptno=30 WITH CHECK OPTION)
  4  VALUES       (9999, 'Smith', SYSDATE, 5000, 20);

     FROM emp
                        *
ERROR at line 2 :
ORA-01402: view WITH CHECK OPTION WHERE-clause violation.
```

Example 2:

```
SQL> INSERT INTO  (SELECT empno,ename,hiredate,sal,deptno
  2                      FROM emp
  3                      WHERE deptno=30 WITH CHECK OPTION)
  4  VALUES       (9999, 'Smith', SYSDATE, 5000, 30);

1 row(s) created.
```

1.5.The MERGE statement

The **MERGE** statement provides the ability to conditionally update or insert data into a table. You can update existing rows and insert new rows using this command.

Because the **MERGE** command combines the **INSERT** and **UPDATE** commands, you need **INSERT** and **UPDATE** privileges on the target table, and **SELECT** privileges on the source table.

Syntax:

```
MERGE INTO      table_name AS table_alias
      USING                  (table | view | subquery) alias
      ON               (join condition)
      WHEN MATCHED THEN
            UPDATE SET
            Col1 = val1
            Col2 = val2
      WHEN NOT MATCHED THEN
            INSERT     (column_list)
            VALUES     (column_values);
```

In the syntax:

INTO:	Specifies the target table
USING:	Identifies the source of the data
ON:	The condition
WHEN MATCHED:	Instructs the server how to respond to the result of the join condition
WHEN NOT MATCHED	

This statement matches the EMPNO column in the COPY_EMP table to the EMPNO column in the EMP table.
If a match is found, the row in the COPY_EMP table is updated; else a row is inserted in it.

```
SQL> MERGE INTO   copy_emp c
  2  USING               emp e
  3  ON           (c.empno=e.empno)
  4  WHEN MATCHED THEN
  5  UPDATE SET
  6               c.sal=e.sal,
  7               c.ename=e.ename
  8  WHEN NOT MATCHED THEN
  9  INSERT VALUES(e.empno,
 10                e.ename,
 11                e.comm,
 12                NULL,
 13                NULL,
 14                NULL,
 15                NULL,
 16                NULL);

11 row(s) merged.
```

The condition specified is evaluated. If rows existed in the COPY_EMP table and EMP IDs matched in both tables, the existing rows in the COPY_EMP table are update

1.6.Database transaction

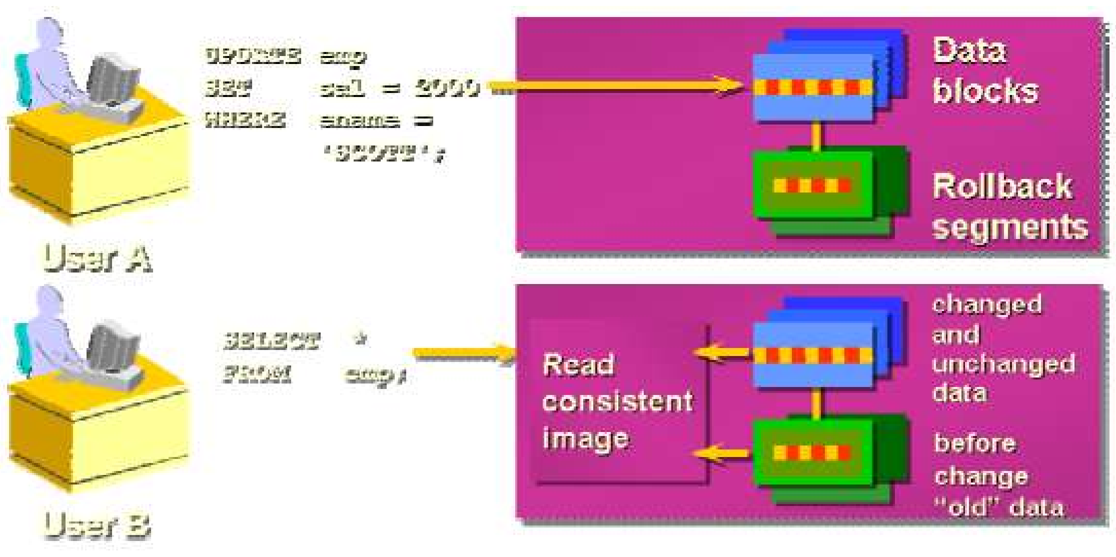
1.6.1.The COMMIT statement

The **COMMIT** statement is used to end the current transaction by making all pending data changes permanent.

It's recommended to issue a **COMMIT** statement after any DML operation of which you're sure. When a commit statement is not issued, Oracle Server keeps a backup of the table being changed to allow users to work on it.

```
SQL> DELETE FROM emp;
```

```
11 row(s) deleted.

SQL> SELECT count(*) FROM emp;

  COUNT(*)
----------
         0

SQL> COMMIT;

Commit completed.
```



```
UPDATE emp
SET    sal = 2000
WHERE  ename =
       'SCOTT';
```

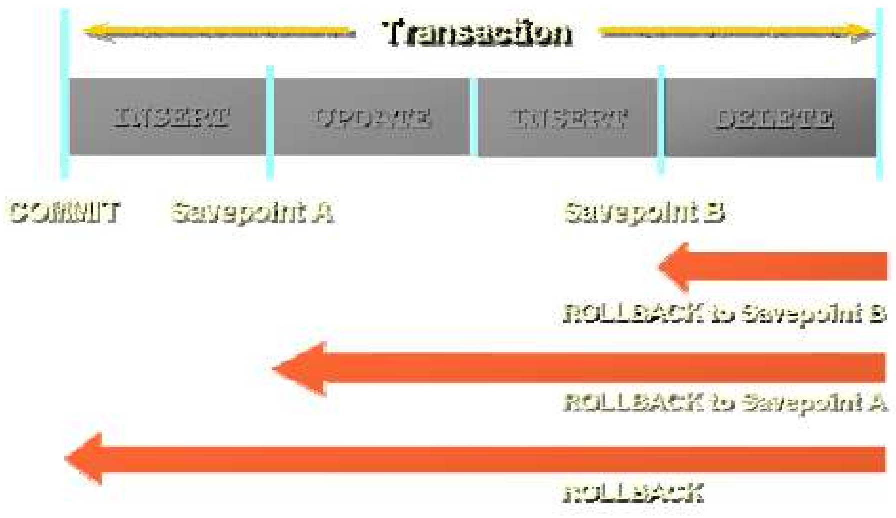
```
SELECT  *
FROM    emp;
```

1.6.2.The ROLLBACK statement

The **ROLLBACK** statement ends the current transaction by discarding all pending data changes.

The **ROLLBACK** statement can be used if you make a mistake in your DML transactions; the command ends the transaction and deletes the locks.

```
SQL> DELETE FROM emp;

11 row(s) deleted.

SQL> SELECT count(*) FROM emp;

  COUNT(*)
----------
         0

SQL> ROLLBACK;

Rollback completed.

SQL> SELECT COUNT(*) FROM emp;

  COUNT(*)
----------
        11
```



It’s possible to roll back the current transaction to a specified save point, by using the **ROLLBACK TO SAVEPOINT** statement.

You have to create a marker in the current transaction by using the **SAVEPOINT** statement, and roll back to that marker by using the **ROLLBACK TO SAVEPOINT** statement.

In this example, we create a save point before the DML transaction. After that, we delete all rows from the EMP table, and then we discard pending changes up to the created marker.

```
SQL> SAVEPOINT delete_rollback;

Savepoint created.

SQL> DELETE FROM emp;

11 row(s) deleted.

SQL> ROLLBACK TO SAVEPOINT delete_rollback;

Rollback completed.
```

1.6.3.Implicit transaction Processing

When DDL, DCL statements or, a normal exit from the (i)SQL*PLUS, without explicitly issuing **COMMIT** or **ROLLBACK** statements, are issued, an automatic commit occurs.

An automatic rollback occurs under an abnormal termination of (i)SQL*PLUS or a system failure.
It's possible to automatically commit DML transactions using the **AUTOCOMMIT** command.
If it's set to ON, the DML statement is committed as soon as it's executed.

1.6.4.Implicit locking

Locks are mechanisms that prevent destructive interaction between transactions accessing the same resource, either a user object or a system object not visible to the user.
The locking is performed automatically by Oracle and requires no user action, and it occurs for every SQL statements except the **SELECT** statement.

When performing DML operations, DML locks occur at two levels:

- A share lock is automatically obtained at the table level during the DML operations.
- An exclusive lock is acquired automatically for each rows modified by the DML operation, which prevents the rows from being changed by other transactions. It ensures that other users cannot modify these rows.
- DDL lock occurs when you modify a database object such as a table.

2.Creating and managing tables

2.1. Creating tables

2.1.1.Naming rules

Here are the most important naming rules:

- Table names and column names must begin with a letter and be 1-30 characters long.
- Names must contain only the character a-z, A-Z, 0-9, _ (Underscore), $ and #.
- Names must not duplicate the name of any other database object.
- Names must not be an Oracle Server reserved word.

Note that names are not case sensitive.

2.1.2.The CREATE TABLE statement

First of all, you must have the **CREATE TABLE** privilege to use this statement, and a storage area to create objects.
The **CREATE TABLE** statement creates tables to store data. It's a DDL statement.
This statement, such as all the DDL statements, has an immediate effect on the database.

Syntax:

CREATE TABLE *[schema].table*
*(Column datatype [DEFAULT expr], […]);**

SCHEMA:	Is the schema on which you want to create the table.
TABLE:	Is the name of the table.
DEFAULT expr:	Specifies a default value, if a value is omitted in the **INSERT** statement.
COLUMN:	Is the name of the column.
DATATYPE:	Is the datatype of the column. (You must specify its length).

This SQL statement creates the EMP2 table, with three columns: EMPNO, ENAME and LOCATION.

