Oracle

SQL

(For Devolopers/DBAs/Others

Notes

By

ORACLE
CERTIFIED EXPERT

MR. SHOAIB

(Oracle Certified Database Administrator

▪ DBA Trainer & Consultant ▪

9+ YEARS OF REALTIME INDUSTRY EXPERIENCE

Providing training for

-ONLINE TRAINING / IN-CLASS TRAINING-

Mobile: +91 8790023206, EMAIL: shoebcertifieddba@gmail.com

Providing Online Training / Classroom Trainings

Regular Batches
Weekend Batches
Sunday Batches

These courses are mainly designed for Oracle Database Administrators, System Administrators & also will be useful for the people who want to learn any individual course.

These courses will be useful for the following category of people who wants to learn Oracle Database Administration.

* Developers who want to acquire DBA or Unix/Linux knowledge.
* Non-Oracle DBAs (e.g. SQL Server DBA) who wants to learn Oracle
* Fresh IT graduates who want to make a career in Oracle Database Administration or Unix Admin
* Management professionals who want to understand about Oracle database and/or Unix/Linux
* Provides well designed Study Materials, lab Sessions (Exercises for more Practice) for quick learning process.

For those people who want to join classes through online

* Attend the class from anywhere from the globe.
* Real time scenario oriented training as making a candidate - job ready..
* Hundred percent job assistance for the preparation to face - technical & non-technical rounds
* Discussion on interview questions
* Online training benefits your precious time and money.

Offered Courses are the following information below:

- ♦ **Oracle (10g/11g/12c) Data Base Administration [Core DBA]**
- ♦ **Microsoft SQL Server DBA**
- ♦ **Oracle Real Application Clusters Database Administration [RAC-DBA]**
- ♦ **Oracle 11g / 12c Data Guard**
- ♦ **Oracle 11g / 12c Golden Gate**
- ♦ **Unix/Linux Administration (For Oracle Database & System Administration)**
- ♦ **Oracle Structural Language (SQL)**
- ♦ **Oracle PL/SQL**
- ♦ **Shell Scripting**

WhatsApp Group — OracleDBA – Help Desk

This group is related to all the people those who are experts, experience & junior resource persons, or those who are learners
If You really want to be include in this group.. simply add-request send to the WhatsApp number: +91 8790023206 with your name, profession, and experience for more discussion, debates, job-related information, doubts, errors for more clarification

Note:

If you are not sure whether you should be doing this course, let me know and In-shaa-Allah, I will schedule discussion with you to understand your career requirements & recommendations for your bright future.

Email: shoebcertifieddba@gmail.com
Mobile: +91-8790023206 (WhatApps)

Good Luck

History of the Success Relational Databases of Edgar Codd

In 1948, Codd moved to New York to work for IBM as a programmer for the Selective Sequence Electronic Calculator, IBM's first electronic computer, an experimental machine with 12500 vacuum tubes. He then invented a novel "multiprogramming" method for the pioneering IBM 7040 STRETCH computer. This method enabled STRETCH, the forerunner to modern mainframe computers, to run several programs at the same time. In 1953, disappointed by the USA policy, Codd moved to Ottawa, Canada. A decade later he returned to the USA and received his doctorate in computer science from the University of Michigan. Two years later he moved to San Jose, California, to work at IBM's San Jose Research Laboratory.

In the 1960s and 1970s Codd worked out his theories of data arrangement, based on mathematical set theory. He wanted to store data in cross-referenced tables, allowing the information to be presented in multiple permutations. It was a revolutionary approach. In 1969 he published an internal IBM paper, describing his ideas for replacing the hierarchical or navigational structure with simple tables containing rows and columns, but without great success and interest. Codd firmly believed that computer users should be able to work at a more natural-language level and not be concerned about the details of where or how the data was stored.

Codd's concept of data arrangement was seen within IBM as an "intellectual curiosity" at best and, at worst, as undermining IBM's existing products. Codd's ideas however were picked up by local entrepreneurs and resulted in the formation of firms such as Oracle (today the number two independent software firm after Microsoft), Ingres, Informix and Sybase.

Let's see how Don Chamberlin, an IBM colleague of Codd and coinventor of SQL, was acquainted with Codd's ideas: "...since I'd been studying CODASYL (the language used to query navigational databases), I could imagine how those queries would have been represented in CODASYL by programs that were five pages long, that would navigate through this labyrinth of pointers and stuff. Codd would sort of write them down as one-liners. ... They weren't complicated at all. I said, 'Wow.' This was kind of a conversion experience for me. I understood what the relational thing was about after that."

To Codd's disappointment, IBM proved slow to exploit his suggestions until commercial rivals started implementing them. Initially, IBM refused to implement the relational model at all for business reasons (to preserve revenue from its current database implementation—IMS/DB.

In 1973 IBM finally included the relational model of Codd in his plans, in System R subproject, but Codd was not involved in the project. Among the critical technologies developed for System R is the Structured Query Language (SQL), (initially called SEQUEL) developed by Chamberlin and Ray Boyce. Boyce later worked with Codd to develop the Boyce-Codd Normal Form for efficiently designing relational database tables so information was not needlessly duplicated in different tables.

In 1981 IBM released to market its first relational database product, SQL/DS. DB2, initially for large mainframe machines, was announced in 1983. IBM's DB2 family of databases proved to be one of IBMs most successful software products and is incorporated in the operating systems of mainframe and middleware servers of IBM.

Still in IBM, Codd continued to develop and extend his relational model. As the relational model started to become fashionable in the early 1980s, Codd fought a sometimes bitter campaign to prevent the term being misused by database vendors who had merely added a relational veneer to older technology. As part of this campaign, he published his famous 12 rules to define what constituted a relational database.

Codd retired from IBM in 1984 at the age of 61, after a serious injury resulting from a fall.

Later he joined up with the British database guru Chris Date, whom Codd had introduced to San Jose in 1971, to form the Codd and Date Consulting Group. The company, which included Codd's second wife Sharon Weinberg, made a good living from conducting seminars, writing books and advising major database vendors. Codd never became rich like the entrepreneurs like Larry Ellison, who exploited his ideas. He remained active as a consultant until 1999.

Codd was a holder of the Turing Award in 1981, and in 1994 he was inducted as a Fellow of the Association for Computing Machinery.

Edgar Codd died of heart failure at his home in Williams Island, Florida, on April 18, 2003, survived by his wife, four children and six grandchildren.

Edgar F. Codd, the father of the relational database model, died April 18, 2003, at the age of 79. Codd's groundbreaking work in the field of relational database management systems (RDBMSs) paved the way for what many of us do professionally. Database administrators' and developers' lives would be radically different today without Codd's work and the subsequent commercialization of the relational model by major database vendors.

Of course, the relational model seems simple and obvious today, but Codd's model was anything but obvious at the time, and it paved the way for a $7 billion industry that's at the center of all business-based computing today. This week, take a moment to honor the contributions of a giant in the world of database. If you haven't already done so, read Codd's original paper. You might be amazed at the innovation of a person who knew how to think outside the box--before most of us even knew there was a box.

****

Database Normalization

Normalization is a database design technique which organizes tables in a manner that reduces redundancy and dependency of data**.**

It divides larger tables to smaller tables and link them using relationships**.**

The inventor of the relational model **Edgar Codd** proposed the theory of normalization with the introduction of First **Normal Form** and he continued to extend theory with **Second and Third Normal Form.** Later he joined with **Raymond F. Boyce** to develop the theory of **Boyce-Codd Normal Form**.

Note: Most practical applications normalization achieves its best in 3rd Normal Form

First Normal Form

- Eliminate duplicative columns from the same table.
- Create a separate table for each set of related data.
- Identify each set of related data with a primary key.

Second Normal Form

- Meet all the requirements of the first normal form.
- Create separate tables for sets of values that apply to multiple records.
- Relate these tables with a foreign key.

Third Normal Form

- Meet all the requirements of the second normal form.
- Remove columns that are not dependent upon the primary key.

Boyce's Codd - Normal Form (BCNF or 3.5NF)

- The Boyce-Codd Normal Form, also referred to as the "third and half (3.5) normal form", adds one more requirement:Meet all the requirements of the third normal form.
- Every determinant must be a candidate key.

