MODULE-III SQL and Integrity Constraints [8L] Concept of DDL, DML, DCL. Basic Structure, Set operations, Aggregate Functions, Null Values, Domain Constraints, Referential Integrity Constraints, assertions, views, Nested Subqueries, Database security application development using SQL, Stored procedures and triggers.

Introduction to Query Language

A query language is a language in which a user requests information from the database. These languages are typically of a level higher than that of a standard programming language (4GL). Query Language can be categorized as

- Procedural - In a procedural language the user instruct the system to perform a sequence of operations on the database to compute the desired result.
- Non procedural - In a non procedural Language the user describe the information desired without giving a specific procedure (what not how).

Introduction to SQL

- SQL is an abbreviation for Structured Query Language
- It is a special-purpose, nonprocedural language that supports the definition, manipulation, and control of data in relational database management systems.
- SQL is used to manipulate and retrieve data stored in a database
- SQL is the most commonly used query language available
- The original version called SEQUEL (structured English query language) was designed by an IBM research center in 1974 and 1975.
- Word SQL is derived from "Sequel"
- Depending on the functionality SQL is generally classified as follows:

Name	Expansion
DDL	Data Definition Language
DML	Data Manipulation Language
DCL	Data Control Language
TCL	Transaction Control Language

Table 1: The Sublanguages of SQL

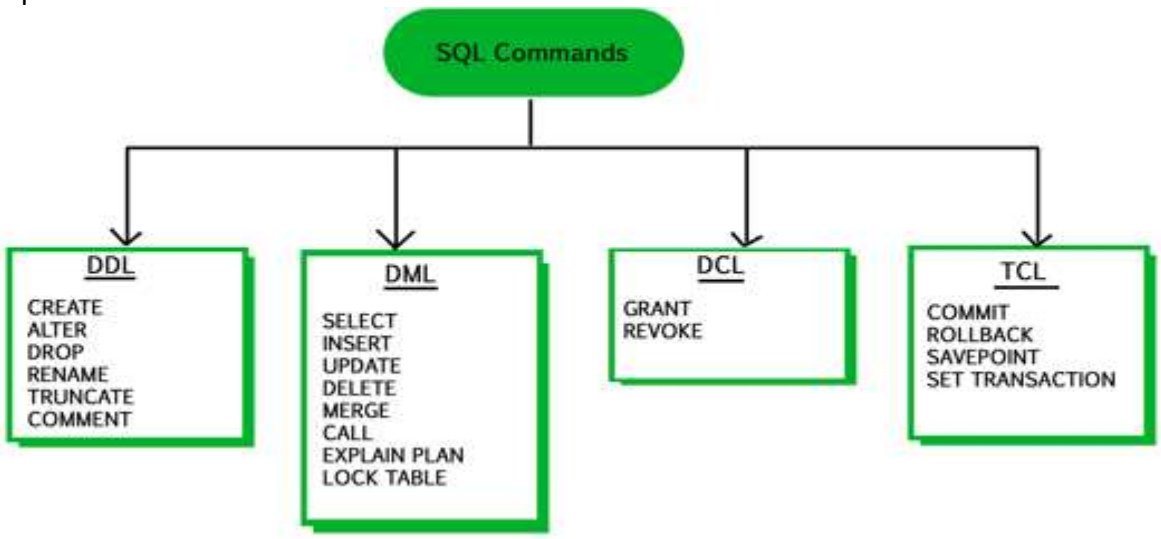
To run SQL we need to follow the steps :

- The client application needs to connect to an SQL server
- Establishing an SQL session
- Execute a series of statements, and finally disconnects the session.

In addition to the normal SQL commands, SQL Environment contains several Components including a user identifier, and a schema. All the SQL statements like DDL and DML statements operate in the

context of a schema.

The SQL language has been standardized by the **ANSI X3H2 Database Standards Committee**. Two of the latest standards are **SQL-89** and **SQL-92**. Over the years, each vendor of relational databases has introduced new commands to extend their particular implementation of SQL. Oracle's implementation of the SQL language conforms to the basic **SQL-92** standard and adds some additional commands and capabilities.



1. Introduction to DDL

DDL – Data Definition Language is used to create and modify the structure or schema of database objects in a database. These database objects include views, schemas, tables, indexes, etc.

DDL helps to specify :

- The schema for each relation
- The domain of value associated with each attribute and integrity constraints
- The set of indices to be maintained for each relation
- The security and authorization information for each relation
- The physical storage structure of each relation on disk - **These statements define the implementation details of the database schema, which are usually hidden from the users.**

Some commonly used DDL commands

Commands	Description
CREATE	To create objects in the database
ALTER	Alters the structure of the database (Change an existing table, view or index definition)
DROP	Delete objects from the database

TRUNCATE	Remove all records from a table, including all spaces allocated for the records are removed

2. Introduction to DML

DML – data manipulation language enables users to access or manipulate data as organized by the appropriate data model. They are used to query and Manipulate existing objects like tables.

Command	Description
SELECT	retrieve data from the a database
INSERT	insert data into a table
UPDATE	updates existing data within a table
DELETE	deletes all records from a table, the space for the records remain

3. Introduction to DCL

DCL is the segment of SQL used for controlling access to data in a database. The owner of database objects, say tables, can allow other database users access to the object as per discretion.

- DCL allows protecting the tables and other objects created by a user from accidental manipulation by another user.
- Granting privileges (insert, select) to others, allows them to perform operations within their scope
- Privileges determine whether or not a particular user can perform a command.
- Using DCL we can control the privileges

For E.g. if a user is granted a select privilege he/she can only view the data but cannot perform any other DML operation on the owner's object. Privileges granted can also be withdrawn by the owner at any time.

Command	Description
GRANT	Providing permissions to the user
REVOKE	Cancellation of the permission which is already been given to the user

4. Introduction to TCL (Transaction control language)

Transaction Control Language(TCL) commands are used to manage transactions in the database. These are used to manage the changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions.

Command	Description
COMMIT	Saving the transaction changes to the database permanently
ROLLBACK	This command restores the database to last committed state. It is also used with savepoint command to jump to a savepoint in a transaction.
SAVEPOINT	Savepoint command is used to temporarily save a transaction so that you can rollback to that point whenever necessary.

In addition Oracle also provides following groups :

Session Control Statements : These statements let a user control the properties of the current session, including enabling and disabling roles and changing language settings. The two session control statements are **ALTER SESSION** and **SET ROLE**.

System Control Statements These statements change the properties of the Oracle server instance. The only system control statement is **ALTER SYSTEM**. It lets users change settings, such as the minimum number of shared servers, kill a session, and perform other tasks.

Embedded SQL Statements These statements incorporate DDL, DML, and transaction control statements in a procedural language program, such as those used with the Oracle pre-compilers. Examples include OPEN, CLOSE, FETCH, and EXECUTE.

Data Integrity Overview

- The term data integrity refers to the accuracy and consistency of data.
- When creating databases, attention needs to be given to data integrity and how to maintain it. A good database will enforce data integrity whenever possible.
  For example, a user could accidentally try to enter a phone number into a date field. If the system enforces data integrity, it will prevent the user from making these mistakes.

- If a DML statement attempts to violate this integrity rule, then DBMS must roll back the invalid statement and return an error to the application.
- DBMS provides various **integrity constraints** (declarative way to define a business rule for a column of a table) and **database triggers** to manage data **integrity rules**.
- If an integrity constraint is created for a table and some existing table data does not satisfy the constraint, then the constraint cannot be enforced.