```
SQL> CREATE TABLE EMP2
  2                (EMPNO NUMBER(2),
  3                ENAME VARCHAR2(50),
  4                LOCATION VARCHAR2(50) DEFAULT 'Not specified');
Table created.
```

You can issue a **DESCRIBE** command to ensure that the table has been created.

```
SQL> DESC emp2;
 Nom                                       NULL ?   Type
 ----------------------------------------- -------- -------------

 EMPNO                                              NUMBER(2)
 ENAME                                              VARCHAR2(50)
 LOCATION                                           VARCHAR2(50)
```

2.1.3.The DEFAULT option

This option specifies a default value for a column during an **INSERT** statement.
So, if no value is specified for this column, a default value will be assigned to it.
The default values can be a literal, an expression, or a SQL function, such as **SYSDATE** or **USER,** but it cannot be the name of a column or a pseudo column.

2.1.4.Querying the data dictionary

User tables are table created by a user such as EMP2. But there is another collection of tables and views created and maintained by the Oracle Server known as *data dictionary,* and which contains information about the database.
All data dictionary tables are owned by *sys,* and are rarely accessed by users.

The data dictionary can be divided in four categories, differenced by prefixes that reflect their intended use:

Prefix	Description
USER_	Views containing information about objects owned by the user.
ALL_	Views containing information about all objects accessible to the user.
DBA_	These views are restricted views, which can be accessed only by people who have been assigned the DBA role.
V$	Dynamic performance views, database performance memory, and locking.

It's possible to query the *data dictionary* tables to view the database objects you own.

Use this query to see the names of table owned by the user:

```
SQL> SELECT       table_name
  2  FROM         user_tables;

TABLE_NAME
------------------------------
EMP2
EMP
DEPT
COPY_EMP
...
9 row(s) selected.
```

Use this query to view distinct object types owned by the user:

```
SQL> SELECT       DISTINCT object_type
  2  FROM         user_objects;

OBJECT_TYPE
------------------
FUNCTION
INDEX
JAVA CLASS
JAVA SOURCE
PACKAGE
PACKAGE BODY
PROCEDURE
SEQUENCE
TABLE

9 row(s) selected.
```

Use this query to view tables, views, synonyms, and sequences owned by the user:

```
SQL> SELECT       *
  2  FROM         user_catalog;

TABLE_NAME                     TABLE_TYPE
------------------------------ -----------
EMP2                           TABLE
ID_ARTICLE_SEQ                 SEQUENCE
ID_CARAC_SEQ                   SEQUENCE
ID_CAT_SEQ                     SEQUENCE
ID_CLIENT_SEQ                  SEQUENCE
ID_COMMANDE_SEQ                SEQUENCE
ID_E_COMMANDE_SEQ              SEQUENCE
T_ARTICLE                      TABLE
T_CATEGORIE                    TABLE
T_CLIENT                       TABLE
T_ENTETE_CARAC                 TABLE

TABLE_NAME                     TABLE_TYPE
------------------------------ -----------
T_ENTETE_COMMANDE              TABLE
T_INTER_COMMANDE               TABLE
T_LANGAGE                      TABLE
T_VALEUR_CARAC                 TABLE

15 row(s) selected.
```

Use this query to see all tables that the user has access to:

```
SQL> SELECT       table_name
  2  FROM         all_tables;

TABLE_NAME
------------------------------
DUAL
SYSTEM_PRIVILEGE_MAP
TABLE_PRIVILEGE_MAP
STMT_AUDIT_OPTION_MAP
AUDIT_ACTIONS
PSTUBTBL
ODCI_SECOBJ$
ODCI_WARNINGS$
DEF$_TEMP$LOB
WM$WORKSPACES_TABLE
...
27 row(s) selected.
```

2.1.5.Data types

Data type	Description
VARCHAR2(*size)*	Variable-length character data(a maximum size must be specified: Min= 1, Max= 4000)
CHAR[(*size*)]	Fixed-length character data of length *size* bytes. (a maximum size can be specified: Min= 1, Max= 2000)
NUMBER [(*p,s*)]	Number having precision *p* and scale *s*.(The precision is the total number of decimal digits, and the scale is the number of digits to the right of the decimal point; the precision can range from 1 to 38 and the scale can range from -84 to 127).
DATE	Date and Time values to the nearest second between January 1,4712 B.C., and December 31,9999 A.D.
LONG	Variable-length character up to 2 gigabytes.
CLOB	Character data up to 4 gigabytes.
RAW(*size*)	Raw binary data of length *size*. (a max size must be specified . max=2000).
LONG RAW	Raw binary data of variable length up to 2 gigabytes.
BLOB	Binary data up to 4 gigabytes.
BFILE	Binary data, stored in an external file; up to 4 gigabytes.
ROWID	A 64 base number system representing the unique address of a row in its table.

2.1.6.DateTime data types

Type	Description	Syntax
TIMESTAMP	Allows the time to be stored as a date with fractional seconds. There are several variations of the data type.	**TIMESTAMP**[(fractional_seconds_precision)]
INTERVAL YEAR TO MON TH	Allows time to be stored as an interval of years and months.	**INTERVAL YEAR** [(year_precision)] **TO MONTH**
INTERVAL DAY TO SECON D	Allows time to be stored as an interval of days to hours, minutes, and seconds.	**INTERVAL DAY** [(day_precision)] **TO SECOND**

2.1.7.Creating a table by using a subquery syntax

There is a second method to create tables which consists on applying the **AS *subquery*** clause.
This method creates a table and inserts rows returned by the subquery.

Syntax:

CREATE TABLE *tablename*
[(column1, column2 ...)]
AS *subquery;*

Table: is the name of the table
Column: is the name of the column, default value, and integrity
constraint.
Subquery: is the SELECT statement that defines

Note that if column specifications are given, the column names of the table are the same as the column names in the subquery.

The following statement creates the DEPT2 table, which contains detail of all employees working in department 30.

```
SQL>  CREATE TABLE       dept2
  2   AS
  3        SELECT        hiredate,
  4                      sal,
  5                      ename
  6        FROM          emp
  7       WHERE          deptno=30;

Table created.
```

2.2. Altering a table

2.2.1.The ALTER TABLE statement

The **ALTER TABLE** statement allows user to change the structure of existing tables.
This statement can be used to add, modify, or drop columns.

Syntax:

ALTER TABLE *tablename*
ADD|MODIFY|DROP *(column datatype [DEFAULT expr]);*

Table: is the name of the table.
ADD|MODIFY|DROP: is the type of modification.
Column: is the name of the column to add, to modify, or to
remove.
Datatype: is the data type and length of the new column
DEFAULT expr: specifies the default value for a the column.

2.2.2.Adding a column

To add columns, you must use the **ALTER TABLE** statement with the **ADD** clause.
Note that it is not possible to specify where the column will appear.
The new column becomes the last one.

Here is the structure of the dept2 table:

```
SQL> DESCRIBE     dept2

 Nom                                         NULL ?   Type
 ------------------------------------------ -------- --------------
 HIREDATE                                             DATE
 SAL                                                  NUMBER(7,2)
 ENAME                                                VARCHAR2(10)
```

The following statement adds a column named job to the dept2 table.

```
SQL> ALTER TABLE  dept2
  2  ADD          (job VARCHAR2(10));

Table altered.
```

Here is now, the new structure of the dept2 table.

```
SQL> DESCRIBE     dept2

 Nom                                         NULL ?   Type
 ------------------------------------------ -------- -------------
 HIREDATE                                             DATE
 SAL                                                  NUMBER(7,2)
 ENAME                                                VARCHAR2(10)
 JOB                                                  VARCHAR2(10)
```

2.2.3.Modifying a column

You can change a column’s data type, size, and default value using the **MODIFY** clause with the **ALTER TABLE** statement.
Note that you cannot change the data type of a column if it contains rows, and you can convert a CHAR column to the VARCHAR2 data type or convert a VARCHAR2 column the to CHAR data type only if the column contains null values or if you do not change the size.

The following statement modifies the length of the job column, and sets it to 20.