(**candidate key** is the combination of attributes that can be uniquely used to identify a database record without any extraneous data. Each table may have one or more candidate keys. One of these candidate keys is selected as the table primary key)

Examples:
Consider a database table that stores employee information and has the attributes employee_id, first_name, last_name, title. In this table, the field employee_id determines first_name and last_name. Similarly, the tuple (first_name, last_name) determines employee_id.

****

Edgar F. CODD'S RULES

Dr. E. F. Codd, an IBM researcher, first developed the relational data model in 1970. Over time it has proved to be flexible, extensible, and robust. In 1985 Codd published a list of 12 rules known as "**Codd's 12 Rules**" that defined how a true RDBMS should be evaluated. Understanding these rules will greatly improve the ability to understand RDBMS's in general, including Oracle. Also note that these rules are "guidelines" because, to date, no commercial relational database system fully conforms to all 12 rules.

It is important to remember that RDBMS is not a product, it is a method of design. Database products like the Oracle Database, SQL Server, DB2, Microsoft Access, etc. all adhere to the relational model of database design. The rules are as follows :

1. **Information Rule:** All information in a relational database including table names, column names are represented by values in tables.

2. **Guaranteed Access Rule**: Every piece of data in a relational database, can be accessed by using combination of a table name, a primary key value that identifies the row and column name which identified a cell.

3. **Systematic Treatment of Nulls Rule**: The RDBMS handles records that have unknown or inapplicable values in a pre-defined fashion.

4. **Active On-line catalog based on the relational model**: The description of a database and in its contents are database tables and therefore can be queried on-line via the data manipulation language. So authorized users must be able to access the database's structure(catalog) using the same query language that they use to access the same database's data

5. **Comprehensive Data Sub-language Rule:** A RDBMS may support several languages. But at least one of them should allow user to do all of the following:

- ✓ define tables and views,
- ✓ query and update the data,
- ✓ set integrity constraints,
- ✓ set authorizations and
- ✓ define transactions.

6. **View Updating Rule:** All views that is theoretically updateable can be update using RDBMS.Data consistency is ensured since the changes made in the view are transmitted to the base table and vice-versa.
7. **High-level Insert, Update and Delete:** The RDBMS supports insertions, updation and deletion at a table level. The performance is improved since the commands act on a set of records rather than one record at a time.

8. **Physical Data Independence**: The execution of adhoc requests and application programs is not affected by changes in the physical data access and storage methods.Database administrators can make changes to the physical access and storage method which improve performance and do not require changes in the application programs or requests. Here the user specified what he wants an need not worry about how the data is obtained.

9. **Logical Data Independence**: Logical changes in tables and views such adding/deleting columns or changing fields lengths need not necessitate modifications in the programs or in the format of adhoc requests.
For example, adding attribute or column to the base table should not disrupt the programs or the interactive command that have no use for the new attribute.

10. **Integrity Independence**: Like table/view definition, integrity constraints are stored in the on-line catalog and can therefore be changed without necessitating changes in the application programs.

11. **Distribution Independence**: Application programs and adhoc requests are not affected by change in the distribution of physical data. Improved systems reliability since application programs will work even if the programs and data are moved in different sites.

12. **No subversion Rule**: If the RDBMS has a language that accesses the information of a record at a time, this language should not be used to bypass the integrity constraints. This is necessary for data integrity.

****

Overviews of SQL

The Structured Query Language (SQL) comprises one of the fundamental building blocks of modern database architecture. SQL defines the methods used to create and manipulate relational databases on all major platforms.

SQL comes in many flavors. Oracle databases utilize their proprietary PL/SQL. Microsoft SQL Server makes use of Transact-SQL. However, all of these variations are based upon the industry standard ANSI SQL. SQL commands that will work on any modern relational database system.

SQL stands for “Structured Query Language” and can be pronounced as “SQL” or “sequel – (Structured English Query Language)”. It is a query language used for accessing and modifying information in the database.

IBM first developed SQL in 1970s. Also it is an ANSI/ISO standard. It has become a Standard Universal Language used by most of the relational database management systems (RDBMS).

Some of the RDBMS systems are: Oracle, Microsoft SQL server, Sybase etc. Most of these have provided their own implementation thus enhancing it's feature and making it a powerful tool. Few of the sql commands used in sql programming are SELECT Statement, UPDATE Statement, INSERT INTO Statement, DELETE Statement, WHERE Clause, ORDER BY Clause, GROUP BY Clause, ORDER Clause, Joins, Views, GROUP Functions, Indexes etc.

****

WHAT IS SQL?

SQL is the set of statements with which all programs and users access data in an Oracle database. A database stores data in form of tables which are composed of rows and columns.

SQL provides statements for a variety of tasks, including:

- **Querying data**
- **Inserting, updating, and deleting rows in a table**
- **Creating, replacing, altering, and dropping objects**
- **Controlling access to the database and its objects**

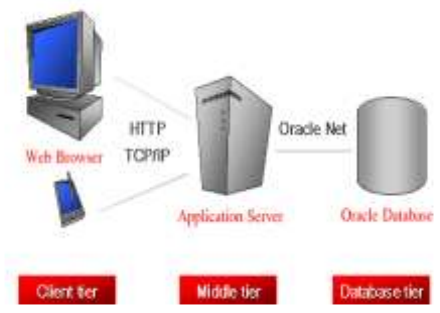
SQL*Plus is a command line environment that enables you to communicate with Database using Structured Query language(SQL). It is installed with every Oracle Database server or client installation.
SQL*Plus Command-line and Windows GUI Architecture

SQL*Plus command-line and the Windows GUI use a two-tier model comprising:

- ✓ Client (command-line user interface).
- ✓ Database (Oracle Database)

ISQL*Plus Architecture

iSQL*Plus is a browser-based interface which uses the SQL*Plus processing engine in a three-tier model comprising:
-- Client (Web browser).
-- Middle tier (Application Server).
-- Database (Oracle Database).



****

SQL Sub-Languages

SQL commands can be divided into sublanguages :

1. Data Definition language (DDL)

DDL is used to modify the schema of the database that describes three statements
CREATE ALTER DROP RENAME TRUNCATE
DDL commands will primarily be used by database administrators during the setup and removal phases of a database project

2. Data Manipulation Language (DML)

DML is used for inserting, deleting and updating data in a database that describes three statements:
INSERT UPDATE DELETE.
These commands will be used by all database users during the routine operation of the database.

3. Transaction Control Language (TCL)

TCL is used for managing changes affecting the data.
The commands are **COMMIT ROLLBACK SAVEPOINT**

4. Data Control Language (DCL)
DCL is used for providing security to database objects.
These commands are **GRANT and REVOKE**

5. Session Control Statement

Dynamically manage the properties of a user session. But do not implicitly commit the current transaction. The statements are **ALTER SESSION , SET ROLE**.

6. System Control Statement

The single system control statement, ALTER SYSTEM, **dynamically manages the properties of an Oracle Database instance.** This statement does not implicitly commit the current transaction and is not supported in PL/SQL.

7. Embedded SQL statements.

Embedded SQL statements place DDL, DML, and TCL within a procedural language program.

****

Data types in Oracle

Each value in ORACLE is manipulated by a Data Type. And these data type associates a fixed set of properties with that values stored.

The values of one data type are different from another data type, and these data type defines the domain of values that each column can contain**:**

Built-in Data Types

CHARACTER DATA TYPES
(CHAR, NCHAR, VARCHAR2, NVARCHAR)

NUMBERIC DATA TYPES
(Zero, Positive & Negative Fixed & Floating Point Numbers)

DATE & TIME DATA TYPES
(Century, Year, Month, Date, Hour, Minute, Second)

LONG & RAW DATA TYPES
(Storage of binary data or bytes strings – Ex: Graphics, Sounds, Scanned Documents

LARGE OBJECT DATA TYPES – (BLOB, CLOB, NLOB)
(It can store large & unstructured data like text, Image, Video & Spatial data)

ROWID DATA TYPES
(Support Partitioned tables & Indexes)

BFILE DATA TYPES
(Binary file LOB's – stored in the systems outside ORACLE)

****

The More in Details

CHAR Data Type:

- It specifies fixed length character strings.
- The size should be specified, if the data is less than the original specified size, balnk pads are applied.
- The default length is 1 Byte and the Maximum is 2000 Bytes.
  Also the size of a character can range from 1 Byte to 4 Bytes depending on the database character set.