Types of Data Integrity

In the database world, data integrity is often placed into the following types:

- Entity integrity : Defines each row to be unique within its table. No two rows can be the same. To achieve this, a primary key can be defined.
- Referential integrity : *Referential integrity* is concerned with relationships. When two or more tables have a relationship, we have to ensure that the foreign key value matches the primary key value at all times. We don't want to have a situation where a foreign key value has no matching primary key value in the primary table. This would result in an orphaned record.

  So referential integrity will prevent users from:

  - Adding records to a related table if there is no associated record in the primary table.
  - Changing values in a primary table that result in orphaned records in a related table.
  - Deleting records from a primary table if there are matching related records.

- Domain integrity : concerns the validity of entries for a given column. Selecting the appropriate data type for a column is the first step in maintaining domain integrity. Other steps could include, setting up appropriate constraints and rules to define the data format and/or restricting the range of possible values.

- User-defined integrity : Allows the user to apply business rules to the database that aren't covered by any of the other three data integrity types.

Note: Database triggers let you define and enforce integrity rules, but a database trigger is not the same as an integrity constraint. Among other things, a database trigger does not check data already loaded into a table.

How to define Integrity Constraints

Integrity constraints are defined with a table and are stored as part of the table's definition in the data dictionary, so that all database applications adhere to the same set of rules. When a rule changes, it only needs be changed once at the database level and not many times for each application.

The following integrity constraints are supported by Oracle:

- **NOT NULL (NN)**: Disallows nulls (empty entries) in a table's column. The Not Null Constraint is automatically defined for primary keys.
- **UNIQUE KEY (UK)**: Disallows duplicate values in a column or set of columns. Primarily used to enforce uniqueness in a candidate key. A candidate key is a column or columns that could have been used as the primary key but was not, but were uniqueness should still be enforced. **Null values may be allowed if desired.**

- **PRIMARY KEY (PK)**: Disallows duplicate values and nulls in a column or set of columns. . A table can have only one primary key constraint.
- **FOREIGN KEY (FK)**: Requires each value in a column or set of columns to match a value in a related table's UNIQUE or PRIMARY KEY. FOREIGN KEY integrity constraints also define referential integrity actions that dictate what Oracle should do with dependent data if the data it references is altered. For example, Oracle will not allow a value in the customer ID column of a customer payments table unless that customer ID exists in the customer table. Null values may be allowed if desired.

- **CHECK**: Disallows values that do not satisfy the logical expression of the constraint. A check constraint allows an entered value to be "checked" against a set of defined conditions. For example we may check to see that a student grade point average is between 0 and 4.0, or that the hours worked in a week by an employee is between 0 and 60, or that the answer to a question is either yes or no. Check constraints may involve more than one column. Null values may be allowed if desired.

NOTE: business rules too complex for these constraints must be enforced via **Triggers** or **Application Code**.

Keys

Key is used in the definitions of several types of integrity constraints. A key is the column or set of columns included in the definition of certain types of integrity constraints. Keys describe the relationships between the different tables and columns of a relational database. Individual values in a key are called key values. The different types of keys include:

- Primary key: The column or set of columns included in the definition of a table's PRIMARY KEY constraint. A primary key's values uniquely identify the rows in a table. Only one primary key can be defined for each table.
- Unique key: The column or set of columns included in the definition of a UNIQUE constraint.
- Foreign key: The column or set of columns included in the definition of a referential integrity constraint.
- Referenced key: The unique key or primary key of the same or a different table referenced by a foreign key.

SQL Data Definition Language (DDL)

During installation Oracle configure a default database for us. We can use that database or create our own database either manually or using **Database Configuration Assistance Tool** available from Start Menu.

An Oracle database can contain **one or more *schemas***. A schema is a collection of database objects that can include: tables, views, indexes and sequences. By default, each user has their own schema which has the **same name** as the Oracle **username**. For example, a single Oracle database can have separate schemas for HOLOWCZAK, JONES, SMITH and GREEN.
Any object in the database must be created in only one schema. The object name is prefixed by the schema name as in: **schema.object_name**

By default, all objects are created in the user's own schema. For example, when JONES creates a database object such as a table, it is created in her own schema. If JONES creates an EMPLOYEE table, the full name of the table becomes: JONES.EMPLOYEE. Thus database objects with the same name can be created in more than one schema. This feature allows each user to have own EMPLOYEE table.

Database objects can be shared among several users by specifying the **schema name**. In order to work with a database object from another schema, a user must be granted authorization.

Create, Modify and Drop Tables, Views and Sequences

- **CREATE TABLE**

Prerequisites

- For a user to be able to create a table, he needs the create table system privilege, otherwise he'll receive the ORA-01031: insufficient privileges error message.
- Additionally, the user needs to have enough quota on the tablespace where he wants to create the table.

Oracle supports four basic data types called CHAR, NUMBER, DATE and RAW. There are also a few additional variations on the RAW and CHAR data types. The basic data types, uses and syntax, are as follows:

1. **VARCHAR2** - Character data type. Can contain letters, numbers and punctuation. The syntax for this data type is: **VARCHAR2(size)** where *size* is the maximum number of alphanumeric characters the column can hold. For example VARCHAR2(25) can hold up to 25 alphanumeric characters. In Oracle8, the maximum size of a VARCHAR2 column is 4,000 bytes.

The VARCHAR data type is a synonym for VARCHAR2. It is recommended to use VARCHAR2 instead of VARCHAR.

2. **NUMBER** - Numeric data type. Can contain integer or floating point numbers only. The syntax for this data type is: **NUMBER(precision, scale)** where *precision* is the total size of the number including decimal point and *scale* is the number of places to the right of the decimal. For example, NUMBER(6,2) can hold a number between -999.99 and 999.99.

3. **DATE** - Date and Time data type. Can contain a date and time portion in the format: DD-MON-YY HH:MI:SS. If no time component is supplied when the date is inserted, the time of 00:00:00 is used as a default. The output format of the date and time can be modified to conform to local standards.
4. **RAW** - Free form binary data. Can contain binary data up to 255 characters. Data type **LONG RAW** can contain up to 2 gigabytes of binary data. RAW and LONG RAW data cannot be indexed and cannot be displayed or queried in SQL*Plus. Only one RAW column is allowed per table.

5. **LOB** - Large Object data types. These include **BLOB** (Binary Large OBject) and **CLOB** (Character Large OBject). More than one LOB column can appear in a table. These data types are the preferred method for storing large objects such as text documents (CLOB), images, or video (BLOB).
6. **ROWID** – Hexadecimal representing a unique address of a row in its table.
7. **BFILE** – Binary data stored in an external file, up to 4 GB.

- A column may be specified as NULL or NOT NULL. This check is made just before a new row is inserted into the table. By default, a column is created as NULL if no option is given.
- Tables can also be created with constraints that enforce referential integrity (relationships among data between tables). Constraints can be added to one or more columns, or to the entire table.
- Each table may have one PRIMARY KEY that consists of a single column containing no NULL values and no repeated values.
- Up to **255** columns may be specified per table.
- Column names and table names must start with a letter and can contain only **A-Z, a-z, 0-9, #, _ and $. Name must be 1 to 30 character long (Oracle restriction) .** Column names and table names are **case insensitive**.

Table Type: The type of table to be created

- Normal: A regular database table. *It can be partitione .*
- External: An external table. **External tables** allow **Oracle** to query data that is stored outside the database in flat files. The ORACLE_LOADER driver can be used to access any data stored in any format that can be loaded by SQL*Loader. They should not be used for frequently queried **tables**.