```
SQL> ALTER TABLE  dept2
  2  MODIFY       (job VARCHAR(20));

Table altered.
```

2.2.4.Dropping a column

You can use the **DROP** clause with the **ALTER TABLE** statement to drop columns from a specified table.
The columns can be empty or contain data, and the table must have at least one column remaining after it’s altered.
Note that the drop action cannot be roll backed.

The following statement drops the job column from the dept2 table.

```
SQL> ALTER TABLE  dept2
```

```
  2  DROP COLUMN  job;

Table altered.
```

2.2.5.The SET UNUSED option

The **SET UNUSED** option can be used to set one or more columns as unused.
The **DROP UNUSED COLUMNS** option is used to remove the columns that are marked as unused.

Syntax:

ALTER TABLE *tablename*
SET UNUSED *(column);*

OR
ALTER TABLE *tablename*
SET UNUSED COLUMN *column;*

ALTER TABLE *table*
DROP UNUSED COLUMNS;

- A **SELECT** statement will not retrieve data from unused columns.
- The names of columns marked unused will not be displayed during a **DESCRIBE** statement.
- You can add to the table a new column with the same name as an unused column.

The following statement sets to unused the empno, and sal columns in the EMP table.

```
SQL> ALTER TABLE          emp
  2  SET UNUSED           (empno,sal);

Table altered.

SQL> DESCRIBE             emp

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

     ENAME                                                    VARCHAR2
(10)
     JOB                                                  VARCHAR2(9)
     MGR                                                  NUMBER(4)
     HIREDATE                                             DATE
     COMM                                                 NUMBER(7,2)
     DEPTNO                                               NUMBER(2)
```

Now, with this statement, we drop all unused columns from the EMP table.

```
SQL> ALTER TABLE         emp
  2  DROP UNUSED         COLUMNS;

Table altered.

SQL> DESCRIBE            emp

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

     ENAME                                                    VARCHAR2
(10)
     JOB                                                  VARCHAR2(9)
     MGR                                                  NUMBER(4)
     HIREDATE                                             DATE
     COMM                                                 NUMBER(7,2)
     DEPTNO                                               NUMBER(2)
```

2.3.Dropping a table

The **DROP TABLE** statement removes the definition of an Oracle table.
When you drop a table:

- All data and structure in the table are deleted.
- Any pending transactions are committed.
- All indexes are dropped.
- You cannot roll back the action.

Syntax:

DROP TABLE *tablename;*

Tablename: Is the name of the table that you want to drop.

The following statement drops the EMP table from the Oracle data base.

```
SQL> DROP TABLE   emp;

Table dropped.
```

2.4.Managing tables

2.4.1.Changing the name of a table

The **RENAME** statement is used to change the name of a table, view, sequence, or synonym. To use this statement, you must be the owner of the object that you want to rename.

Syntax:

RENAME *old_name* **TO** *new_name;*

Old_name:	Is the old name of the object.
New_name:	Is the new name of the object.

The following statement changes the name of the dept table to department.

```
SQL> RENAME dept TO department;

Table renamed.
```

2.4.2.Truncating a table

The **TRUNCATE TABLE is** a DDL statement used to remove all rows from a table and to release the storage space used by that table.
Note that you cannot roll back row removal when using **TRUNCATE.**

Syntax:

TRUNCATE TABLE *tablename;*

Tablename: is the name of the table.

You can also remove all rows from a table using the **DELETE** statement, but it does not release storage space, and it's slower than the **TRUNCATE** statement.
If the table is the parent of a referential integrity constraint, you cannot truncate the table, if you don't disable constraint before using the statement.

This example removes all rows from the DEPT table, and releases the storage space used by it.

```
SQL> TRUNCATE TABLE      dept;

Table truncated.
```

2.4.3.Adding comments to a table

It's possible to add comments on a table or a column using the **COMMENT** statement.
These comments can be viewed trough the data dictionary views:

- **ALL_COL_COMMENTS**
- **USER_COL_COMMENTS**
- **ALL_TAB_COMMENTS**
- **USER_TAB_COMMENTS**

Syntax:

COMMENT ON TABLE *table* | **COLUMN** *table. column*
IS *'text';*

Table:	Is the name of the table.
Column:	Is the name of the column.
Text:	Is the text of the comment.

The text must be enclosed within simple quotation marks.
To drop comments from the database, you just have to set it to empty string (''):

COMMENT ON TABLE tablename IS *'';*

3.Including constraints

3.1.What are constraints

The Oracle server uses constraints to prevent invalid data entry into tables.
Constraints are used to do the following:

- Enforce rules on the data in a table whenever a row is inserted, updated or deleted from that table.
- Prevent the deletion of a table if there are dependencies from other tables.
- Provide rules for Oracle tools, such as Oracle Developer.

Here is a list of the data integrity constraints:

1. **NOT NULL**
2. **UNIQUE**
3. **PRIMARY KEY**
4. **FOREIGN KEY**
5. **CHECK**

If you don't name a constraint when creating it, the Oracle Server generates a name by using the *SYS_Cn* format.

These constraints can be created at the same time as the table is created, or after using the **ALTER TABLE** statement.
All constraints are stored in the data dictionary.

Syntax:

```
CREATE TABLE   [schema.]table
               (column datatype [DEFAULT expr]
               [column constraint],
               …
               [table_constraint][, … ]);
```

Schema:	Is the same as the user's login.
Table:	Is the name of the table
DEFAULT expr:	Specifies a default value to use if a value is omitted in the **INSERT** statement.
Column:	Is the name of the column.
Datatype:	Is the column's data type.
Column_constraint:	Is an integrity constraint as part of the column definition.
Table_constraint:	Is an integrity constraint as part of the table definition.

3.2.Defining constraints

3.2.1.The NOT NULL constraint

The **NOT NULL** constraint ensures that the column contains no null values.
This constraint is defined at the column level.

The following example creates the EMPLOYEE table which contains two columns with the **NOT NULL** constraint.

```
 SQL> CREATE TABLE        employee
   2                      (name        VARCHAR2(30) NOT NULL,
   3                      phone        VARCHAR2(30)        CONSTRAINT
employee_phone_nn NOT NULL);

 Table created.
```

You can verify that the constraint has been added, by inserting a null value into the name and phone columns.

```
 SQL> INSERT INTO  employee
   2  VALUES              (NULL,NULL);

 INSERT INTO employee
 *
 ERROR at line 1 :
 ORA-01400:    impossible    to    insert    null    value    in
("BENKIR_R"."EMPLOYEE"."NAME")
```

3.2.2.The UNIQUE constraint

The **UNIQUE** integrity constraint requires that every value in a column or a set of columns be unique. If the **UNIQUE** constraint comprises more than one column, that group of columns is called a *composite unique key* or else, it's called the *unique key.*

This constraint can be defined at the column level or the table level, and it allows the input of nulls unless the **NOT NULL** constraint is also defined for the same columns.

The following example creates a table called DEPARTMENT2, which contains the EMAIL column with the dept2_email_u **UNIQUE** constraint.

```
 SQL> CREATE TABLE department2(
   2              email VARCHAR2(30) CONSTRAINT dept2_email_u UNIQ
 UE);

 Table created.
```

You can issue two **INSERT** statements with the same v alue, to ensure that the constraint has been applied.

```
 SQL> INSERT INTO department2
   2  VALUES     ('oracle@oracle.com');
 INSERT INTO department2
 *
 ERROR at line 1 :
 ORA-00001: Unique constraint violation (BENKIR_R.DEPT2_EMAIL_U)
```

3.2.3.The PRIMARY KEY constraint

The **PRIMARY KEY** constraint creates a primary key for the table.
Only ONE primary key can be created for a table.

The primary key can be a column or a set of columns, and it enforces uniqueness of the column or the set of columns and ensures that no column that is part of the primary key can contain a null value.

The following statement creates a table called EMP, and defines a **PRIMARY KEY** constraint on the EMPNO column, and a **UNIQUE** constraint on the EMAIL column.

```
SQL> CREATE TABLE   emp(
  2                      empno NUMBER(8) PRIMARY KEY,
  3                      ename VARCHAR2(50),
  4                      email VARCHAR2(50) UNIQUE);

Table created.
```

3.2.4.The FOREIGN KEY constraint

This constraint designates a column or combination of columns as a foreign key and establishes a relationship between a primary key, or a unique key in the same table or a different table.

Note that a foreign key MUST match an existing value in the parent table or be NULL.

FOREIGN KEY constraint keywords:

Keyword	Description
FOREIGN KEY	Defines the column in the child table at the table constraint level.
REFERENCES	Identifies the table and column in the parent table.
ON DELETE CASCADE	Deletes the dependent rows in the child table when a row in the parent table is deleted.
ON DELETE SET NULL	Converts dependent foreign key values to NULL when the parent value is removed.

The following statement creates a table called EMPLOYEES, and defines a **FOREIGN KEY** constraint on the DEPTNO column, using table-level syntax. The name of the constraint is EMP_DEPTNO_FK.