EXAMPLE:
CHAR(10)

NCHAR Data Type:

- First time defined in Oracle 9i, and contains Unicode data only.
- Maximum length is determined by the National Character Set Definition.
- Maximum size is 2000 Bytes & size has to be specified.
- Padded if data is shorter than specified.

EXAMPLE:
NCHAR(10)

VARCHAR2 Data Type:

- Main advantage is Storing character data as Varchar2 will save space
- Specifies the Variable Length Character String.
- Minimum size is 1 Byte and the Maximum size is 4000 Bytes.
- It occupies only that space for which the data is supplied

EXAMPLE:
VARCHAR2(10)

NVARCHAR2 Data Type:

- First time defined in Oracle 9i
- It is defined for UNICODE data only
  The Minimum size is 1 Byte & Maximum size is 4000 Bytes.

EXAMPLE:
NVARCHAR2(10)

NUMBER Data Type:

- It stores Zero, Positive & Negative Fixed and Floating Point numbers.
- The range of magnitude is
- $1.0*10^{-130}$ to $9.9\ldots9*10^{-125}$
- The general declaration is Number(P,S)
- P→ It specifies the precision ie the total number of digits (1 to 38)
- S→ It specifies the scale ie the number of digits to the right of the decimal point, it can range from -84 to 127

EXAMPLE:
NUMBER(10,2)

FLOAT DATATYPE

It facilitates to have a decimal point anywhere from the first to the last digit, OR can have no decimal point at all.

Syntax:

- It specifies to have decimal point any Where from the first to the last digit or can have no decimal point at all.
- The Scale values is not applicable to floating point numbers, as the number of digits that can appear after the decimal point is not restricted.

- **FLOAT** → It specifies a floating point numbers with decimal precision 38 or binary precision of 126
- **FLOAT(b)**→ It specifies a floating point number with binary precision b
- The precision can range from 1 to 126
- To convert from binary to decimal precision multiply 'b' by 0.30103
- To convert from decimal to binary precision multiply the decimal precision by 3.32193
- The maximum of 126 digits of binary precision is roughly equivalent to 38 digits of decimal precision.

EXAMPLE:
Fixed-point NUMBER

NUMBER(p,s)
precision p = length of the number in digits
scale s = places after the decimal point, or (for negative scale values) significant places before the decimal point.

Integer NUMBER

NUMBER(p)
This is a fixed-point number with precision p and scale 0. Equivalent to NUMBER(p,0)

Floating-Point NUMBER

NUMBER
floating-point number with decimal precision 38

Confusingly the Units of measure for PRECISION vary according to the datatype.
For NUMBER data types: precision p = Number of Digits
For FLOAT data types: precision p = Binary Precision (multiply by 0.30103 to convert)

{So FLOAT = FLOAT (126) = 126 x 0.30103 = approx 37.9 digits of precision.}

Example

The value 7,456,123.89 will display as follows

NUMBER(9)	7456124
NUMBER(9,1)	7456123.9
NUMBER(*,1)	7456123.9
NUMBER(9,2)	7456123.89
NUMBER(6)	[not accepted exceeds precision]
NUMBER(7,-2)	7456100
NUMBER	7456123.89
FLOAT	7456123.89
FLOAT(12)	7456000.0

LONG DATATYPES

- This data type stores variable length character strings.
- The maximum storage size is up to 2Gb.
- It is used to store very lengthy text strings.
- The length of LONG values may be limited by the memory available on the computer.
- LONG columns can be referenced in
  - Select lists.
  - SET clauses of UPDATE statements.
  - VALUES clauses of INSERT statements.

RESTRICTIONS FOR LONG DATATYPE

- A single table can contain only one LONG column.
- Object types cannot be created on LONG attribute.
- LONG columns cannot appear in Where clause or in integrity constraints.
- Indexes cannot be created on LONG columns.
- LONG can be returned through a functions, but not through a stored procedure.
- It can be declared in a PL/SQL unit but cannot be referenced in SQL.

DATE & TIME DATATYPES

- It is used to stored date and time information.
- The dates can be specified as literals, using the calendar.

- The information revealed by date is :
  CENTURY , YEAR, MONTH,DATE,HOUR,MINUTE,SECOND
- The default date format in oracle is DD-MON-YY, and is specified in NLS_DATE_FORMAT.
- The default time accepted by oracle date is 12:00:00Am (mid night)
- The default date accepted by oracle date is the first day of the current month.
- The date range provided by oracle is January 1,4712 B.c to Dec 31,9999 A.d

TIMESTAMP DATATYPE

- It is an extension of the Date data type.
- It stores - DAY,MONTH,YEAR,HOUR,MINUTE,SECOND

SYNTAX:
TIMESTAMP [{Fraction-second-precision}]

- Fraction_seconds_precision optionally specifies the number of digits in the fractional part of the second date time field
- It can be a number in the range of 0-9, with default as 6.

RAW AND LONG DATATYPES

- RAW and LONG RAW data types are intended for storage of binary data or byte strings.
- RAW and LONG RAW are variable length data types.
- They are mostly used to store graphics, sounds, documents etc.
- The oracle converts the RAW and LONG RAW data into hexadecimal form.
- Each hexadecimal character represents four bits of RAW data.

LARGE OBJECT (LOB) DATATYPES

- The built in LOB data types are
  - BLOB , CLOB, NCLOB
  - These data types are stored internally.
- The BFILE is an LOB which is stored externally.
- The LOB data types can store large and unstructured data like text, image, video and spatial data.
- The maximum size is up to 4Gb.
- LOB columns contain Lob location, which can refer to out-of-line or in-line LOB values.
- LOB's Selection actually returns the LOB's locator.

BFILE DATATYPE

- It enables access to binary file LOB's which are stored in the file systems outside oracle.
- A BFILE column or the attributes stores the BFILE locator.
- The BFILE locator maintains the directory alias and the filEname.
- The binary file LOB's do not participate in transactions and are not recoverable.
- The maximum size is 4 Gb.

BLOB DATATYPE

- It stores unstructured binary large objects.
- They are bit streams with no character set semantics.
- They are provided with full transactional support.

CLOB DATATYPE

- It stores single byte and multi byte character data.
- Both fixed width and variable width character sets are supported.
- They are provided with full transactional support.

NCLOB DATATYPE

- It stores Unicode data using the national character set.

****

Syntax for Creating table statements with Examples

SYNTAX

```
CREATE TABLE <table_name>
(column_name1 <Datatype> (width),
 column_name1 <Datatype> (width),
 column_name1 <Datatype> (width)
)
```

```
CREATE TABLE Student
(
            Stud_id  NUMBER (6),
            First_Name     VARCHAR2(30),
            Last_Name      VARCHAR2 (30),
            Dob            DATE,
            Doj            DATE,
            Fees           NUMBER (8,2),
            Gender         CHAR (1)
);
```

```
CREATE TABLE Studio
(
            Id NUMBER (6),
            Photo BLOB,
            Graphic BFILE,
            Description LONG
)
```

```
DROP TABLE Student;                    DROP TABLE Studio;
```

The default format for date is DD-MM-YYYY (Ex: 21-November-2005)

****

SQL Integrity Constraints

CONSTRAINT clause allows you to restrict the data that is entered into a column to ensure that it is valid or that it meets certain conditions. Integrity Constraints are used to apply business rules for the database tables.

Properly written constraints give you as much control over the data in your tables as you need. If an attempt is made to store data in a column that violates a constraint, an error (or exception) is raised.

Constraints can be defined when a table is first created via the CREATE TABLE statement, or after the table is already created by using the ALTER TABLE statement.

The constraints available in SQL are **Foreign Key, Not Null, Unique, Check.**

Advantages of Constraint Names:

Constraint naming standard is important for one reason: The SYS_* name oracle assigns to unnamed constraints is not very understandable. By correctly naming all contraints, we can quickly associate a particular constraint with our data model. This gives us two real advantages:

- We can quickly identify and fix any errors.
- We can reliabily modify or drop constraints

Constraints can be defined in two ways

1) The constraints can be specified immediately after the column definition.
This is called **column-level definition**.