- Index Organized: An index-organized table. Table data is also maintained in index order.

- Temporary Table: It is not stored permanently in the database. The temporary table definition persists in the same way as the definition of a regular table, but the table segment and any data in the temporary table persist only for the duration of either the transaction (Transaction option) or the session (Session option).

A Normal table is created like this:

create table [schema.]table (column datatype [Default expression] [, …..]

schema → Name of owner's schema (if omitted login user schema is used)
table → Name of the table
Default expression → default value of the column, if a value is omitted during Insert this value is Inserted in this column
column → Name of the column datatype → Data type of the column

Table belonging to other users are not in user's schema. If a table does not belong to the user, the owner's name must be prefixed to the table. For example, if there is a schema USER_B, and USER_B has an Employees table, then specify the following to retrieve data from that table SELECT * FROM USER_B.Employees

Example :

```
SQL>  CREATE TABLE employee (

  2   fname       VARCHAR2(8),
  3   minit       VARCHAR2(2),
  4   lname       VARCHAR2(8),
  5   ssn         VARCHAR2(9) NOT NULL,
  6   bdate       DATE,
  7   address     VARCHAR2(27),
  8   sex         VARCHAR2(1),
  9   salary        NUMBER(7)  DEFAULT (0)
                                  NOT NULL,
 10   superssn    VARCHAR2(9),
 11   dno         NUMBER(1) NOT NULL ) ;

Table created.

SQL>
```

- The numbers 2 through 11 before each line indicate the line number supplied by the SQL*Plus program as this statement was typed in. We will omit these numbers in the rest of the examples.

- Default: For relevant types, the default value inserted into the column if no value is specified when a row is inserted.

A new table can also be created with a subset of the columns in an existing table. In the following example, a new table called emp_department_1 is created with only the *fname,*

minit, lname and *bdate* columns from the employee table. This new table is also populated with data from the employee table where the employees are from department number 1.

```
SQL> CREATE TABLE emp_department_1
            AS SELECT fname, minit, lname,
              bdate
               FROM employee
               WHERE dno = 1 ;
Table created.
```

```
SQL> DESCRIBE emp_department_1
 Name                          Null?    Type
 ----------------------------- -------- ----
 FNAME                                  VARCHAR2(8)
 MINIT                                  VARCHAR2(2)
 LNAME                                  VARCHAR2(8)
 BDATE                                  DATE
```

- **DESCRIBE** - SQL*Plus command that displays the columns of a table and their data types.
- The **copying of data can be suppressed** by giving a WHERE clause that always evaluates to FALSE for each record in the source table. The following example makes a duplicate of the employee table but does not copy any data into it.

```
SQL>  CREATE TABLE copy_of_employee
    AS SELECT *
    FROM employee
    WHERE 3=5 ;

 Table created.
```

```
SQL> DESCRIBE copy_of_employee
 Name                          Null?    Type
 ----------------------------- -------- ----
 FNAME                                  VARCHAR2(8)
 MINIT                                  VARCHAR2(2)
 LNAME                                  VARCHAR2(8)
 SSN                           NOT NULL VARCHAR2(9)
 BDATE                                  DATE
 ADDRESS                                VARCHAR2(27)
 SEX                                    VARCHAR2(1)
 SALARY                        NOT NULL NUMBER(7)
 SUPERSSN                               VARCHAR2(9)
 DNO                           NOT NULL NUMBER(1)
```

After creating various objects in the database we can check successful creation of those objects by querying following data dictionary tables.

See the names of the table owned by the user → SELECT table_name FROM **user_tables**;

Selct distinct object types owned by the user → SELECT DISTINCT object_type
FROM **user_objects**

View tables, views, synonames and sequence owned by the user
SELECT * FROM **user_catalog** **cat** is a synonym of catalog so we can also write
SELECT * FROM cat

Constraints can be added to the table at the time it is **created**, or at **a later time using the ALTER TABLE statement.**

Constraints

Some simple rules:

- Constraints can be entered as part of the table definition or can be added after table is created.
- DBMS will not allow a constraint to be added to a table that already violates the constraint.
- Constraints can generally be defined at the column level (i.e. be part of a column definition) or at the table level.
- Multi-column constraints must be defined at the table level.
- The Not Null constraint must be defined at the column level.
- Users can find constraint metadata in the **user_contraints** view
- No two constraints in a schema can have the **same name**
- If you do not name a constraint DBMS will assign a nondescript name which is not helpful when an exception is raised in response to a constraint violation. Always name your Constraints.
- Constraints more complex than those described here will have to be enforced through **triggers or application code**

Default option

A column can be given a default value by using the DEFAULT option. This option prevents null values from entering the columns if a row is inserted without a value for the column. The default value can be a :

- literal value or expression
- A SQL function such as SYSDATE and USER
- Value cannot be the name of another **column** or **psedu-column**, such as NEXTVAL or CURRVAL
- The default expression must match the data type of the column

Primary key and Unique key constraints

```
CREATE TABLE department
(dnumber      NUMBER(1),
 dname        VARCHAR2(15),
 mgrssn       VARCHAR2(9),
 mgrstartdate DATE
 CONSTRAINT pk_department PRIMARY KEY (dnumber) Enable );
```

- An **index** is automatically created on the primary key.
- Name of the constraint to be associated

Referential integrity constraints

```
CREATE TABLE employee
(fname        VARCHAR2(8),
 minit        VARCHAR2(2),
 ……..
 ssn          VARCHAR2(9) NOT NULL,
……..
dno           NUMBER(1) NOT NULL,
CONSTRAINT pk_emp PRIMARY KEY (ssn),
CONSTRAINT fk_dno FOREIGN KEY (dno)
REFERENCES department (dnumber) ON
DELETE CASCADE);
```

with the primary key definition. Must be **unique within the database**.

- Enable means the primary key constraint is enforced: that is, the data in the primary key column (or set of columns) must be unique and not null. The DISABLE clause causes Oracle to define the constraint but not enable it.
- Primary key also can be defined inline as shown below:

```
CREATE TABLE department
(dnumber      NUMBER(1) CONSTRAINT
              pk_department PRIMARY KEY,
 dname        VARCHAR2(15),
 mgrssn       VARCHAR2(9),
 mgrstartdate DATE );
```

- The following statement defines a composite primary key on the combination of the dnumber and dname columns :

```
ALTER TABLE department
  ADD CONSTRAINT dept_pk PRIMARY KEY
(dnumber, dname ) DISABLE;
```

The dno column in the employee table references the dnumber column in the department table. If a department is deleted, all employees that reference the department are also deleted. This is given by the **ON DELETE CASCADE option.** Other options include ON DELETE SET DEFAULT and ON DELETE SET NULL. For **ON UPDATE CASCADE** , if we update a dnumber in a row of table department the DBMS will update it accordingly on all Employee rows referencing this department (but no triggers activated on employee table).

In order to specify a foreign key constraint, the column in the child (or detail) table (e.g., the dnumber column in the department table in the above example) must be either the primary key or a unique key for the table. Thus, the child (or detail) table must be created first before the parent (or master) table is created using the above constraints.

The constraint **fk_dno** ensures that all departments given for employees are present in the **department** table. If we do not give NOT NULL , then NULL is allowed for dno (meaning they are not assigned to any department). To ensure that all employees are assigned to a department, you could create a NOT NULL constraint on the **dno** column , in addition to the REFERENCES constraint.