```
SQL> CREATE TABLE   employees(
  2                 empno   NUMBER(8) PRIMARY KEY,
  3                 deptno NUMBER(8) FOREIGN KEY REFERENCES dept(deptno));

Table created.
```

3.2.5.The CHECK constraint

This constraint defines a condition that each row must satisfy.
The condition can use the same constructs as query conditions.
Note that some expressions are cannot be used with the **CHECK** constraint.

- References to the CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudo columns.
- Calls the SYSDATE, UID, USER, and USERENV functions.
- Queries that refer to other values in other rows.

Note that a single column can have more than one **CHECK** constraints.

The following statement creates a table called EMP3, with the **CHECK** constraint.
The constraint ensures, that all employees have a salary at least equal to 20 $.

```
SQL> CREATE TABLE emp3   (
  2                      ename VARCHAR2(2),
  3                      sal NUMBER CHECK (sal>20));

Table created.
```

3.3.Managing constraints

3.3.1.Adding constraints

It's possible to add constraints to an existing table by using the **ALTER TABLE** statement with the **ADD CONSTRAINT** clause.

Syntax:

ALTER TABLE *tablename*
ADD [CONSTRAINT *constraint*] *type (column);*

Tablename:	Is the name of the table.
Constraint:	Is the name of the constraint.
Type:	Is the type of the constraint.
Column:	Is the name of the column affected by the constraint.

If you want to add a **NOT NULL** constraint to an existing column, you must use the **MODIFY** clause.
Note that the table must be empty or the column must have a value for every row.

The following statement adds a PRIMARY KEY constraint called emp3_pk on the emp3 table.

```
SQL>  ALTER TABLE                emp3
   2  ADD CONSTRAINT     emp3_pk PRIMARY KEY (ename);

Table altered.
```

3.3.2.Dropping constraints

To drop a constraint from a table, you just have to use the **DROP CONSTRAINT** clause with the **ALTER TABLE** statement.

Syntax:

ALTER TABLE	*tablename*
DROP CONSTRAINT	*constraint [**CASCADE**];*

Tablename:	is the name of the table.
Constraint:	Is the constraint to drop.

The **CASCADE** option causes any dependent constraints also be dropped.

For example, if you attempt to drop a **PRIMARY KEY** constraint, you should use the **CASCADE** option, to drop any dependent **FOREIG KEY**.

```
SQL> ALTER TABLE  dept
  2        DROP PRIMARY KEY CASCADE;

Table altered.
```

All **FOREIGN KEY** that references to this **PRIMARY KEY** are dropped.

3.3.3.Disabling constraints

If you don't want to drop a constraint, but disable it, you can use the **DISABLE** clause with the **ALTER TABLE** statement.

Syntax:

ALTER TABLE	*tablename*
DISABLE CONSTRAINT	*constraint [**CASCADE**];*

Tablename:	Is the name of the table.
Constraint:	Is the name of the constraint.

The **CASCADE** option is used, to disable all dependent constraints.

The following statement disables the **PRIMARY KEY** constraint on the EMP3 table.

```
SQL> ALTER TABLE          emp3
  2  DISABLE CONSTRAINT  emp3_pk;

Table altered.
```

3.3.4.Enabling constraints

You can enable a constraint without dropping it and re-creating it by using the **ENABLE** clause with the **ALTER TABLE** statement.

Syntax:

ALTER TABLE	*tablename*
ENABLE CONSTRAINT	*constraint [**CASCADE**];*

Tablename:	Is the name of the table.
Constraint:	Is the name of the constraint.

Here are some rules that must be respected when enabling constraints:

1. All the data in the table must fit the constraint.
2. When enabling a **UNIQUE** key or a **PRIMARY KEY** constraint, a **UNIQUE** or **PRIMARY KEY** index is created automatically.
3. Enabling a primary constraint with the **CASCADE** option does not enable foreign key that references to this primary key.

The following statement enables the **PRIMARY KEY** constraint on the EMP3 table.

```
SQL> ALTER TABLE          emp3
  2  ENABLE CONSTRAINT    emp3_pk;

Table altered.
```

3.3.5.Cascading constraints

The **CASCADE CONSTRAINTS** clause is used to drop all referential integrity constraints that refer to the primary and unique keys that are going to be dropped.
This clause drops all multicolumn constraints defined on the dropped columns.

Example:

```
SQL> ALTER TABLE  dept
  2  DROP (deptno) CASCADE CONSTRAINTS;

Table altered.
```

3.3.6.Viewing constraints

With the **DESCRIBE** command, you can only verify the existence of the **NOT NULL** constraint.
To view all constraints, you must query the USER_CONSTRAINTS table.

The following statement displays the constraints on the DEPT table.

```
SQL> SELECT       constraint_name, constraint_type
  2  FROM         user_constraints
  3  WHERE              table_name='DEPT';


     CONSTRAINT_NAME                C
     ------------------------------ -
     DEPT_DEPTNO_PK                 P
```

Note that if the user does not specify a constraint name, the Oracle Server will do it automatically.

You can view the names of the columns involved in constraints by querying the USER_CONS_COLUMNS data dictionary view.

```
SQL> SELECT       constraint_name, column_name
  2  FROM         user_cons_columns
  3  WHERE              table_name='DEPT';

     CONSTRAINT_NAME                COLUMN_NAME
     ------------------------------ ----------------------
     DEPT_DEPTNO_PK                 DEPTNO
```

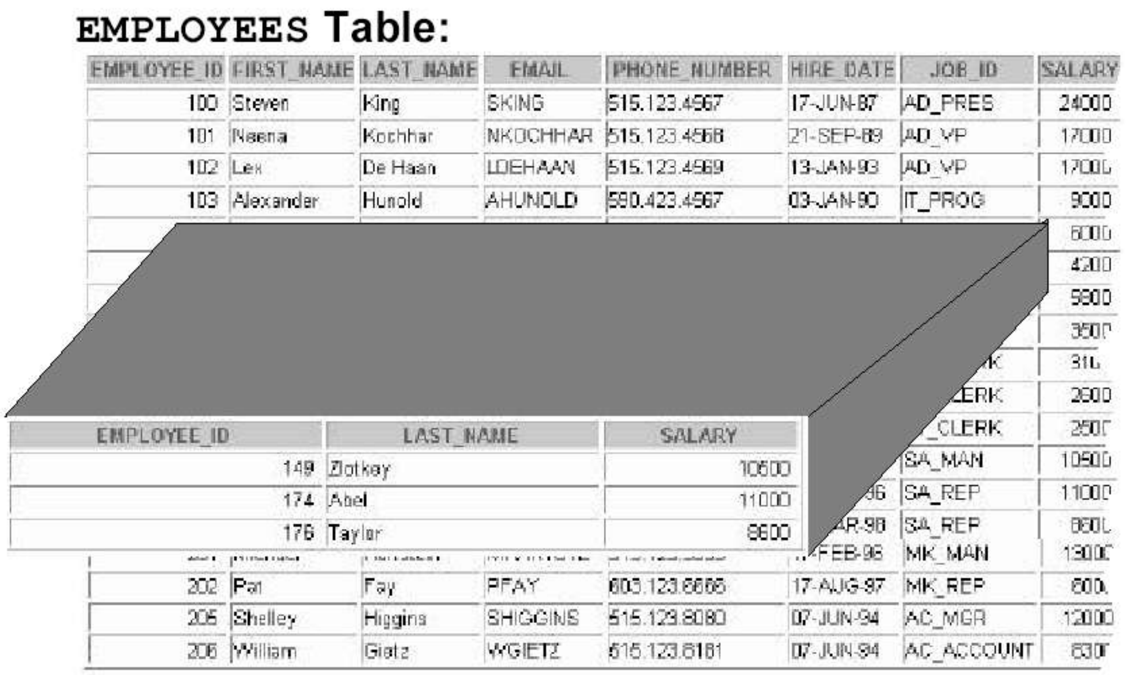
4.Creating views

4.1.Presentation

4.1.1.Database objects

Object	Description
Table	Basic unit of storage; composed of rows and columns.
View	Logically represents subsets of data from one or more tables.
Sequence	Generates primary key values
Index	Improves the performance of some queries
Synonym	Alternative name for an object.

4.1.2.What is a view



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000
							6000
							4200
							5800
							3500
						K	3100
						ERK	2600
						_CLERK	2500
						SA_MAN	10500
					96	SA_REP	11000
					AR-98	SA_REP	8600
					FEB-96	MK_MAN	13000
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	6000
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	12000
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	8300

EMPLOYEE_ID	LAST_NAME	SALARY
149	Zlotkey	10500
174	Abel	11000
176	Taylor	8600

A view is a logical table based on a table or another view. It doesn't contain any data.

The view is like window through which data of a table can be selected or changed; the table on which the view is based, is called base table.

Views are stored as a **SELECT** statement in the data dictionary.

4.1.3.Why use views

Here are the main advantages of views:

- Views restrict access to the data, because the view can display selective columns from a table.

- Views can be used to make simple queries to retrieve the results of complicated queries.
- One view can be used to retrieve data from more than one table.

There are two classifications for views: Simple views and complex views.

Feature	Simple	Complex
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML operation	Yes	Not always

4.2.Managing views

4.2.1.Creating views

To create a view, you have to use the **CREATE VIEW** statement.

Syntax:

```
CREATE [OR REPLACE] VIEW
 [FORCE|NOFORCE]                 viewname
                                 [(alias)]
AS                               subquery
[WITH CHECK OPTION               [CONSTRAINT constraint]
[WITH READ ONLY                  [CONSTRAINT constraint];
```

OR REPLACE:	Re-creates the view if it already exists.
FORCE:	Creates the view regardless of whether or not the base table exists.
NOFORCE:	Creates the view only if the base table exists.
Viewname:	Is the name of the table.
Alias:	Specifies names for the expressions selected by the view's query.
Subquery:	Is a complete **SELECT** statement.
WITH CHECK OPTION:	Specifies that only rows accessible to the view can be inserted or updates.
Constraint:	is the name assigned to the CHECK OPTION constraint.
WITH READ ONLY:	Ensures that no DML operation can be performed.

The following statement creates the EMPVU view based on the EMP table.
The view contains the employee number, last name, and salary for each employee in department 30.

```
SQL> CREATE OR REPLACE VIEW    empvu
  2  AS
  3        SELECT       empno, ename,sal
  4        FROM         emp
  5        WHERE        deptno=30;

View created.
```

You can issue a **DESCRIBE** command, to ensure that the view has been created.

```
SQL> DESCRIBE empvu;

 Name                                      NULL ?   Type
 ----------------------------------------- -------- -------------

 NUMBERID                                  NOT NULL NUMBER(4)
 LASTNAME                                           VARCHAR2(10)
 SALARY                                             NUMBER(7,2)
```

It's possible to control the column names by including column aliases in the subquery.
The number of aliases must match the number of columns selected in the subquery statement.

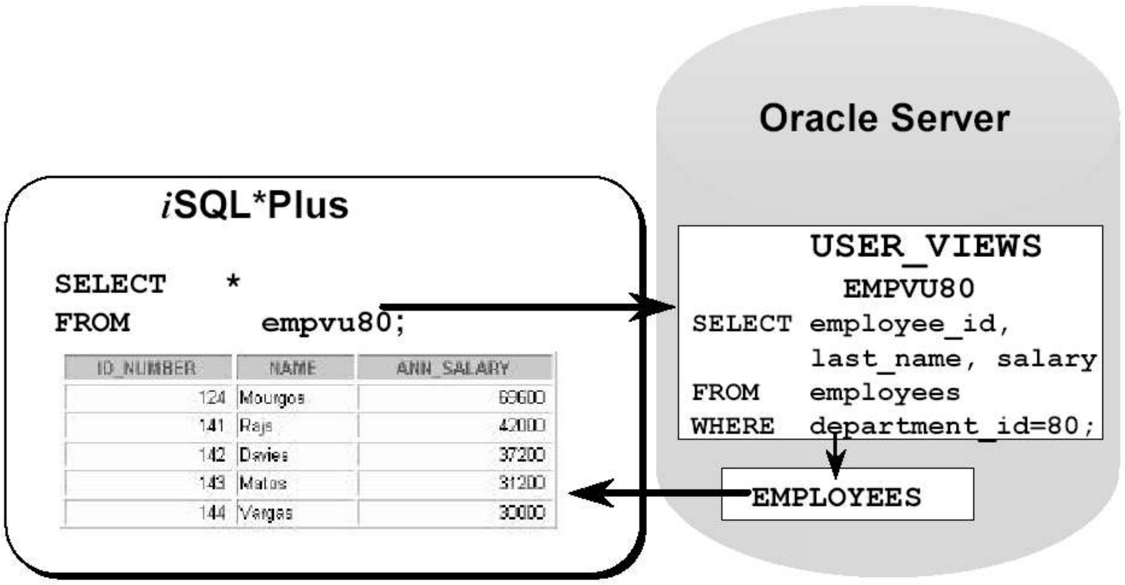
The following statement creates the EMPVU2 view based on the EMP table.
The view contains the employee number, last name, and salary for each employee in department 30, with the aliases NUMBER_ID, and LAST_NAME.

```
SQL> CREATE OR REPLACE VIEW    empvu
  2  AS
  3        SELECT       empno number_id,ename last_name, sal
  4        FROM         emp
  5        WHERE        deptno=30;

View created.
```

4.2.2.Retrieving data from a view

Retrieving data from a view can be done by a simple **SELECT** statement.

When a user accesses data from a view the Oracle server, retrieves the view definition from the USER_VIEWS table.
Then, it checks the privileges for the view base table, and finally the Oracle Server performs the operation in the base table.



```
SELECT    *
FROM         empvu80;
```

ID_NUMBER	NAME	ANN_SALARY
124	Mourgos	69600
141	Rajs	42000
142	Davies	37200
143	Matos	31200
144	Vargas	30000

```
SELECT employee_id,
       last_name, salary
FROM   employees
WHERE  department_id=80;
```

4.2.3.Modifying a view

If a view was created with the **OR REPLACE** option, it's possible to modify the view's structure by re-creating it.

The EMPVU view created contains the *empno, ename, and sal* columns.

```
SQL> DESCRIBE empvu;

 Name                                      NULL ?   Type
 ----------------------------------------- -------- -------------

 EMPNO                                     NOT NULL NUMBER(4)
 ENAME                                              VARCHAR2(10)
 SAL                                                NUMBER(7,2)
```

The following statement modifies the EMPVU view by adding the COMMISION column.

```
SQL> CREATE OR REPLACE VIEW    empvu
  2  AS
  3         SELECT       empno, ename, sal, comm
  4         FROM         emp
  5         WHERE        deptno=30;

View created.
```

You can issue the **DESCRIBE** command to ensure that changes have been made.

```
SQL> DESC empvu
 Nom                                       NULL ?   Type
 ----------------------------------------- -------- ---------------

 EMPNO                                     NOT NULL NUMBER(4)
 ENAME                                              VARCHAR2(10)
 SAL                                                NUMBER(7,2)
 COMM                                               NUMBER(7,2)
```

4.2.4.Creating a complex view

Two classes of views exist: Simple views and complex views.

The following example creates a complex view which contains group functions.
The view created is based on the department names, minimum salaries, maximum salaries, and average salaries by department.

```
SQL> CREATE OR REPLACE VIEW    deptvu
  2                            (name, minsal, maxsal, avgsal)
  3  AS
  4         SELECT       d.dname, MIN(e.sal), MAX(e.sal), AVG(e.sal)
  5         FROM         dept d, emp e
  6         WHERE        e.deptno=d.deptno
  7         GROUP BY     d.dname;

View created.
```

You can issue a **DESCRIBE** command to ensure that the complex view has been created.

```
SQL> DESCRIBE deptvu;

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

     NAME                                                     VARCHAR2
(14)
     MINSAL                                               NUMBER
     MAXSAL                                               NUMBER
     AVGSAL                                               NUMBER
```

4.2.5.Removing views

When removing a view, data are not lost, because a view is based on underlying tables in the database. To remove view, you just have to use the **DROP VIEW** statement.

Syntax:

DROP VIEW *viewname*;

Viewname: Is the name of the view to drop.

Note that you cannot roll back this operation.

The following statement drops the DEPTVU view from the database.

```
SQL> DROP VIEW deptvu;

View dropped.
```

4.3.DML operations on a view

4.3.1.Rules for performing DML operations on a view

You can perform DML operations on simple views, but note that it is not possible to remove rows from a view that contains:

- Group functions.
- A **GROUP BY** clause.
- The **DISTINCT** keyword.
- The pseudo column ROWNUM keyword.

You can add data through a view if it doesn't contain any listed items, or if there are not **NOT NULL** columns without default values in the base table that are not selected by the view.

Note that you are adding data directly in the base table.

4.3.2.Using the WITH CHECK OPTION clause

The **WITH CHECK OPTION** clause specifies that **INSERTs** and **UPDATEs** cannot create rows that cannot be selected by the view, and therefore it allows integrity constraints and data validation checks to be enforced on data being inserted or updated.

The EMPVU2 is created with the **WITH CHECK OPTION**.