2) The constraints can be specified after all the columns are defined.
This is called **table-level definition**.

1) Primary key:

A primary is a single column values used to uniquely identify a database record.

It has following attributes

✓ A primary key cannot be NULL
A primary key value must be unique
✓ The primary key values can not be changed
✓ The primary key must be given a value when a new record is inserted.
This constraint defines a column or combination of columns which uniquely identifies each row in the table.

Syntax to define a Primary key at column level:

column name datatype [CONSTRAINT constraint_name] PRIMARY KEY

Syntax to define a Primary key at table level

[CONSTRAINT constraint_name] PRIMARY KEY (column_name1,column_name2,..)

column_name1, column_name2 are the names of the columns which define the primary Key.

The syntax within the bracket i.e.

[CONSTRAINT constraint_name] is optional.

For Example:
To create an employee table with Primary Key constraint, the query would be like.

Primary Key at column level

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10)
);
```

OR

```
CREATE TABLE employee
( id number(5) CONSTRAINT emp_id_pk PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10)
);
```

Primary Key at table level:

```
CREATE TABLE employee
( id number(5),
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10),
CONSTRAINT emp_id_pk PRIMARY KEY (id)
);
```

More Examples:

create table dept (dept_id number(9) primary key);

alter table dept add primary key(dept_id);

alter table dept add constraint dept_pk primary key(dept_id);

alter table user add constraint user_pk primary key(fname,lname);
Note: the shows that you can give the index a name instead of a system generated one.

2) Foreign key or Referential Integrity :

This constraint identifies any column referencing the PRIMARY KEY in another table. It establishes a relationship between two columns in the same table or between different tables.

For a column to be defined as a Foreign Key, it should be a defined as a Primary Key in the table which it is referring. One or more columns can be defined as Foreign key.

Syntax to define a Foreign key at column level:

[CONSTRAINT constraint_name] REFERENCES Referenced_table_name(column_name)

Syntax to define a Foreign key at table level:

[CONSTRAINT constraint_name] FOREIGN KEY(column_name) REFERENCES
referenced_table_name(column_name);

For Example

1) Lets use the "product" table and "order_items".

Foreign Key at column level

[CONSTRAINT constraint_name] REFERENCES Referenced_table_name(column_name)

CREATE TABLE product
(product_id number(5) **CONSTRAINT pd_id_pk PRIMARY KEY**,
product_name char(20),
supplier_name char(20),
unit_price number(10)
);

CREATE TABLE order_items
(order_id number(5) **CONSTRAINT od_id_pk PRIMARY KEY**,
product_id number(5) **CONSTRAINT pd_id_fk REFERENCES, product(product_id)**,
product_name char(20),
supplier_name char(20),
unit_price number(10)
);

Foreign Key at table level

[CONSTRAINT constraint_name] FOREIGN KEY(column_name) REFERENCES referenced_table_name(column_name);

CREATE TABLE order_items
(order_id number(5) ,
product_id number(5),
product_name char(20),
supplier_name char(20),
unit_price number(10)
CONSTRAINT od_id_pk PRIMARY KEY(order_id),
CONSTRAINT pd_id_fk FOREIGN KEY(product_id) REFERENCES product(product_id)
);

2) If the employee table has a 'mgr_id' i.e, manager id as a foreign key which references primary key 'id' within the same table, the query would be like,

CREATE TABLE employee
(id number(5) **PRIMARY KEY**,
name char(20),
dept char(10),
age number(2),
mgr_id number(5) **REFERENCES employee(id)**,
salary number(10),
location char(10)
);

More Examples

create table employee (

```
 employee_id number(7),
 last_name varchar2(30),
 first_name varchar2(30),
 dept_id number(3) not null,
constraint dept_fk references dept(dept_id));
```

Note: To create a foreign key constraint on an object in a different schema you must have the REFERENCES privilege on the columns of the referenced key in the parent table or view.

3) SQL Not Null Constraint

This constraint ensures all rows in the table contain a definite value for the column which is specified as not null. Which means a null value is not allowed.

Syntax to define a Not Null constraint

[CONSTRAINT constraint name] NOT NULL

For Example:

To create a employee table with Null value, the query would be like

```
CREATE TABLE employee
( id number(5),
name char(20) CONSTRAINT constraint_name NOT NULL,
dept char(10),
age number(2),
salary number(10),
location char(10)
);
```

```
create table employee (nat_sur varchar(9) not null);
alter table employee modify dob not null;
```

4) SQL Unique Key:

This constraint ensures that a column or a group of columns in each row have a distinct value. A column(s) can have a null value but the values cannot be duplicated.

Syntax to define a Unique key at column level:

[CONSTRAINT constraint_name] UNIQUE

Syntax to define a <u>Unique key at table level</u>

[CONSTRAINT constraint_name] UNIQUE(column_name)

For Example:

To create an employee table with Unique key, the query would be like,

<u>Unique Key at column level</u>

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10) UNIQUE
);
```

OR

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10) CONSTRAINT loc_un UNIQUE
);
```

Unique Key at table level

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10),
CONSTRAINT loc_un UNIQUE(location)
);
```

More Examples

```
create table dept (
 dept_no number(3),
 dept_name varchar2(15),
 location varchar2(25),
constraint dept_name_ukey unique(dept_name,location));
```

```
alter table dept add constraint dept_idx unique(dept_no);
alter table dept add constraint name_location_idx unique(dept_name,location);
```

5) SQL Check Constraint:

This constraint defines a business rule on a column. All the rows must satisfy this rule. The constraint can be applied for a single column or a group of columns.

Syntax to define a Check constraint:

[CONSTRAINT constraint_name] CHECK (condition)

For Example: In the employee table to select the gender of a person, the query would be like

Check Constraint at column level:

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
gender char(1) CHECK (gender in ('M','F')),
salary number(10),
location char(10)
);
```

Check Constraint at table level:

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
gender char(1),
salary number(10),
location char(10),
CONSTRAINT gender_ck CHECK (gender in ('M','F'))
);
```

alter table employee add constraint gender_chk check(gender in('M','F'));

To know information about Constraints by using Dictionary Views:

DBA_CONSTRAINTS

It describes all constraint definitions on all tables in the database including the search condition.

DBA_CONS_COLUMNS

You can query this view to find the names of constraints, what columns they affect, and other information to help you manage constraints

Constraint States

Sometimes when you are loading data you need to override the constraints , oracle allows you to disable the constraint thus speeding up the data loading, there are 4 states

disable validate state

the validate part ensures that the data in the table satisfies the constraint , the disable part you're doing away with the requirements of maintaining the constraint meaning the index, so table is correct but index will not be.

disable no validate state

the table and the index are not maintained
enable validate state - ensures that all data is checked to ensure compliance with the constraint

enable no validate state

all new data will be checked but the existing data will not be .

Altering	Constraints cannot be altered they must be dropped and created.
Remove a constraint	alter table <table name> drop constraint <constraint name>;
Rename a constraint	alter table <table name> rename constraint <old name> to <new name>;
Enable/Disable a constraint	alter table <table name> disable constraint <constraint name>; alter table <table name> enable constraint <constraint name>;

Display constraint condition	column search_condition format a50;
	select constraint_name, constraint_type, table_name, search_condition from user_constraints;

Constraint States

Disable validate state	alter table sales_data add constraint quantity_unique unique (prod_id,customer_id) disable validate;
Disable no validate state	alter table sales_data add constraint quantity_unique unique (prod_id,customer_id) disable no validate;
Enable validate state	alter table sales_data add constraint sales_region_fk foreign key (sales_region) references region(region_id) enable validate;
Enable no validate state	alter table sales_data add constraint sales_region_fk foreign key (sales_region) references region(region_id) enable no validate;

Deferrable and Immediate Constraints

In Oracle you can specify when the constraint is to be checked after each modification (not deferrable) which is the default behavior in oracle or a one time check after the whole transaction is committed (deferrable). If you choose deferrable there are a further two options, initially deferred (will defer checking until the transaction completes) or initially immediate (check the constraint before any data is changed).