The foreign key constraint can also be defined inline as shown below, in this case no FOREIGN KEY clause and datatype of the dno column are not needed, because Oracle automatically assigns to this column the datatype of the referenced key.

```
CREATE TABLE employee
(fname       VARCHAR2(8),
 minit       VARCHAR2(2),
 ……..
 ssn         VARCHAR2(9) NOT NULL,
```

…….
dno CONSTRAINT fk_dno REFERENCES department(**dnumber**) ON DELETE CASCADE),
CONSTRAINT pk_emp PRIMARY KEY (ssn)) ;

Unique Constraint - prohibits multiple rows from having the same value in the same column or combination of columns but allows some values to be null.

When you define a unique constraint **inline**, you need only the UNIQUE keyword. When you define a unique constraint **out of line**, you must also specify one or more columns. You must define a composite unique key out of line.

CREATE TABLE student (

2 fname VARCHAR2(8),
3 lname VARCHAR2(8),
5 rollno VARCHAR2(9) NOT NULL,
6 regno VARCHAR2(9) NOT NULL,
7 address VARCHAR2(27),
8 sex VARCHAR2(1) ,
CONSTRAINT pk_rollno PRIMARY KEY (rollno),
CONSTRAINT "student_UK1" UNIQUE ("fname", "regno")) ;

Inline Syntax :

create table ri_unique (
a number unique,
b number)

- To satisfy a composite unique key, no two rows in the table or view can have the same combination of values in the key columns. Any row that contains nulls in all key columns automatically satisfies the constraint. However, two rows that contain nulls for one or more key columns and the same combination of values for the other key columns violate the constraint.
- When you specify a unique constraint on one or more columns, Oracle implicitly creates an index on the unique key.
- A table or view can have **only one unique key**.
- None of the columns in the unique key can have datatype LOB, LONG, LONG RAW, VARRAY, NESTED TABLE, OBJECT, BFILE, or REF, or TIMESTAMP WITH TIME ZONE. However, the unique key can contain a column of TIMESTAMP WITH LOCAL TIME ZONE.
- A composite unique key cannot have more than 32 columns.
- You cannot designate the same column or combination of columns as both a primary key and a unique key.

CHECK constraints

Use CHECK constraints when you need to enforce integrity rules based on logical expressions, such as comparisons. Never use CHECK constraints when any of the other

- CHECK constraints can be added to check the values for a given column.
- The CHECK constraints are activated when

types of integrity constraints can provide the necessary checking.

Examples of CHECK constraints include the following:

- A CHECK constraint on employee salaries so that no salary value is greater than 10000.
- A CHECK constraint on department locations so that only the locations "BOSTON", "NEW YORK", and "DALLAS" are allowed.
- A CHECK constraint on the salary and commissions columns to prevent the commission from being larger than the salary.

```
CREATE TABLE employee
(fname   VARCHAR2(8),
 …..
 ssn       VARCHAR2(9) NOT NULL,
 …….
 sex       VARCHAR2(1)
CONSTRAINT ck_sex CHECK (sex IN ('M', 'F')),
 salary    NUMBER(7) NOT NULL
    CONSTRAINT ck_salary CHECK (salary >
    10000),
 superssn      VARCHAR2(9),
 dno           NUMBER(1) NOT NULL,
 CONSTRAINT pk_emp PRIMARY KEY (ssn),
 CONSTRAINT fk_dno FOREIGN KEY (dno)
   REFERENCES department (dnumber) ON
DELETE CASCADE);
```

inserting a new row or when updating existing data

- The condition in CHECK constraint must be a boolean expression that can be evaluated using the values in the row being inserted or updated.
- The condition cannot contain **subqueries** or **sequences**.
- The condition cannot include the **SYSDATE**, **UID**, **USER**, or **USERENV** SQL functions.
- The condition cannot contain the pseudo columns LEVEL, PRIOR, or ROWNUM.
- The condition cannot contain a **user-defined SQL function**.

```
create table tbl_ril  (
  a number check (a between 0 and 100),
  b number
);
```

In the previous examples, constraints were given names with the following prefixes:

Primary key constraints: pk_ , Foreign key constraints: fk_ , Check constraints: ck_
Naming constraints in this fashion is simply a convenience. Any name may be given to a constraint.

- **ALTER TABLE** - Change an existing table definition. This statement can be used to
  - Add a new column or drop an existing column in a table
  - Modify the data type for an existing column
  - Add or remove a constraint
  - Define a default value for the new column

Syntax :

ALTER TABLE tableName **MODIFY** (column datatype [DEFAULT expr] [, column datatype].....); You can change dtatype, size and default value	ALTER TABLE tableName **DROP** (column);
In ALTER TABLE command **ADD | MODIFY | DROP** → indicates the type of modification More syntax of ALTER is discussed below	
Adding a new column to an existing table: Example : ALTER TABLE my_birthdays ADD (age NUMBER(3)); **Add Default Value constraint in a column** ALTER TABLE tbl_name modify bp_type default('0');	**Rename the my_employees table or a column** ALTER TABLE my_employees RENAME to temp_employees; ALTER TABLE my_employees RENAME COLUMN old_ColName new_ColName;
Change Datatype of column. Syntax : ALTER TABLE MODIFY (<column_name> <new data type> <null |not null>) ;	**Add a constraint to a table - PRIMARY KEY** Syntax : ALTER TABLE ADD CONSTRAINT <constraint name> PRIMARY KEY (<column name>); ALTER TABLE Persons ADD CONSTRAINT pk_PersonID PRIMARY KEY (P_Id,LastName) Primary key on a particular column : ALTER TABLE Persons ADD PRIMARY KEY (P_Id)
Add a constraint to a table - FOREIGN KEY Syntax : ALTER TABLE ADD CONSTRAINT <constraint name> FOREIGN KEY (<column name>) REFERENCES <parent_table_name> (column-name); Example : ALTER TABLE Orders ADD FOREIGN KEY (P_Id) REFERENCES Persons(P_Id) ALTER TABLE Orders ADD CONSTRAINT fk_PerOrders FOREIGN KEY (P_Id)	**Dropping the constraints from a table** ALTER TABLE my_employees DROP UNIQUE (email); ALTER TABLE Dept_tab DROP UNIQUE (Loc); ALTER TABLE Emp_tab DROP PRIMARY KEY, DROP CONSTRAINT Dept_fkey; DROP TABLE Emp_tab CASCADE CONSTRAINTS;