```
 SQL> CREATE OR REPLACE VIEW    empvu2
   2  AS
   3                            SELECT       *
   4                            FROM         emp
   5                            WHERE        deptno=20
   6                            WITH    CHECK    OPTION    CONSTRAINT
empvu20_ck;

 View created.
```

The following statement failed, because it attempts to change the entire department id to 10, and the view would no be able to see the employees, because it can see only employees working into the department 20.

```
 SQL> UPDATE       empvu2
   2  SET          deptno=10;

      UPDATE empvu2
             *
      ERROR at line 1 :
    ORA-01402: view WITH CHECK OPTION WHERE-clause violation
```

4.3.3.Denying DML operations

You can ensure that no DML operations are performed on the view you created by using the **WITH READ ONLY** option when creating or modifying it.
If you create a view with this option, any DML operation will fail and an Oracle Server error will occur.

The following statement creates a view based on the dept table with the **WITH READ ONLY** option.

```
 SQL> CREATE OR REPLACE VIEW    deptvu3
   2  AS
   3                            SELECT       *
   4                            FROM         dept
   5                            WITH     READ     ONLY     CONSTRAINT
deptvu3_ck;

 View created.
```

This statement fails, because it attempts to perform a DML operation on a view created with the **WITH READ ONLY** option.

```
 SQL> UPDATE deptvu3
   2  SET dname='Oracle';

      SET dname='Oracle'
      *
      ERROR at line 2 :
      ORA-01733: virtual columns are not allowed here.
```

```
 SQL> DELETE FROM deptvu3
   2  WHERE deptno=10;

      DELETE FROM deptvu3
                  *
      ERROR at line 1 :
 ORA-01752:  cannot  delete  from  view  without  exactly  one  key-
preserved table
```

4.4.Inline views

An INLINE VIEW is a subquery with an alias which can be used in a SQL statement.
This type of view is created by placing a subquery within the **FROM** clause of a **SELECT** statement.
It represents a data source that can be referenced in the main query.

In the following example, the inline view *B* returns the detail of all department numbers and the maximum salary for each department from the EMP table.

The **WHERE a.deptno=d.deptno AND a.sal <b.maxsal** clause of the main query displays employee names, salaries, department numbers, and maximum salaries for all employees who earn less than the maximum salary in their department.

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


     ENAME             SAL     DEPTNO     MAXSAL
     ---------- ---------- ---------- ----------
     CLARK            2450         10       5000
     MILLER           1300         10       5000
     SMITH             800         20       3000
     ADAMS            1100         20       3000
     JONES            2975         20       3000
     ALLEN            1600         30       2850
     MARTIN           1250         30       2850
     JAMES             950         30       2850
     TURNER           1500         30       2850
     WARD             1250         30       2850

     10 row(s) selected.
```

4.5.TOP-N analysis

You can choose to display only the *n* top-most or the *n* bottom-most records from a table based on a condition using **TOP-N** queries.

For example, you can choose to display:

- The top three earners in a company.
- The four most recent hired employees.
- The top three products that have had the maximum sales in last month.

Syntax:

```
SELECT      [column list], ROWNUM
FROM
                  (SELECT     [column list]
                  FROM              table
                  ORDER BY   Top-n column)

WHERE ROWNUM condition;
```

Subquery: Or inline view that generates sorted data. It includes the **ORDER BY** clause to ensure that the ranking is in the desired order.

Outerquery: Limits the number of rows in the final result.

ROWNUM: Pseudo column which assigning a sequential value.

WHERE: Specified the *n* rows to be returned.

The following statement displays the names and the salaries of the top four employees from the EMP table.
The subquery returns the details of all employees' names and salaries from the EMP table, sorted in the descending order of the salaries.
The **WHERE ROWNUM <=4** clause of the main query ensures that only the first four records are returned.

```
SQL> SELECT       ROWNUM, e.ename,e.hiredate
  2  FROM         (SELECT ename, hiredate
  3               FROM emp
  4               ORDER BY hiredate) e
  5  WHERE ROWNUM < = 4;

       ROWNUM ENAME      HIREDATE
   ---------- ---------- --------
            1 SMITH      17/12/80
            2 ALLEN      20/02/81
            3 WARD       22/02/81
            4 JONES      02/04/81
```

5.Other objects

5.1.Sequences

5.1.1.What is a sequence

A sequence is a database object created by the user.
The typical use for sequence is to create primary key values, which must be unique.
The sequence is generated and incremented (or decremented) by an internal Oracle routine.

Note that sequences are stored and generated independently of table, so it can be used by more than one table.

A sequence can replace application code.

5.1.2.The CREATE SEQUENCE statement

Syntax:

```
CREATE SEQUENCE      sequencename
        [INCREMENT BY n]
        [START WITH n]
        [{MAXVALUE n}]
        [{MINVALUE n}]
        [{CYCLE | NOCYCLE}]
        [{CACHE n | NOCACHE}];
```

Sequencename: Is the name of the sequence

INCREMENT BY *n*: Specifies the interval between sequence numbers where *n* is an integer. (If it's omitted, n=1)

START WITH *n*: Specifies the first value of the sequence numbers. (If it's omitted, *n*=1)

MAXVALUE *n*: Specifies the maximum value the sequence can generate.

MINVALUE *n*: Specifies the minimum sequence's value.

CYCLE | NOCYLE: Specifies whether the sequence continues to generate values after reaching its max or min value. (NOCYLE is the default option)

CACHE *n* | NOCACHE: Specifies how many values the Oracle Server reallocates and keeps in memory.

The following statement creates the DEPT_DEPTNO_SEQ, which will be used for the *deptno* column of the DEPT table.
The sequence starts to 10, is incremented by 10 and does no allow caching and cycle.

```
SQL> CREATE SEQUENCE     dept_deptno_seq
  2     INCREMENT BY     10
  3     START WITH       10
  4     NOCYCLE
  5     NOCACHE;

Sequence created.
```

5.1.3.Confirming sequences

You can verify that the sequence has been created by querying the USER_OBJECTS data dictionary table.

```
SQL> SELECT       object_name
  2  FROM         user_objects
  3  WHERE              object_type='SEQUENCE';

     OBJECT_NAME
     --------------------------------------

     DEPT_DEPTNO_SEQ
     EDITEUR_ID_SEQ
     ID_CIRCUIT_SEQ
     ID_CLIENT_SEQ
     ID_ETAPES_SEQ
     ID_HOTEL_SEQ
     ID_SITE_TOURISTIQUE_SEQ
     ...
     12 row(s) selected.
```

You can also have information about it by querying the USER_SEQUENCES data dictionary table.

```
SQL> SELECT       min_value, max_value, increment_by, last_number
  2  FROM         user_sequences
  3  WHERE              sequence_name='DEPT_DEPTNO_SEQ';


      MIN_VALUE  MAX_VALUE INCREMENT_BY LAST_NUMBER
     ---------- ---------- ------------ -----------
              1 1,0000E+27           10          10
```

5.1.4.NEXTVAL and CURRVAL pseudo columns

After you create a sequence, you can use the NEXTVAL and CURRVAL pseudo columns to retrieve the next and the current value of this sequence.

The NEXTVAL is used to extract successive sequence numbers from a specific sequence.
When using it, you must specify the NEXTVAL pseudo column with the name of the sequence.
Note that when you use NEXTVAL, a new sequence number is generated and the current value is placed in CURRVAL.

```
SQL> SELECT       dept_deptno_seq.NEXTVAL
  2  FROM         dual;


     NEXTVAL
     ---------
            30
```

The CURRVAL pseudo column is used to refer to a sequence number that the current user has just generated.

If the sequence is called for the first time, you MUST use the NEXTVAL pseudo column before using the CURRVAL pseudo column.
As for the NEXTVAL pseudo column, you MUST specify the CURRVAL with the name of the sequence.

```
SQL> SELECT        dept_deptno_seq.NEXTVAL
  2  FROM          dual;

   NEXTVAL
   ----------
          50

SQL> SELECT        dept_deptno_seq.CURRVAL
  2  FROM          dual;

   CURRVAL
   ----------
          50
```

You can use NEXTVAL and CURRVAL in the following contexts:

- The **SELECT** list of a **SELECT** statement that is not part of a subquery.
- The **SELECT** list of a subquery in an **INSERT** statement.
- The **VALUES** clause of an **INSERT** statement.
- The **SET** clause of an **UPDATE** statement.