Not deferrable	create table employee (employee_id number, dept varchar2(30) unique references department(dept_name)); **Note: by default oracle set it as not deferrable**

Deferrable	create table employee (employee_id number, dept varchar2(30) unique references department(dept_name) **deferrable initially deferred**); set constraint <constraint name> **deferred**; set constraint <constraint name> **immediate**;

(I recommend you to go through this link for the new features about Constraints, ones you finished reading my method)
http://docs.oracle.com/cd/B28359_01/server.111/b28310/general005.htm

****

Two ways to run sql commands:

1. Interactive mode: type commands directly at the SQL prompt.
At the SQL*PLUS prompt, you can type in any SQL query. The query will be sent to the Oracle server for executing and the results will be given back to your terminal interactively.

For instance to create a table 'STUDENT', type at prompt:

2. Batch: create a set of commands in a file and run it in SQL*PLUS.
You can use the internal SQL command line editor to make changes and then reexecute. The internal SQL command line editor only works with your last SQL command.

For example: define the default editor using define_editor in sql prompt
SQL> **define_editor=vi**

Now you can **use EDIT** to invoke a text editor on the contents of the buffer (the last SQL command)), then make your changes, save and exit the editor. You then return to the SQL and reexecute your command by entering slash (**/**) at the prompt,

```
SQL>
SQL> ed
```

The following commands to make changes based on the situation in the sql prompt.

Command	Abbr.	Purpose
APPEND text	A text	adds text at the end of a line
CHANGE /old/new	C /old/new	changes old to new in a line
CHANGE /text	C /text	deletes text from a line
CLEAR BUFFER	CL BUFFER	deletes all lines
DEL	(none)	deletes the current line
DEL n	(none)	deletes line n
DEL *	(none)	deletes the current line
DEL LAST	none)	deletes the last line
DEL m n	(none)	deletes lines from m to n
INPUT	I	adds one or more lines

INPUT text	I text	adds a line consisting of text
LIST	L	lists all lines in the SQL buffer
LIST n	L n or n	lists line n
LIST *	L *	lists the current line
LIST LAST	L LAST	lists the last line
LIST m n	L m n	lists a range of lines (m to n)

After you change the buffer, you can re-execute it by entering slash (**/**).

SQL*PLUS : Miscellaneous

This is the closing page for the SQL*PLUS. Some useful miscellaneous items are listed here:

Save your last SQL stmt.

SQL> SAVE [filename]

Spool the screen to a file

SQL> SPOOL [filename]

the SQL*PLUS will start recording your displayed output into your specific [filename] with the default file extension '.lst'. To stop spooling and save the file:

SQL> SPOOL OFF

Dual table: It is a dummy table which is generally used to perform some calculation is seeing to the system date and etc. Dual table is collection of one row and one column with 'X' in it.

Ex:
Select SYSDATE from dual;
Select 10+20 from dual;
Select 20+40, 50+60 from dual;

ROWNUM: It is ROWNUM is a pseudo column which starts with one and increment by 1. (1 and 1+1)

Syn: SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number

Ex:
Select Rownum, empno, ename, sal, deptno from emp;

Rownum values are temporary.
Rownum values are generated when query is executed.
Rownum values generation always starts with one and increment by one.

Query to display first three rows from emp table?
Select * from emp where Rownum <=3;
Select * from emp where Rownum <=10;

Write a query to display 5th row of emp table?
Select * from emp where Rownum <=5
Minus
Select * from emp where Rownum <=4; //5th row is display

Write a query to display 3rd row to 7th row?
Select * from emp where Rownum <=7

Minus
Select * from emp where Rownum <=2; //3rd to 7th row display

Example
SELECT *
FROM Persons
WHERE ROWNUM <=5

ROWID:

ROWID is pseudo column which contains hexadecimal values.
ROWID value indicates the address where the row is store in the database.
ROWID values are permanent.

Ex:
Select ROWID, empno, ename, sal, deptno from emp;
Select min(rowid) from emp;

Difference between ROWNUM and ROWID?

ROWNUM	**ROWID**
1.Rownum values starts with 1 and increment by one.	1. Rowid's are hexadecimal values.
2.Rownum values are temporary.	2. Rowid values are permanent.
3.Rownum values are generated when query is exiecuted.	3. The Rowid values are generated when Row is created or inserted.

****

SQL COMMANDS (with more examples)

~~~~~~~~~~~~~~~~~~~~~~

SQL CREATE TABLE Statement

~~~~~~~~~~~~~~~~~~~~~~

CREATE TABLE is used in creation of table(s).

To create a table, the basic structures to hold user data, specifying this information are: column definitions, integrity constraints, the table's tablespace, storage characteristics, an optional cluster, data from an arbitrary query.

The Syntax for the CREATE TABLE Statement is:

```
CREATE TABLE table_name
(column_name1 datatype,
column_name2 datatype,
... column_nameN datatype
);
```

- ➔ **table_name** - is the name of the table.
- ➔ **column_name1, column_name2....** - is the name of the columns
- ➔ **datatype** - is the datatype for the column like char, date, number etc.

Example: If you want to create the diffrent tables, the statements would be like,

```
SQL> CREATE TABLE Student
( Name  VARCHAR2(30),
StudentNumber NUMBER(4) ,
Class NUMBER(4),
Major VARCHAR2(4),
Primary key (StudentNumber)
```

```
CREATE TABLE employee
( id number(5),
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10)
```

);	);

In Oracle database, the datatype for an integer column is represented as "number". In Sybase it is represented as "int".

Oracle provides another way of creating a table.

CREATE TABLE temp_employee
SELECT * FROM employee

In the above statement, temp_employee table is created with the same number of columns and datatype as employee table.

Creating duplicate tables or backup tables: By using the combination of create and select.

We can create copy of a table.
Syn: CREATE TABLE.. AS SELECT command

Explanation :
To copy only the structure, the WHERE clause of the SELECT command should contain a FALSE statement as in the following.

Syn:

Structure and Data
CREATE TABLE NEWTABLE AS SELECT * FROM EXISTINGTABLE;

Structure of a table:
CREATE TABLE NEWTABLE AS SELECT * FROM EXISTINGTABLE WHERE 1=2;

If the WHERE condition is true, then all the rows or rows satisfying the condition will be copied to the new table.
Ex:

```
Create table emp1 AS select * from emp; //total table copy
Create table emp2 AS select * from emp where deptno = 30; // only deptno = 30
Create table emp3 AS select * from emp where 10; // only deptno =10 is copy
Create table emp4 AS select empno, ename, wages, deptno from emp where
deptno = 20; //empno,ename,wages,deptno is coped by emp4 table
Create table emp5 AS select *from emp where 1=2; //This mean total table copy

Select * from emp where 1 = 1;
Select * from 'malli' from emp;
```

~~~~~~~~~~~~~~~~~
SQL INSERT Statement
~~~~~~~~~~~~~~~~~

INSERT The INSERT Statement is used to add new rows of data to a table.

Insert statement to add rows to a table, the base table of a view, a partition of a partitioned table or a subpartition of a composite-partitioned table, or an object table or the base table of an object view.

INSERT INTO TABLE_NAME **[(col1, col2, col3,...colN)]** **VALUES (value1, value2, value3,...valueN);** SQL> INSERT INTO employee (id, name, dept, age, salary location) VALUES (105, 'Srinath', 'Aeronautics', 27, 33000);	**INSERT INTO TABLE_NAME** **VALUES (value1, value2, value3,...valueN);**); SQL> INSERT INTO Student VALUES ('Smith', 17, 1, 'CS');

Also Inserting data to a table through a select statement.

Syntax for SQL INSERT is:

INSERT INTO table_name
[(column1, column2, ... columnN)]
SELECT column1, column2, ...columnN
FROM table_name [WHERE condition];

For Example: To insert a row into the employee table from a temporary table, the sql insert query would be like,

SQL> INSERT INTO employee (id, name, dept, age, salary location) SELECT emp_id, emp_name, dept, age, salary, location FROM temp_employee;

If you are inserting data to all the columns, the above insert statement can also be written as,

SQL> INSERT INTO employee
SELECT * FROM temp_employee;

While inserting a row, if you are adding value for all the columns of the table you need not specify the column(s) name in the sql query. But you need to make sure the order of the values is in the same order as the columns in the table.