REFERENCES Persons(P_Id)	Specify **CASCADE CONSTRAINTS** to drop all referential integrity constraints that refer to primary and unique keys in the dropped table. If you omit this clause, and such referential integrity constraints exist, then the database returns an error and does not drop the table.
Add a unique constraint on an existing table: ADD CONSTRAINT supplier_unique UNIQUE (supplier_id); **Note : After table creation, constrained defined at column level (inline) not allowed except for NOT NULL constraints**	**Add Null Constraint after table creation** The not null/null constraint can be altered with **alter table ri_not_null modify col1 null;** After this modification, the column col1 can contain null values
After table creation, defined at table level ALTER TABLE STUDENT ADD (CONSTRAINT Student_SID_PK PRIMARY KEY(SID), CONSTRAINT Student_SSNO_UK UNIQUE(SSNO));	As with PK and FK constraints, a multi-column UK would have to be defined at the table level. For example: ALTER TABLE STUDENT ADD (CONSTRAINT Student_SID_PK PRIMARY KEY(SID), CONSTRAINT Student_lname_fname_UK UNIQUE(lastname,firstname));
After table creation, not null constraint defined at column level ALTER TABLE STUDENT MODIFY (LASTNAME CONSTRAINT Student_lname_nn NOT NULL, FIRSTNAME CONSTRAINT Student_fname_nn NOT NULL);	**After table creation, defined at table level** ALTER TABLE STUDENT ADD CONSTRAINT Student_Status_CC CHECK(STATUS='PT' or STATUS='FT' or STATUS='NE'); ALTER TABLE PROJECTS ADD (CONSTRAINT Projects_dates_CC CHECK(project_start_date<project_end_date));
Dropping a Table **DROP** TABLE my_birthdays; (All pending transaction committed, table and associated index dropped)	**Dropping a Column** ALTER TABLE dept DROP COLUMN job_id;
Disabling Constraints create table foo (bar number, baz number, unique (bar, baz));	**Enabling Existing Constraints** Once you have finished cleansing data and filling in empty columns, you can enable

alter table foo disable unique (bar, baz); **Disabling Named Constraints** If you need to perform a large load or update when the table already contains data, you can temporarily disable constraints to improve performance of the bulk operation. The following statements are examples of statements that disable enabled integrity constraints: ALTER TABLE Dept_tab DISABLE CONSTRAINT Dname_ukey;	constraints that were disabled during data loading. The following statements are examples of statements that enable disabled integrity constraints: ALTER TABLE Dept_tab ENABLE CONSTRAINT Dname_ukey; ALTER TABLE Dept_tab ENABLE PRIMARY KEY ENABLE UNIQUE (Dname) ENABLE UNIQUE (Loc); ALTER TABLE Dept_tab DISABLE PRIMARY KEY DISABLE UNIQUE (Dname) DISABLE UNIQUE (Loc);

Oracle stores the definitions of integrity constraints in the data dictionary.

- **Listing all of the constraints for the table along with any referenced constraint on other tables:**

1. Metadata API (DBMS_METADATA)

From Oracle 9i the DBMS_METADATA package was introduced to retrieve object definitions as XML or SQL DDL. The signature of functions are as follows :

```
FUNCTION Get_XML (  object_type IN VARCHAR2,   name IN VARCHAR2,
        schema IN VARCHAR2 DEFAULT NULL,        version IN VARCHAR2 DEFAULT 'COMPATIBLE',
        model IN VARCHAR2 DEFAULT 'ORACLE',     transform IN VARCHAR2 DEFAULT NULL)
        RETURN CLOB;

FUNCTION Get_DDL (  object_type IN VARCHAR2,   name IN VARCHAR2,
        schema IN VARCHAR2 DEFAULT NULL,        version IN VARCHAR2 DEFAULT 'COMPATIBLE',
        model IN VARCHAR2 DEFAULT 'ORACLE',     transform IN VARCHAR2 DEFAULT NULL)
        RETURN CLOB;
```

Example : Shown the ddl statement for creating table TBL_ACTIONCODE XML or SQL DDL:

```
SQL> set long 20000;
SQL> set pagesize 0;
SQL> select DBMS_METADATA.get_ddl ('TABLE', 'TBL_ACTIONCODE', 'ERWS_GLOBAL') from dual;
```

Note : All parameters in get_ddl must in Capital, as Data Dictionary stores everything in caps.

```
CREATE TABLE "ERWS_GLOBAL"."TBL_ACTIONCODE"
 (    "IDN" NUMBER(10,0),
      "ENAME" NVARCHAR2(50),
      "DESCRIPTION" NVARCHAR2(512),
      "IS_DELETED" NUMBER(1,0) DEFAULT (0),
      "CREATED_BY" NVARCHAR2(20),
      "CREATED_DATE" DATE,
      "UPDATED_BY" NVARCHAR2(20),
      "UPDATED_DATE" DATE,
      "REC_VERSION" NUMBER(10,0) DEFAULT (0)
 ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
TABLESPACE "USERS"
```

Renaming Columns And Constraints

In addition to renaming tables and indexes Oracle9i Release 2 allows the renaming of columns and constraints on tables. In this example once the TEST1 table is created it is renamed along with it's columns, primary key constraint and the index that supports the primary key:

```
SQL> CREATE TABLE test1 (
  2    col1  NUMBER(10) NOT NULL,
  3    col2  VARCHAR2(50) NOT NULL);


Table created.
```

```
SQL> ALTER TABLE test1 ADD (
  2    CONSTRAINT test1_pk PRIMARY KEY
(col1));

SQL> DESC test1
 Name                      Null?    Type
 ------------------------- -------- -------------------
 COL1                      NOT NULL NUMBER(10)
 COL2                      NOT NULL VARCHAR2(50)
```

```
SQL> SELECT constraint_name
  2  FROM   user_constraints
  3  WHERE  table_name      = 'TEST1'
  4  AND    constraint_type = 'P';

CONSTRAINT_NAME
------------------------------
TEST1_PK
```

```
SQL> SELECT index_name, column_name
  2  FROM   user_ind_columns
  3  WHERE  table_name = 'TEST1';

INDEX_NAME          COLUMN_NAME
--------------------  --------------------
TEST1_PK            COL1
```

SQL> ALTER TABLE test1 RENAME TO test; SQL> ALTER TABLE test RENAME CONSTRAINT test1_pk TO test_pk;	SQL> ALTER TABLE test RENAME COLUMN col1 TO id; SQL> select * from user_cons_columns;

Creating and Dropping a Sequence

A **sequence** is a database object from which multiple users may generate unique integers. You can use sequences to automatically generate primary key values.

When a sequence number is generated, the sequence is incremented, independent of the transaction committing or rolling back. If two users concurrently increment the same sequence, then the sequence numbers each user acquires may have gaps, because sequence numbers are being generated by the other user. One user can never acquire the sequence number generated by another user. Once a sequence value is generated by one user, that user can continue to access that value regardless of whether the sequence is incremented by another user.

Sequence numbers are generated independently of tables, so the same sequence can be used for one or for multiple tables.

Once a sequence is created, you can access its values in SQL statements with the CURRVAL pseudocolumn, which returns the current value of the sequence, or the NEXTVAL pseudocolumn, which increments the sequence and returns the new value.

Note :

DUAL is a table automatically created by Oracle Database along with the data dictionary. **DUAL** is in the schema of the user **SYS** but is accessible by the name **DUAL** to all users. It has one column, **DUMMY**, defined to be **VARCHAR2(1)**, and contains one row with a value **X**. Selecting from the **DUAL** table is useful for computing a constant expression with the **SELECT** statement. Because **DUAL** has only one row, the constant is returned only once.

CREATE SEQUENCE has the following syntax: CREATE SEQUENCE INCREMENT BY START WITH MAXVALUE CYCLE NOORDER CACH ; **Cycle/Nocycle**: Indicates whether the sequence "wraps around" to reuse numbers	Created sequence object can be used in INSERT statements to generate unique primary key values in a table. Example : CREATE SEQUENCE patron_id_seq START WITH 100 INCREMENT BY 1;

<table>
<tr><td>after reaching its maximum value (for an ascending sequence) or its minimum value (for a descending sequence). If cycling of values is not enabled, the sequence cannot generate more values after reaching its maximum or minimum value.

Cache and Cache size: If Cache is checked, sequence values are preallocated in cache, which can improve application performance; Cache size indicates the number of sequence values preallocated in cache. If Cache is not checked, sequence values are not preallocated in cache.