You cannot use the NEXTVAL and CURRVAL in the following contexts:

- The **SELECT** list of a view.
- A **SELECT** statement with the **DISTINCT** keyword.
- A **SELECT** statement with the **GROUP BY, ORDER,** or **HAVING** clause.
- A subquery in a **SELECT, UPDATE** or **DELETE** statement.
- The **DEFAULT** expression in a **CREATE TABLE** or **ALTER TABLE** statements.

5.1.5.Using a sequence

Sequences can be used to insert values in a table using the **INSERT INTO** statement.

The following example inserts a new department into the DEPT table.
It uses the *dept_deptno_seq* sequence to generate a new department number.

```
SQL> INSERT INTO   dept
  2                (deptno, dname, loc)
  3  VALUES        (dept_deptno_seq.NEXTVAL,'SUPPORT','NY');

1 row created.
```

Now, the following example inserts a new employee into the EMP table.
This employee works in the SUPPORT department.

```
SQL> INSERT INTO   emp(empno,ename,hiredate,deptno)
  2  VALUES        (7777,'GEROGES',SYSDATE,dept_deptno_seq.CURRVAL);

1 row created.
```

Note that caching sequence values provides faster access.
The cache is populated the first time you refer to the sequence, and then the next value is retrieved from the cached sequence.

5.1.6.Modifying a sequence

You can modify a sequence using the **ALTER SEQUENCE** statement.
Thanks to this statement, you can change the increment value, the maximum value, the minimum value, the cycle option, or the cache option.

Syntax:

ALTER SEQUENCE *sequencename*
[INCREMENT BY n]
[MAXVALUE n | NOMAXVALUE]
[MINVALUE n | NOMINVALUE]
[CYLE | NOCYCLE]
[CACHE n | NOCACHE]

Sequencename: Is the name the sequence to change.

To modify a sequence, you must be the owner of the sequence, or have the **ALTER** privilege on the sequence.
The **START WITH** option can be changed only by dropping the sequence and recreating it.
A new **MAXVALUE** value cannot be less than the current value of the sequence that is changed.

The following statement attempts to change the **MAXVALUE** option of the dept_deptno_seq, by a value less than its current value.

```
 SQL> ALTER SEQUENCE      dept_deptno_seq
   2  MAXVALUE            30;

 ALTER SEQUENCE dept_deptno_seq
 *
 ERROR at line 1 :
 ORA-04009:  MAXVALUE  cannot  be  made  to  be  less  than  the  current
value.
```

5.1.7.Removing a sequence

To remove a sequence from the database, you just have to use the **DROP SEQUENCE** statement.
Note that you must be the owner of the sequence or have the **DROP ANY SEQUENCE** privilege to perform this operation.

Syntax:

DROP SEQUENCE *sequencename;*

Sequencename: Is the name of the sequence to drop.

This statement drops the *dept_deptno_seq* sequence.

```
 SQL> DROP SEQUENCE dept_deptno_seq;

 Sequence dropped.
```

5.2.Indexes

5.2.1.What is an index

An index is a schema object that can speed up the retrieval of rows from a table using pointers. There are to ways to create indexes, they can be created automatically by the Oracle Server, or manually by the user.

An index is used to reduce the disk I/O operations, it's used and maintained automatically by the Oracle Server; there is no direct activity required by the user.
They are logically and physically independent from tables, so it means that they can be created and dropped without altering any database's table.

But note that when you drop an indexed table, corresponding indexes are also dropped.

When you define a **PRIMARY KEY** or a **FOREIGN KEY** on a table, indexes are automatically created, and their name is the same as the name given to the constraints.

5.2.2.Creating an index

To create an index on a table, you have to use the **CREATE INDEX** statement.

Syntax:

CREATE INDEX	*indexname*
ON	*table (column1, column2 ...);*

Indexname:	Is the name of the index to create.
Table:	Is the name of the table.
Column:	Is the name of the column in the table to be indexed.

When to create an index:

- The column contains a wide range of values.
- The column contains a large number of null values.
- One or more columns are frequently used in a **WHERE** clause or join conditions.
- The table is large and most queries are expected to retrieve less than 2-4 percent of the rows.

When not to create an index:

- The table is small.
- The columns are not often used as a condition in the query.
- Most queries are expected to retrieve more than 2 to 4 percent of the rows table.
- The table is updated frequently.
- The indexed columns are referenced as part of an expression.

The following statement creates an index to improve the speed of query access to the *ename* column in the EMP table.

```
SQL> CREATE INDEX        emp_ename_idx
  2  ON                  emp(ename);

Index created.
```

5.2.3.Confirming an index

You can confirm the existence of created indexes by querying the USER_INDEXES data dictionary table, and you can also check the columns involved in an index by querying the USER_INS_COLUMNS data dictionary table.

The following statement displays all the created indexes, with the names of affected column, and the index's uniqueness in the EMP table.

```
SQL> SELECT     cix.index_name, cix.column_name,
  2             cix.column_position, uix.uniqueness
  3  FROM       user_indexes uix, user_ind_columns cix
  4  WHERE      cix.index_name = uix.index_name
  5  AND        cix.table_name='EMP';


INDEX_NAME        COLUMN_NAME        COLUMN_POSITION    UNIQUENES
----------------  ----------------   ---------------    ---------
EMP_ENAME_IDX     ENAME                            1    NONUNIQUE
```

5.2.4.Function-Based indexes

A function-based index is an index based on expressions.
These indexes facilitate processing queries which contain these expressions.

For example, function-based indexes defined with the **UPPER** or **LOWER** keywords allow case insensitive searches.

The following index facilitates processing queries on the *ename* column of the EMP table.

```
SQL> CREATE INDEX        upper_emp_ename_idx
  2  ON                  emp(UPPER(ename));

Index created.
```

You must be sure that the value of the function is not null in subsequent queries, to ensure that the index is used by the Oracle Server.

5.2.5.Removing an index

It is not possible to modify an index, you have to drop it and then re-create it.
To drop an index, you just have to use the **DROP INDEX** statement.

Syntax:

DROP INDEX *indexname;*

Indexname: Is the name of the index to drop.

To perform this operation, you MUST be the owner of the index to drop, or have the **DROP ANY INDEX** privilege.

The following statement drops the EMP_ENAME_IDX index.

```
SQL> DROP INDEX    emp_ename_idx;

Index dropped.
```

5.3.Synonyms

5.3.1.What is a synonym

If a user wants to refer to a table owned by another user, he must prefix the name of the table by the name of the user who created it.

A synonym eliminates the need to qualify the object name with the schema and provides you with an alternative object name.

Note that the object cannot be contained in a package, and that a private synonym must be distinct from all other objects owned by the same user.

5.3.2.Creating a synonym

To create a synonym, you just have to perform the **CREATE SYNONYM** statement.

Syntax:

CREATE [PUBLIC] SYNONYM *synonym*
FOR *object;*

PUBLIC: Specifies that the synonym will be accessible by all users.

Synonym: Is the name of the synonym.
Object: Is the name of the object for which the synonym is created.

The following statement creates the EMPLO synonym for the EMP table.

```
SQL> CREATE SYNONYM     emplo
  2  FOR                emp;

Synonym created.
```

Now, you can retrieve information from the EMP table using the EMPLO synonym.

```
SQL> SELECT       ename
  2  FROM         emplo;

    ENAME
    ----------
    SMITH
    ALLEN
    WARD
    JONES
    MARTIN
    BLAKE
    CLARK
    TURNER
    ADAMS
    ...
  15 row(s) selected.
```

5.3.3.Removing a synonym

To drop a synonym, you just have to use the **DROP SYNONYM** statement.

Syntax:

DROP SYNONYM *Synonymname;*

Synonymname:	Is the name of the synonym to drop.

The following statement drops the EMPLO synonym.

```
SQL> DROP SYNONYM        emplo;

Synonym droppped.
```

6.Controlling user access

6.1.System privileges

6.1.1.What are privileges

Privileges are the right to execute particular statements on the database.
The high-level user is the DBA, he has the ability to grant users access to the database and its objects.
Users need *system privileges* to connect to the database and *object privileges* to manipulate the content of the database objects.

6.1.2.Typical DBA privileges

There are more than one hundred *system privileges* available for users and roles.
These *system privileges* are typically provided by the DataBase Administrator.

System privilege	Operations authorized
CREATE USER	Grantee can create other Oracle's users.
DROP USER	Grantee can drop another user.
DROP ANY TABLE	Grantee has the ability to drop any table in any schema.
BACKUP ANY TABLE	Grantee can back up any table in any schema.
SELECT ANY TABLE	Grantee can query tables, views, or snapshots in any schema.
CREATE ANY TABLE	Grantee can create tables in any schema.

6.1.3.Creating users

The DBA can create users by executing the **CREATE USER** statement.
When the user is created, he doesn't have any privilege; the DBA can after the creation of the user grant privileges to him.

Syntax:

CREATE USER *user*
IDENTIFIED BY *password;*

User:	Is the name of the user created.
Password:	Specifies that the user must log in with this password.