- ➔ For you to insert rows into a table, the table must be in your own schema or you must have INSERT privilege on the table.
- ➔ For you to insert rows into the base table of a view, the owner of the schema containing the view must have INSERT privilege on the base table. Also, if the view is in a schema other than your own, you must have INSERT privilege on the view.
- ➔ The INSERT ANY TABLE system privilege also allows you to insert rows into any table or any view's base table.

The INSERT Statement is used to ADD rows to a

- ✓ Relational Table,
- ✓ Views Base Table,
- ✓ A partition of a Partition Table,
- ✓ A Sub Partition of a Composite Partitioned Table
- ✓ An Object Table
- ✓ An Object View’s Base Table,

IMPORTANT NOTE:

1) When adding a new row, you should ensure the datatype of the value and the column matches

2) You follow the integrity constraints, if any, defined for the table.

****

~~~~~~~~~~~~~~~~

SQL SELECT COMMAND

~~~~~~~~~~~~~~~~~

SELECT: To retrieve data from one or more tables, views, or snapshots

- ➔ For you to select data from a table or snapshot, the table or snapshot must be in your own schema or you must have SELECT privilege on the table or snapshot.

- ➔ For you to select rows from the base tables of a view, the owner of the schema containing the view must have SELECT privilege on the base tables. Also, if the view is in a schema other than your own, you must have SELECT privilege on the view.

- ➔ The SELECT ANY TABLE system privilege also allows you to select data from any table or any snapshot or any view's base table.

SYNTAX:

SELECT column name1, column name2, …. FROM tablename;

If we want to select all column values from a table then SQL command use is

Example:

SELECT * from tablename ;

To select all senior (class = 5) from the Student Table, enter

SQL> SELECT Name, Major FROM Student WHERE Class = 5;

Oracle shows something like the following as the results:

```
NAME                  MAJO
-------------------   -------
Senior Answer1        COSC
Benjamin Bayer        EPW
Senior Crew           COSC
```

****

~~~~~~~~~~~~~~~~~~~~~

SQL ALTER TABLE Statement

~~~~~~~~~~~~~~~~~~~~~

ALTER TABLE: If you want to modify a table definition we can use this command. This command is used to alter column definition in table.

To alter the definition of a table in one of these ways:

→ to add a column
→ to add an integrity constraint
→ to redefine a column (datatype, size, default value)
→ to modify storage characteristics or other parameters
→ to enable, disable, or drop an integrity constraint or trigger
→ to explicitly allocate an extent
→ to allow or disallow writing to a table
→ to modify the degree of parallelism for a table

SYNTAX1:

ALTER TABLE tablename MODIFY (Column definition);

SYNTAX2:

```
ALTER TABLE [schema.]table
     [ADD {    { column datatype [DEFAULT expr] [column_constraint] ...
           | table_constraint}
        |  ( { column datatype [DEFAULT expr] [column_constraint] ...
           | table_constraint}
         [, { column datatype [DEFAULT expr] [column_constraint] ...
           | table_constraint} ] ... ) } ]
     [MODIFY {   column [datatype] [DEFAULT expr] [column_constraint] ...
          |  (column [datatype] [DEFAULT expr] [column_constraint] ...
   [, column datatype [DEFAULT expr] [column_constraint] ...] ...) } ]
     [DROP drop_clause] ...
```

schema : is the schema containing the table. If you omit schema, Oracle assumes the table is in your own schema.
table : is the name of the table to be altered.
ADD : adds a column or integrity constraint.
MODIFY : modifies a the definition of an existing column. If you omit any of the optional parts of the column definition (datatype, default value, or column constraint), these parts remain unchanged.
column : is the name of the column to be added or modified.
datatype : specifies a datatype for a new column or a new datatype for an existing column.
DEFAULT : specifies a default value for a new column or a new default for an existing column. Oracle assigns this value to the column if a subsequent INSERT statement omits a value for the column. The datatype of the default value must match the datatype specified for the column. A DEFAULT expression cannot contain references to other columns, the pseudocolumns CURRVAL, NEXTVAL, LEVEL, and ROWNUM, or date constants that are not fully specified.
column_constraint : adds or removes a NOT NULL constraint to or from and existing column.
table_constraint : adds an integrity constraint to the table.

Syntax to add a column

```
ALTER TABLE table_name ADD column_name datatype;
```

For Example: To add a column "experience" to the employee table, the query would be like

```
ALTER TABLE employee ADD experience number(3);
```

Syntax to drop a column

```
ALTER TABLE table_name DROP column_name;
```

For Example: To drop the column "location" from the employee table, the query would be like

```
ALTER TABLE employee DROP location;
```

Syntax to modify a column

```
ALTER TABLE table_name MODIFY column_name datatype;
```

For Example: To modify the column salary in the employee table, the query would be like

```
ALTER TABLE employee MODIFY salary number(15,2);
```

****

**~~~~~~~~~~~~~~~~~**

SQL RENAME Command

**~~~~~~~~~~~~~~~~~**

The SQL RENAME command is used to change the name of the table or a database object.

In Oracle 9i and above we have a simple rename column command that makes it easy to rename any table column. Internally, the rename column syntax adjusts the Oracle data dictionary only, since the column names are not stored directly inside the segment itself:

```
SQL> desc sales

Name             Null?    Type
---------------- -------- ---------------
STORE_KEY                  VARCHAR2(4)
BOOK_KEY                   VARCHAR2(6)
ORDER_NUMBER                   VARCHAR2(20)
ORDER_DATE                   DATE
```

SQL> alter table sales rename column order_date to date_of_order;

```
SQL> desc sales
Name             Null?    Type
---------------- -------- ---------------
STORE_KEY                  VARCHAR2(4)
BOOK_KEY                   VARCHAR2(6)
ORDER_NUMBER               VARCHAR2(20)
DATE_OF_ORDER              DATE
```

alter table sales rename to sales1; **-->** new in Oracle sql 11g

****

**~~~~~~~~~~~~~~~~**

SQL Delete Statement

**~~~~~~~~~~~~~~~~**

DELETE: In order to delete rows from a table we use this command
SYNTAX: DELETE FROM tablename WHERE condition;

Based on the condition specified the rows gets fetched from the table and gets deleted in table. Here the WHERE clause is optional.

SYNTAX:	**Example:**
DELETE [FROM] [schema.]{table | view}[@dblink] [WHERE condition]	To delete all students who have major in 'FRENCH', enter: SQL> DELETE FROM Student WHERE MAJOR = 'FRENCH';

schema : is the schema containing the table or view. If you omit schema, Oracle assumes the table or view is in your own schema.
table | view : is the name of a table from which the rows are to be deleted. If you specify view, Oracle deletes rows from the view's base table.
dblink : is the complete or partial name of a database link to a remote database where the table or view is located. You can only delete rows from a remote table or view if you are using Oracle with the distributed option. If you omit dblink, Oracle assumes that the table or view is located on the local database.
WHERE : deletes only rows that satisfy the condition. The condition can reference the table and can contain a **subquery**. If you omit this clause, Oracle deletes all rows from the table.

- ➔ For you to delete rows from a table, the table must be in your own schema or you must have DELETE privilege on the table.
- ➔ For you to delete rows from the base table of a view, the owner of the schema containing the view must have DELETE privilege on the base table. Also, if the view is in a schema other than your own, you must be granted DELETE privilege on the view.
- ➔ The DELETE ANY TABLE system privilege also allows you to delete rows from any table or any view's base table.

****

**~~~~~~~~~~~~~~~~~~**

SQL UPDATE STATEMENT

**~~~~~~~~~~~~~~~~~~**

UPDATE: This SQL command is used to modify the values in an existing table.

SYNTAX:	**Example:**

UPDATE table_name
SET column_name1 = value1,
column_name2 = value2, ...
[WHERE condition]
	To update the location of an employee, the sql update query would be like,
	SQL> UPDATE employee
	SET location ='Mysore'
	WHERE id = 101;
	To change the salaries of all the employees, the query would be,
	SQL>UPDATE employee
	SET salary = salary + (salary * 0.2);SQL> DELETE FROM Student
	WHERE MAJOR = 'FRENCH';

table_name - the table name which has to be updated.

column_name1, column_name2.. - the columns that gets changed.

value1, value2... - are the new values.