Order/NoOrder: Indicates whether sequence numbers are generated in the order in which they are requested. If no ordering is specified, sequence numbers are not guaranteed to be in the order in which they were requested.</td><td>CREATE SEQUENCE new_employees_seq
START WITH 1000 INCREMENT BY 1;

CREATE SEQUENCE
"ERWS_GLOBAL"."TBL_ACTIONCODE_ID_SEQ"
MINVALUE 1 MAXVALUE 9999999999
INCREMENT BY 1 START WITH 13 CACHE 20
NOORDER NOCYCLE ;

NoOrder : If multiple request for inset at the same time, there is no guarantee that sequence will be in the order of request</td></tr>
<tr><td colspan="2">To use the sequence

• first initialize the sequence with NEXTVAL
SELECT new_employees_seq.NEXTVAL FROM DUAL;

• after initializing the sequence, use CURRVAL as the next value in the sequence during Insert

INSERT INTO employees VALUES
(new_employees_seq.CURRVAL, 'Pilar', 'Valdivia', 'pilar.valdivia',
'555.111.3333', '01-SEP-05', 'AC_MGR', 9100, .1, 101, 110);</td></tr>
<tr><td colspan="2">We can also use trigger to generate auto-increment numbers using sequence during insert.

CREATE OR REPLACE TRIGGER "ERWS_GLOBAL"."TBL_PLANT_ID_TRG" before
INSERT ON "TBL_PLANT" FOR EACH row
BEGIN
IF inserting THEN IF :NEW."IDN" IS NULL THEN
SELECT TBL_PLANT_ID_SEQ.nextval INTO :NEW."IDN" FROM dual;</td></tr>
<tr><td>DROP SEQUENCE</td><td>Sequences can also be altered to change the INCREMENT BY, MAXVALUE or START WITH</td></tr>
</table>

DROP SEQUENCE new_employees_seq;	values. The **ALTER SEQUENCE** statement achieves these changes.

Creating and Dropping a Synonym

A synonym is an **alias** for any schema object such as a table or view. Synonyms provide an easy way to provide an alternative name for a database object and can be used to simplify SQL statements for database users. For example, you can create a synonym named **emps** as an alias for the **employees** table in the HR schema.

If a table in an application has changed, such as the personnel table has replaced the employees table, you can use the employees synonym to refer to the personnel table so that the change is transparent to the application code and the database users.

You can create both **public** and **private** synonyms. A public synonym is owned by the special user group named **PUBLIC** and every user in a database can access it. A private synonym is in the schema of a **specific user** who has control over its availability to others.

Public: If this option is checked, the synonym is accessible to all users. (However each user must have appropriate privileges on the underlying object in order to use the synonym.) If this option is not checked, the synonym is a **private** synonym, and is accessible only within its schema.

Schema: Database schema in which to create the synonym.

Name: Name of the synonym. A private synonym must be unique within its schema; a public synonym must be unique within the database.

Create a synonym for the employees table : **CREATE SYNONYM emps for HR.employees;**

CREATE PUBLIC SYNONYM employees FOR hr.employees@sales;

Query the employees table using the emps synonym
SELECT employee_id, last_name FROM emps WHERE employee_id < 105;

Dropping a Synonym : **DROP SYNONYM emps;**

Views

- A view is a virtual table that does not physically exist.
- A view is created by a query joining one or more tables i.e. a logical representation of another table or combination of tables. These tables are called **base tables**.
- Base tables might in turn be **actual tables** or might be **views** themselves.
- All operations performed on a view actually affect the base table of the view.
- You can use views in almost the same way as tables. You can query, update, insert into, and delete from views, just as you can standard tables.

When a view is defined, a SQL statement (SELECT) is associated with the view name. Whenever the view is accessed, the SQL statement will be executed

Advantage :

- To hide the complexity of relationships between tables or to provide security for sensitive data in tables
- To make complex queries easy
- Need not write complex query repeatedly
- To present different views of the same data

There are two classifications : Simple and Complex View

Simple view :

- Derives data from single table
- Contains no functions or group of data
- Can perform DML operations through the view

Complex view :

- Data from many table
- Contains functions/ group
- Not always allow DML operations through the view

Syntax :

Create irrespective of existence of base tables
Create if base tables exist

```
CREATE  [OR REPLACE]  [FORCE | NOFORCE]
        VIEW  viewName   [ (alias[ , alias] ….)]
AS subquery
[ WITH CHECK OPTION  [ CONSTRAINT  constraint ]]
[ WITH READ ONLY [ CONSTRAINT  constraint ]]
```

Alias can be used in column level of SELECT subquery or in the CREATE statement. No of aliases listed in the CREATE statement must match the number of expressions selected in the subquery.

If **alias** is omitted the database derives them from the columns or column aliases in the query.

- **subquery -** complete **SELECT** statement (col. alias can be used in the select list)

- **WITH READ ONLY** – prevents any updates, inserts, or deletes from being done to the base table through the view. If no WITH clause is specified, the view, with some restrictions, is inherently updatable. DML operations can be performed on this view

- **WITH CHECK OPTION -** indicate that Oracle Database prohibits any changes to the table or view that would produce rows that are not included in the subquery. It is basically checking the where clause violation during inserting/updating row using view.

- **Constraint** – name assigned to the CHECK OPTION constraint (if no name provided name is auto-generated by DBMS)

Constraint in view –

- Declarative primary key, unique key, foreign key constraints can be defined against view.
- View read only constraint can be defined by WITH READ ONLY clause (read only a view)
- View Check Option constraint can also be defined
- NOT NULL → inherited directly from base table

```
CREATE  VIEW  Emp_View
( id  number  PRIMARY KEY  DISABLE NONVALIDATE,
  Firstname  varchar(20)
  AS   SELECT  id,  firstname
       FROM  employees
       WHERE department_id = 0
```

```
ALTER VIEW  Emp_View
ADD CONSTRAINT  emp_vw_unq
UNIQUE(first_name)   DISABLE
        NONVALIDATE
```

- ENABLE ensures that all incoming data conforms to the constraint
- DISABLE allows incoming data, regardless of whether it conforms to the constraint
- VALIDATE ensures that existing data conforms to the constraint
- NOVALIDATE means that some existing data may not conform to the constraint

- ENABLE VALIDATE is the same as ENABLE. The constraint is checked and is guaranteed to hold for all rows.
- ENABLE NOVALIDATE means that the constraint is checked, but it does not have to be true for all rows (existing rows may violate the constraint but all new or modified rows must obey the constraint).

```
CREATE VIEW  itm_view ( ITEMCODE,
DESCRIPTION,  ITEM_VALUE)
AS SELECT  itCD,  itName, Rate*QTY
FROM ItemMaster
WHERE ItemCat = 'EE'
```

```
CREATE VIEW  itm_view
AS SELECT  itCD AS ITEMCODE,  itName
DESCRIPTION,  Rate*QTY AS ITEM_VALUE
FROM ItemMaster
WHERE ItemCat = 'EE'
```

** Alias can be used in column level of SELECT or in the CREATE statement. No of aliases listed in the CREATE statement must match the number of expressions selected in the subquery

Guidelines for creating a view :

- Subquery can contain complex SELECT syntax including joins, groups other subqueries
- Subquery cannot contain an ORDER BY clause. Order by clause is specified when you retrieve data from the view.
- If constraint name not specified, system assigns a default name in the format SYS_Cn

Once the view is created, it can be queried with a SELECT statement as if it were a table.
SELECT * FROM **itm_view**;

Some important conditions for working with view :

DELETE using View is permissible if subquery do not contain:

- Group function
- Group by clause
- **Distinct** or pseudo column **ROWNUM** keyword

Modify using view is permissible if subquery do not contain:

- Group function or Group by clause
- **Distinct** or pseudo column **ROWNUM** keyword
- Columns defined by the expressions

INSERT using View is permissible if subquery do not contain:

- Group function or Group by clause
- **Distinct** or pseudo column **ROWNUM** keyword
- Columns defined by the expressions
- Not NULL columns in the base tables without any default value clause that are not selected by the view

Example : The following statement creates a view on a subset of data in the emp table:

```
CREATE VIEW sales_staff AS
   SELECT empno, ename, deptno FROM emp
   WHERE deptno = 10
   WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
```

The CHECK OPTION creates the view with the constraint (named sales_staff_cnst) that INSERT and UPDATE statements issued against the view cannot result in rows that the view cannot select.