6.1.4.User system privileges

Once the DBA has created a user, he can assign to him some privileges.

For example, if the DBA wants that the user be able to log in the database, he must assign to him the **CREATE SESSION** privilege.

Syntax:

GRANT *privilege [,privilege, ...]*
TO *user [, user, role, PUBLIC ...];*

Privilege: Is the system privilege to be granted.
User| role| PUBLIC: Is the name of the user, or the name of the role or PUBLIC designates that every user is granted the privilege.

Privilege	**Operation authorized**
CREATE SESSION	Connect to the database
CREATE TABLE	Create tables in the user's schema
CREATE SEQUENCE	Create sequences in the user's schema
CREATE VIEW	Create views in the user's schema
CREATE PROCEDURE	Create stored procedures, functions, or packages in the user's schema.

6.1.5.Granting system privileges

The following statements do the following operations:

- Creates the SCOTT2 user.
- Grants the CREATE SESSION system privilege to SCOTT.
- Grants to SCOTT the CREATE TABLE system privileges.

```
SQL> CREATE USER         scott2
  2  IDENTIFIED BY       tiger2;

User created.

SQL> GRANT        CREATE SESSION
  2  TO                  scott2;

Grant succeeded.

SQL> GRANT CREATE TABLE
  2  TO                  scott2;

Grant succeeded.
```

6.1.6.Creating and granting privileges to a role

A role is a named group of related privileges that can be granted to users.
A user can have access to several roles, and several users can be assigned to the same role.
Roles are typically created for a database application.

Syntax:

CREATE ROLE *role;*

Role: Is the name of the role created.

Once the role is created, the DBA can use the **GRANT** statement to assign users to that role, and assign privileges to that role.

The following example creates the MANAGER role, and then it gives to all users of this role the ability to create tables and retrieves data from any tables in any schema.

```
SQL> CREATE ROLE         manager;


Role created.


SQL> GRANT CREATE TABLE, SELECT ANY TABLE
  2  TO                  manager;

Grant succeeded.
```

6.1.7.Changing password

To change a password, you have to use the **ALTER USER** statement.

Syntax:

ALTER USER *user*
IDENTIFIED BY *Newpassword;*

User:	Is the name of the user.
Newpassword:	specifies the new password.

The following statement changes the SCOTT2 password to ORACLE.

```
SQL> ALTER USER          scott2
  2  IDENTIFIED BY       oracle;

User altered.
```

6.2.Object privileges

6.2.1.Object privileges

An *object privilege* is a right to perform particular operations on a specific table, view, sequence, or procedure.

The following table lists privileges for various objects:

Object privilege	Table	View	Sequence	Procedure
ALTER	X		X	
DELETE	X	X		
EXECUTE				X
INDEX	X			
INSERT	X	X		
REFERENCES	X	X		
SELECT	X	X	X	
UPDATE	X	X		

Object privileges vary from object to object.
Note that the owner of an object has all object privileges on it, and he can gives all privileges he wants on it.

6.2.2.Granting object privileges

If he wants, the owner of an object can give any object privilege to another user using the **GRANT** statement.
If this statement is used with the **WITH GRANT OPTION** option then, the grantee will be able to give the privilege he was granted to another user.

Syntax:

GRANT	*privilege* **[(column)]**
ON	*objectname*
TO	*username* | *role* | ***PUBLIC***
[WITH GRANT OPTION];	

Privilege:	is an object privilege to be granted.
Column:	specifies the column from a table or a view on which privileges are granted.
Objectname:	Is the object on which the privileges are granted.
Username:	Is the name of the grantee.

Public: Specifies that the privileges are granted to all users.

WITH GRANT OPTION: Allows the grantee to grant the object privileges to other users or roles.

The following statement gives to the user SCOTT the ability to update the EMP table.

```
SQL> GRANT         UPDATE
  2  ON            emp
  3  TO            scott;

Grant succeeded.
```

6.2.3.Using the WHIT GRANT OPTION and PUBLIC keywords

The WITH GRANT OPTION keyword:

A privilege granted with the **WITH GRANT OPTION** clause can be passed to other users and roles by the grantee.
These privileges are revoked when the grantor's privileges are revoked.

The following statement gives user SCOTT access to the DEPT table with privileges to query the table and add rows on it.
The statement also allows SCOTT to give others these privileges.

```
SQL> GRANT                          SELECT, UPDATE
  2  ON                      dept
  3  TO                      scott
  4  WITH GRANT OPTION;

Grant succeeded.
```

The PUBLIC keyword:

An owner of a table can grant access to all users by using the **PUBLIC** keyword.

This example allows all users on the system to query data from the DEPT table.

```
SQL> GRANT                   SELECT
  2  ON            emp
  3  TO PUBLIC;

Grant succeeded.
```

6.2.4.Confirming privileges granted

You can access the data dictionary to view the privileges that you have granted.

Data dictionary view	Description
ROLE_SYS_PRIVS	System privileges granted to role.
ROLE_TAB_PRIVS	Table privileges granted to role.
USER_ROLE_PRIVS	Role accessible by the user.
USER_TAB_PRIVS_MADE	Object privileges granted on the user's object.
USER_TAB_PRIVS_RECD	Object privileges granted to the user.
USER_COL_PRIVS_MADE	Object privileges granted on the column of the object's user.
USER_COL_PRIVS_RECD	Object privileges granted to the user on specific columns
USER_SYS_PRIVS	Lists system privileges granted to the user.

6.2.5.Revoke object privileges

You can remove privileges granted to other users by using the **REVOKE** statement.

Syntax:

REVOKE *privilege[,.., ALL]*
ON *objectname*
FROM *user | role | PUBLIC;*
[CASCADE CONSTRAINTS]

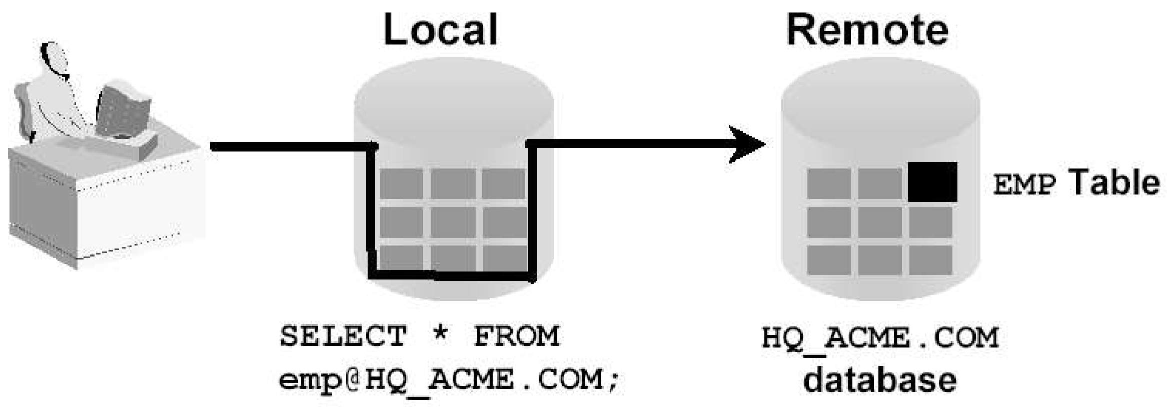
CASCADE CONSTRAINTS: is required to remove any referential integrity constraints made to the object using the **REFERENCES** privilege

The following statement revokes the **SELECT** object privilege given to SCOTT on the EMP table.

```
SQL> REVOKE        SELECT
  2  ON            emp
  3  FROM          scott;

Revoke succeeded.
```

6.3.Database links

Database links allow user to access data on a remote database.



A database link is a pointer that defines a one-way communication path from an Oracle database Server to another database server.

A database link connection is one-way in the sense that a client connected to local database **A** can use a link stored in database **A** to access information in remote database **B**, but users connected to database **B** cannot use the same link to access data in database **A.**
So, if local users on database **B** want to access data on database **A**, they must define a link that is stored in the data dictionary of database **B.**

The great advantage of links is that they allow users to access another user's objects in a remote database so that they are bounded by the privilege set of the object's owner.

Note that the dictionary view USER_DB_LINKS contains information on links to which a user has access.

```
 SQL> DESCRIBE USER_DB_LINKS;


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

  DB_LINK                                  NOT NULL VARCHAR2(128)
  USERNAME                                          VARCHAR2(30)
  PASSWORD                                          VARCHAR2(30)
  HOST                                              VARCHAR2(2000)
  CREATED                                  NOT NULL DATE
```

You can create a database link using the **CREATE DATABASE LINK.**

The example creates a database link. The **USING** clause identifies the service name of a remote database.

```
SQL> CREATE PUBLIC DATABASE LINK      hq.acme.com
  2  USING                                    'sales';

Database link created.
```