NOTE:

In the Update statement, WHERE clause identifies the rows that get affected. If you do not include the WHERE clause, column values for all the rows get affected.

- ➔ The rows which satisfies the WHERE condition are fetched and for these rows the column values we placed in command above in SET statement gets updated.
- ➔ For you to update values in a table, the table must be in your own schema or you must have UPDATE privilege on the table.
- ➔ For you to update values in the base table of a view, the owner of the schema containing the view must have UPDATE privilege on the base table. Also, if the view is in a schema other than your own, you must have UPDATE privilege on the view.
- ➔ The UPDATE ANY TABLE system privilege also allows you to update values in any table or any view's base table.

****

**~~~~~~~~~~~~~**
SQL DROP TABLE
**~~~~~~~~~~~~~**

DROP TABLE: This command is used to remove an object from the database. If you drop a table, all the rows in the table is deleted and the table structure is removed from the database. Once a table is dropped we cannot get it back, so be careful while using RENAME command. When a table is dropped all the references to the table will not be valid.

SYNTAX:	**Example:**
DROP TABLE [schema.]table [CASCADE CONSTRAINTS]	SQL> DROP TABLE Student

schema : is the schema containing the table. If you omit schema, Oracle assumes the table is in your own schema.

table : is the name of the table to be dropped.

CASCADE CONSTRAINTS

drops all referential integrity constraints that refer to primary and unique keys in the dropped table. If you omit this option, and such referential integrity constraints exist, Oracle returns an error and does not drop the table.

The table must be in your own schema or you must have DROP ANY TABLE system privilege.

****

SQL TRUNCATE Statement

The SQL TRUNCATE command is used to delete all the rows from the table and free the space containing the table.

Syntax to TRUNCATE a table:

TRUNCATE TABLE table_name;

For Example: To delete all the rows from employee table, the query would be like,

TRUNCATE TABLE employee;

****

Difference between DELETE and TRUNCATE Statements:

DELETE Statement: This command deletes only the rows from the table based on the condition given in the where clause or deletes all the rows from the table if no condition is specified. But it does not free the space containing the table.
TRUNCATE statement: This command is used to delete all the rows from the table and free the space containing the table.

****

Difference between DROP and TRUNCATE Statement:

If a table is dropped, all the relationships with other tables will no longer be valid, the integrity constraints will be dropped, grant or access privileges on the table will also be dropped, if want use the table again it has to be recreated with the integrity constraints, access privileges and the relationships with other tables should be established again. But, if a table is truncated, the table structure remains the same, therefore any of the above problems will not exist.

****

SQL Rollback COMMAND

ROLLBACK : This SQL command is used to undo the current transaction

SYNTAX: ROLLBACK;

To undo work done in the current transaction.

-- You can also use this command to manually undo the work done by an in-doubt distributed transaction.

-- To roll back your current transaction, no privileges are necessary.

-- To manually roll back an in-doubt distributed transaction that you originally committed, you must have FORCE TRANSACTION system privilege.

-- To manually roll back an in-doubt distributed transaction originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

****

SAVEPOINT: This is used to identify a point in a transaction to which we can later rollback.

SYNTAX: SAVEPOINT savepoint_identifier;

****

COMMIT: This command is used to make all changes permanent in database and also marks the end of transaction.

SYNTAX: COMMIT;

Always it is better to commit changes to database at regular intervals since it will help loss of data or loss of work done when computer shuts due to unavoidable reasons.

-- You need no privileges to commit your current transaction.
-- To manually commit a distributed in-doubt transaction that you originally committed, you must have FORCE TRANSACTION system privilege.
-- To manually commit a distributed in-doubt transaction that was originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

****

**~~~~~~~~~~~~~~~~**

SQL GRANT Command

**~~~~~~~~~~~~~~~~**

SQL GRANT is a command used to provide access or privileges on the database objects to the users.

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.

object_name is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.

user_name is the name of the user to whom an access right is being granted.

user_name is the 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.

For Eample: GRANT SELECT ON employee TO user1;This command grants a SELECT permission on employee table to user1.You should use the WITH GRANT option carefully because for example if you GRANT SELECT privilege on employee table to user1 using the WITH GRANT option, then user1 can GRANT SELECT privilege on employee table to another user, such as user2 etc. Later, if you REVOKE the SELECT privilege on employee from user1, 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}
```

Example:

REVOKE SELECT ON employee FROM user1;This commmand 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.

1) System privileges - This allows the user to CREATE, ALTER, or DROP database objects.
2) 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.

Few of the object privileges are listed below:

Object Privileges	Description
INSERT	allows users to insert rows into a table.
SELECT	allows users to select data from a database object.
UPDATE	allows user to update data in a table.
EXECUTE	allows user to execute a stored procedure or a function.

**~~~~**
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];

Example:

To create a role called "developer" with password as "pwd",the code will be as follows

```
CREATE ROLE testing
[IDENTIFIED BY pwd];
```

It's easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege direclty to every user. If a role is identified by a password, then, when you GRANT or REVOKE privileges to the role, you definetely 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;
```

For example: To drop a role called developer, you can write:

```
DROP ROLE testing;
```

Checking Roles Granted to a User

You can check which roles have been granted to a user by querying user_role_privs. A user who creates a role is also granted that role by default.

```
SQL> desc user_role_privs;
```

Checking System Privileges Granted to a Role

```
SQL> desc role_sys_privs;
```

Checking Object Privileges Granted to a Role

```
SQL> desc role_tab_privs;
```

****

Working with Aliases

- Aliases are especially be helpful for more readable, if the SQL statement is complex or the table or column names are long.
- An ALIAS is an alternate name given for any ORACLE OBJECT.
- Aliases in Oracle are of two types... (Column Alias & Table Alias)

Column Alias

Sometimes you want to change the column headers in the report. For this you can use column aliases in oracle. oracle has provided excellent object oriented techniques as it's robust database. It always good for you to practise and implement column aliases since it will make your code readable while using this columns.

So You can follow this to add column aliases to your sql queries.

1).Give a column alias name separated by space after the column name. you can use none or one or more column aliases as per your requirement. e.g is the query to use this kind of column aliases.

Example-1

Select last_name surname, first_name, commission_pct commission employee salary from employees

In the above query, the words in bold are column aliases.

2) If we observe the above query results we see all the aliases in all capital letters, some how if you do not want them and you want your own case Use double quotes ". Also if you want to keep space between words and you need it as alias use doble quote e.g.("Employee Salary"). Please follow the below query for this kind of situation. Please note that you can also use "AS" in the query for aliases

Example-2

Select last_name **"Surname"**, first_name AS name, commission_pct as commission, salary **"Employee Salary"**, Employee_id As ID from employees;

Table Alias

Table aliases give you the ability to clarify which table you are referring to in a query, and to do so with an abbreviated name or code. Use table aliases in all but trivial, single-table queries. Ensure that your table aliases are short, but intuitive. Use them consistently. Do not use them for the one or two ambiguous columns in your query which the compiler required you to clarify.