For example, the following INSERT statement successfully inserts a row into the emp table by means of the sales_staff view, which contains all rows with department number 10:

```
INSERT INTO sales_staff VALUES (7584, 'OSTER', 10);
```

However, the following INSERT statement returns an error because it attempts to insert a row for department number 30, which cannot be selected using the sales_staff view:

```
INSERT INTO sales_staff VALUES (7591, 'WILLIAMS', 30);
```

It will fail with an error : **view WITH CHECK OPTION where-clause violation**.

Complex view joining more than one tables

- Normally an update acts on a single table. Oracle allows you, with some restrictions in addition to above conditions stated earlier, to modify views that **involve joins**.

- All updatable columns of the join view must map to columns of **key-preserved table**

So first try to understand what is key-preserved table?

Key-Preserver Table - A table is key preserved if every key of the table can also be a key of the result of the join. Key preserved table guarantees to return only one copy of each row from the base table. In simple terms, a table is key preserved if the table key participates in the view as a key.

CREATE VIEW **emp_dept** AS
SELECT a.empno, a.ename, a.sal, a.deptno, b.deptname
FROM emp a, dept b WHERE a.deptno = b.deptno

Result of the SELECT against above view is shown below

Empno	Ename	Sal	Deptno	Deptname
1	….	….	**20**	…..
2	….	….	30	…..
3	….	….	**20**	…..
4	….	….	10	…..

The table emp is a key-preserved table because in the view rows from emp appears only once. But table dept is not key-preserved table because deptno is not the key col. of the view

In the example of emp_dept view, you can say that the column emp_no which is a primary key of the table emp is also a primary key of the view emp_dept becuase it helps you fetch unique and single record for each employee, so emp table is KEY PRESERVED.

Now you take dept table, if you use dept_no column, it will give you multiple records because a single dept can have many employees, so dept table in NOT KEY PRESERVED and you cannot Insert a data through View in to a table which is NOT KEY PRESERVED.

Normally an update acts on a single table. Oracle allows you to update a view (or subquery) as long as it is still able to easily map the changes you are making onto real underlying rows in a table. This is possible if the set clause only modifies columns in a 'key preserved' table.

- For an UPDATE statement, all columns updated must be extracted from a key-preserved table. If the view has the CHECK OPTION, **join columns** and columns taken from tables that are referenced more than once in the view must be shielded from UPDATE.
- For an INSERT statement, all columns into which values are inserted must come from a key-preserved table, and the view must not have the CHECK_OPTION.

Modifying a Join View

```
CREATE VIEW Emp_dept_view AS
 SELECT e.Empno, e.Ename, e.Deptno, e.Sal, d.Dname, d.Loc
  FROM Emp e, Dept d    /* JOIN operation */
   WHERE e.Deptno = d.Deptno
    AND d.Loc IN ('HYD', 'BOM', 'DEL');
```

There are restrictions on modifying either the EMP or the DEPT base table through this view as it is a join view.

Any UPDATE, INSERT, or DELETE statement on a join view can modify only one underlying base table.

The following example shows an UPDATE statement that successfully modifies the EMP_DEPT_VIEW view:
 	
`UPDATE Emp_dept_view`
`SET Sal = Sal * 1.10`
`WHERE Deptno = 10;`
	The following UPDATE statement would be disallowed on the EMP_DEPT_VIEW view:
	
	`UPDATE Emp_dept_view`
	`SET Loc = 'BOM'`
	`WHERE Ename = 'SAMI';`
	
	This statement fails with an ORA-01779 error ("cannot modify a column which maps to a non key-preserved table"), because it attempts to modify the underlying DEPT table, and the DEPT table is not key preserved in the EMP_DEPT view.

So, for example, if the EMP_DEPT view were defined using WITH CHECK OPTION, then the following UPDATE statement would fail:

```
UPDATE Emp_dept_view
  SET Deptno = 10
    WHERE Ename = 'SAMI';
```

The statement fails because it is trying to update a join column.

Deleting from a Join View

You can delete from a join view provided there is one and only one key-preserved table in the join. The following DELETE statement works on the EMP_DEPT view:

```
DELETE FROM Emp_dept_view
  WHERE Ename = 'SMITH';
```

This DELETE statement on the EMP_DEPT view is legal because it can be translated to a DELETE operation on the base EMP table, and because the EMP table is the only key-preserved table in the join.

In the following view, a DELETE operation cannot be performed on the view because both E1 and E2 are key-preserved tables:

```
CREATE VIEW emp_emp AS
  SELECT e1.Ename, e2.Empno, e1.Deptno
    FROM Emp e1, Emp e2
    WHERE e1.Empno = e2.Empno;
```

If a view is defined using the WITH CHECK OPTION clause and the key-preserved table is repeated, then rows cannot be deleted from such a view. For example:

```
CREATE VIEW Emp_mgr AS
   SELECT e1.Ename, e2.Ename Mname
     FROM Emp e1, Emp e2
        WHERE e1.mgr = e2.Empno
        WITH CHECK OPTION;
```

No deletion can be performed on this view because the view involves a self-join of the table that is key preserved.

```
CREATE VIEW Emp_dept_view AS
  SELECT e.Empno, e.Ename, e.Deptno, e.Sal, d.Dname, d.Loc
    FROM Emp e, Dept d    /* JOIN operation */
     WHERE e.Deptno = d.Deptno AND d.Loc IN ('HYD', 'BOM', 'DEL');
```

Inserting into a Join View

The following INSERT statement on the EMP_DEPT view succeeds, because only one key-preserved base table is being modified (EMP), and 40 is a valid DEPTNO in the DEPT table (thus satisfying the FOREIGN KEY integrity constraint on the EMP table).

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
   VALUES ('ASHU', 119, 40);
```

The following INSERT statement fails for the same reason: This UPDATE on the base EMP table would fail: the FOREIGN KEY integrity constraint on the EMP table is violated.

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
   VALUES ('ASHU', 110, 77);
```

The following INSERT statement fails with an ORA-01776 error ("cannot modify more than one base table through a view").

```
INSERT INTO Emp_dept (Ename, Empno, Deptname)
   VALUES (110, 'TANNU', 'BOMBAY');
```

An INSERT cannot, implicitly or explicitly, refer to columns of a non-key-preserved table. If the join view is defined using the WITH CHECK OPTION clause, then you cannot perform an INSERT to it.

To create a view, you must meet the following requirements:

- To create a view in your schema, you must have the CREATE VIEW privilege. To create a view in another user's schema, you must have the CREATE ANY VIEW system privilege. You can acquire these privileges explicitly or through a role.

- The owner of the view (whether it is you or another user) must have been explicitly granted privileges to access all objects referenced in the view definition. The owner cannot have obtained these privileges through roles. Also, the functionality of the view is dependent on the privileges of the view owner. For example, if the owner of the view has only the INSERT privilege for Scott's emp table, the view can only be used to insert new rows into the emp table, not to SELECT, UPDATE, or DELETE rows.

- If the owner of the view intends to grant access to the view to other users, the owner must have received the object privileges to the base objects with the GRANT OPTION or the system privileges with the ADMIN OPTION.

Views can be dropped in a similar fashion to tables. The DROP VIEW command provides this facility. In the following example, the view just created is dropped.

```
DROP VIEW item_view ;
```

Views can also be created to join several tables together. The following is an example of creating a view that joins two tables:

```
SQL> CREATE VIEW dept_managers AS
  2  SELECT dnumber, dname, mgrssn, lname, fname
  3  FROM   employee, department
  4  WHERE  employee.ssn = department.mgrssn ;

View created.

SQL> SELECT * FROM dept_managers ;

DNUMBER DNAME          MGRSSN LNAME    FNAME
------- -------------- ---------- -------- --------
      5 RESEARCH       333445555 WONG     FRANKLIN
      4 ADMINISTRATION  987654321 WALLACE  JENNIFER
      1 HEADQUARTERS    888665555 BORG     JAMES
```

A view can be created that contains an aggregate function. In the following example, a view is created that returns the average salary of all employees per department.

```
SQL> CREATE VIEW dept_average_salary AS
       2  SELECT dnumber, dname, AVG(salary) AS average_salary
       3  FROM   department, employee
       4  WHERE  employee.dno = department.dnumber
       5  GROUP BY dnumber, dname ;

 View created.
```

This views is read-only as it contains GROUP BY and aggregate functions

```
SQL> SELECT * FROM dept_average_salary ;

  DNUMBER DNAME          AVERAGE_SALARY
---------- -------------- --------------
```

To see which views are defined in a schema, submit a query to the USER_VIEWS view:

```
SQL> SELECT view_name FROM user_views ;
```

1 HEADQUARTERS 55000 4 ADMINISTRATION 31000 5 RESEARCH 33250	VIEW_NAME ---------- DEPT_AVERAGE_SALARY DEPT_MANAGERS EMP_DNO_1

Grant and Revoke Statements

The GRANT and REVOKE statements allow a user to control access to objects (Tables, Views, Sequences, Procedures, etc.) in their schema. The Grant command grants authorization for a subject (another user or group) to perform some action (SELECT, INSERT, UPDATE, DELETE, ALTER, INDEX) on an object (Table, View, stored procedure, sequence or synonym).

Subject means another user or group

The access right or privilege or actions are defined as follows:

- **SELECT** - allows a subject to select rows from the object.
- **INSERT** - allows a subject to insert rows into the object.
- **UPDATE** - allows a subject to update rows in the object.
- **DELETE** - allows a subject to delete rows from the object.
- **ALTER** - allows a subject to alter the object. For example, add a column or change a constraint.
- **INDEX** - allows a subject to create an index on the object.
- **EXECUTE** - allows a subject to execute a **stored procedure** or **trigger**.

In addition to objects such as tables, the SELECT and UPDATE actions may also be granted on individual columns in a table or view.

The Syntax for the GRANT command is:

GRANT privilege_name ON object_name
TO {user_name |PUBLIC |role_name} [WITH GRANT OPTION];

- **privilege_name -** is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT etc. as listed above (use comma separated access rights).
- **object_name** - name of an database object like TABLE, VIEW etc
- **user_name** - name of the user to whom an access right is being granted.
- **PUBLIC** is used to grant access rights to all users.
- **ROLES** are a set of privileges grouped together.
- **WITH GRANT OPTION -** allows a user to grant access rights to other users**.**

Eample:

For example, assume user ALICE wishes to allow another user BOB to view the rows in the employee table. ALICE would execute the following GRANT statement:

GRANT SELECT ON employee TO BOB ;

You should use the **WITH GRANT** option carefully because for example if you GRANT SELECT privilege on employee table to BOB WITH GRANT option, then BOB can GRANT SELECT privilege on employee table to another user, such as user2 etc. Later, if you REVOKE the SELECT privilege on employee from BOB, still user2 will have SELECT privilege on employee table.

SQL REVOKE Command:

The REVOKE command removes user access rights or privileges to the database objects.
The Syntax for the REVOKE command is:

REVOKE privilege_name ON object_name FROM {user_name |PUBLIC |role_name}

For Eample:

REVOKE SELECT ON employee FROM user1;

This command will REVOKE a SELECT privilege on employee table from user1. When you REVOKE SELECT privilege on a table from a user; the user will not be able to SELECT data from that table anymore. However, if the user has received SELECT privileges on that table from more than one users, he/she can SELECT from that table until everyone who granted the permission revokes it. You cannot REVOKE privileges if they were not initially granted by you.

Privileges and Roles:

Privileges: Privileges defines the access rights provided to a user on a database object. There are two types of privileges.

- **System privileges** - This allows the user to CREATE, ALTER, or DROP database objects.
- **Object privileges** - This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply.

Few CREATE system privileges are listed below:

System Privileges	Description
CREATE object	allows users to create the specified object in their own schema.
CREATE ANY object	allows users to create the specified object in any schema.

The above rules also apply for ALTER and DROP system privileges.

Roles: Roles are a collection of privileges or access rights. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if you define roles, you can grant or revoke privileges to users, thereby automatically granting or revoking privileges. You can either create Roles or use the system roles pre-defined by oracle.

Some of the privileges granted to the system roles are as given below:

System Role	Privileges Granted to the Role
CONNECT	CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE SEQUENCE, CREATE SESSION etc.
RESOURCE	CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER etc. The primary usage of the RESOURCE role is to restrict access to database objects.
DBA	ALL SYSTEM PRIVILEGES

Creating Roles:

The Syntax to create a role is:

CREATE ROLE role_name [IDENTIFIED BY password];

For example: To create a role called "developer" with password as "pwd",the code will be as follows
CREATE ROLE developer IDENTIFIED BY pwd;

It's easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege directly to every user. If a role is identified by a password, then, when you GRANT or REVOKE privileges to the role, you definitely have to identify it with the password.

We can GRANT or REVOKE privilege to a role as below.

For example: To grant CREATE TABLE privilege to a user by creating a testing role:

- First, create a testing Role : CREATE ROLE testing
- Second, grant a CREATE TABLE privilege to the ROLE testing. You can add more privileges to the ROLE. - GRANT CREATE TABLE TO testing;
- Third, grant the role to a user - **GRANT testing TO user1;**
- To revoke a CREATE TABLE privilege from testing ROLE, you can write:
  REVOKE CREATE TABLE FROM testing;
- **The Syntax to drop a role from the database is as below:** DROP ROLE role_name;

Oracle Pseudo-Columns

The Oracle implementation of SQL adds several pseudo columns to each table. These columns do not exist in a physical table, yet they can be used in any SQL statement for a variety of purposes.

The following table lists the major pseudo columns:

- **CURRVAL** - Returns the current value of an Oracle sequence.
- **NEXTVAL** - Returns the current value of an Oracle sequence and then increments the sequence.
- **LEVEL** - The current level in a hierarchy for a query using STARTWITH and CONNECT BY.
- **ROWID** - An identifier (data file, block and row) for the physical storage of a row in a table.
- **ROWNUM** - The integer indicating the order in which a row is returned from a query.