In the following examples, we've lengthened the table names in the SCOTT schema to illustrate the use of short aliases that don't duplicate the table name. Assume the company name is FYZXX, Inc.

```
SELECT ename  emp_name,
       job    job_name,
       dname dept_name,
       loc   location
  FROM fyzxx_employee   a,
       fyzxx_department b
 WHERE a.deptno = b.deptno
 ORDER BY b.deptno,
          a.sal DESC;
```

In this example, the aliases chosen are too short, not intuitive, and aren't used consistently. If they are going to be this short, at least use a letter that can be intuitively mapped to the table it represents, like "e" for the employee table, and "d" for the department table. However, queries are rarely this simple; you may be joining more than one table that could be interpreted as the "e" table. So you should use 2 to 5 characters for your table aliases, to ensure clarity, readability and maintainability, e.g.

```
SELECT emp.ename  emp_name,
       emp.job    job_name,
       dept.dname dept_name,
       dept.loc   location
  FROM fyzxx_employee   emp,
       fyzxx_department dept
 WHERE emp.deptno = dept.deptno
 ORDER BY dept.deptno,
          emp.sal DESC;
```

Some tools will automatically prepend every column name with the full name of the table as the table "alias." SQL like the following often shows up in tools that cater to Access and SQL-Server:

```
SELECT "FYZXX_EMPLOYEE"."ENAME" "EMP_NAME",
       "FYZXX_EMPLOYEE"."JOB" "JOB_NAME",
       "FYZXX_DEPARTMENT"."DNAME" "DEPT_NAME",
       "FYZXX_DEPARTMENT"."LOC"    "LOCATION"
  FROM "FYZXX_EMPLOYEE",
       "FYZXX_DEPARTMENT"
 WHERE "FYZXX_EMPLOYEE"."DEPTNO" = "FYZXX_DEPARTMENT"."DEPTNO"
 ORDER BY "FYZXX_DEPARTMENT"."DEPTNO",
          "FYZXX_EMPLOYEE"."SAL" DESC;
```

Do not allow this style in production code, as it is nearly unreadable, much longer than it needs to be, and more difficult to maintain.

Finally, when unnesting PL/SQL collections or nested table columns, aliases are very important. First, scalar collections have no column name. The single, implied field in a scalar collection can only be referred to in SQL using the pseudo-column COLUMN_VALUE. It is a best practice to give the pseudo-column a column alias. Second, in earlier versions of 9i, unless you used a table alias for the unnested collection, weird and seemingly random errors would pop up. This seems to be fixed in 10g, but it is still a good idea to alias whatever gets wrapped with the TABLE() operator. Study the following example:

```
CREATE OR REPLACE TYPE type_currency_tab AS TABLE OF NUMBER(12,4)
/
SET SERVEROUTPUT ON
DECLARE
   l_prices type_currency_tab := type_currency_tab(5.50, 10.95, 995.99);
BEGIN
   FOR lrec IN (SELECT curr.COLUMN_VALUE AS price
                  FROM TABLE(l_prices) curr
                 WHERE curr.COLUMN_VALUE > 100) LOOP
      dbms_output.put_line('Large Price: ' || lrec.price);
   END LOOP;
END;
/
```

LITERALS in Oracle

- A literal & a Constant value are Synonyms to one another and refer to a fixed data value.
- The Types of LITERALS recognized by oracle are:
  TEXT Literals,
  INTEGER Literals,
  NUMBER Literals,
  INTERVAL Literals.

TEXT Literals

It specifies a TEXT or CHARACTER literal.

It is used to specify values whenever 'TEXT' or CHAR appear in
Expression, Conditions, SQL Function, SQL Statements.

TEXT Literal should be enclosed in single quotes.

They have Properties of both CHAR anc VARCHAR2 data types.

A TEXT Literal can have a maximum length of 4000 bytes.

Example:

'Employee Information'
'Manager's Specification'

Using LITERAL Character Strings:

- A literal that is declared in a SELECT list can be a CHARACTER, a NUMBER, or a DATE.

- A literal is not a column name or a column Alias.
- A literal is printed for each row, that is retrieved by the SELECT Statement.
- Literal strings of free-form text can be included in the query as per the requiremnet.
- DATE and CHARACTER Literals must be enclosed within the SINGLE QUOTATION Marks.
- Literals increase the readability of the output.

Examples:

SQL> select ename ||’:’’ Month Salary = ‘|| Sal As Salaries from emp;

SQL> select ‘The Designation of ‘|| Ename||’ is ‘ || Job As Designation from emp;

SQL> select ‘The Annual Salary of ‘||ename||’ is’|| sal*12 as annual_Salary from emp;

SQL> select dname||’ Department is located at ‘ ||loc Locations from dept;

SQL> select ename ||’ Joined The Organization on ‘||Hiredate from emp;

SQL> select ename||’ works in department number ‘||deptno||’ as’||job from emp;

Applying CONCATENATION OPERATOR:

- The concatenation operator links columns to other columns, Arithmetic Expressions, or Constant Values.
- Columns on either side of the operator are combined to make a single outpuit column.
- The resultant column is traated as an CHARACTER EXPRESSION.
- The concatenation operator is represented in oracle by DOUBLE PIPE symbol (||).

Examples

SQL> SELECT empno||’ ‘||ename||’, Designation is ‘||job “Employees Information” from emp;

****

SQL Operators

An operator manipulates individual data items and returns a result. The data items are called *operands* or *arguments*. Operators are represented by special characters or by keywords.

For example, the multiplication operator is represented by an asterisk (*) and the operator that tests for nulls is represented by the keywords IS NULL.

There are two general classes of operators: Unary and Binary. Oracle Lite SQL also supports set operators.

Unary Operators

A unary operator uses only one operand. A unary operator typically appears with its operand in the following format: operator operand

Binary Operators

A binary operator uses two operands. A binary operator appears with its operands in the following format: operand1 operator operand2

Set Operators

Set operators combine the results of two queries into a single result. Set operators combine sets of rows returned by queries, instead of individual data items. All set operators have equal precedence. Oracle Lite supports the following set operators:

UNION
UNION ALL
INTERSECT
MINUS

Table Set Operators

Operator	Description	Example

UNION	Returns all distinct rows selected by either query.
	Syn:
	Select *From <tablename1>
	Union
	Select *from <tablename2>
		SELECT * FROM
		(SELECT ENAME FROM EMP
		WHERE JOB = 'CLERK'
		UNION
		SELECT ENAME FROM EMP
		WHERE JOB = 'ANALYST');

UNION ALL	Returns all rows selected by either query, including all duplicates.
	Syn:
	Select *From <tablename1>
	Unionall
	Select *from <tablename2>
		SELECT * FROM
		(SELECT SAL FROM EMP
		WHERE JOB = 'CLERK'
		UNIONALL
		SELECT SAL FROM EMP
		WHERE JOB = 'ANALYST');

Intersect	It is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement.
	Syn:
	Select *From <tablename1>
	intersect
	Select *from <tablename2>
		SELECT * FROM
		(SELECT ENAME FROM EMP
		WHERE JOB = 'CLERK'
		Intersect
		SELECT ENAME FROM EMP WHERE JOB = 'ANALYST');

Minus	Combines two SELECT statements and returns rows from the first SELECT statements that are not returned by the second SELECT statement. **Syn:** **Select *From <tablename1>** **minus** **Select *from <tablename2>**	**EX:** Select eno,ename from emp Minus Select eno,ename from emp;

As there are many sql operators mainly used for sorting the data as per the user's specified query based on the particular information by using sql operators.

Namely - Comparison Operators, Logical operators, Special operators, Negation operators, and Arithmetic operators These operators are used mainly in the WHERE clause, HAVING clause to filter the data to be selected. we can use operators with some of the commands like.. create, alter, delete, update, insert.

**~~~~~~~~~~~~~~~~**

Comparison Operators

**~~~~~~~~~~~~~~~~**

Comparison operators are used to compare the column data with specific values in a condition.
Comparison Operators are also used along with the SELECT statement to filter data based on specific conditions.

The below table describes each comparison operator.

Comparison Operators	Description
=	equal to
<>, !=, ^=	is not equal to
<	less than
>	greater than
>=	greater than or equal to
<=	less than or equal to

~~~~~~~~~~~~~~~

SQL Logical Operators

~~~~~~~~~~~~~~~

There are three Logical Operators namely, AND, OR, and NOT. These operators compare two conditions at a time to determine whether a row can be selected for the output. When retrieving data using a SELECT statement, you can use logical operators in the WHERE clause, which allows you to combine more than one condition.

Logical Operators	Description
OR	For the row to be selected at least one of the conditions must be true.
AND	For a row to be selected all the specified conditions must be true.
NOT	For a row to be selected the specified condition must be false.

"OR" Logical Operator:

If you want to select rows that satisfy at least one of the given conditions, you can use the logical operator, OR.

For example: if you want to find the names of students who are studying either Maths or Science, the query would be like,

```
SELECT first_name, last_name, subject
FROM student_details
WHERE subject = 'Maths' OR subject = 'Science'
```

The output would be something like,

```
first_name  last_name  subject
----------  ---------  -------
Abdul       Raheem     Maths
Anil        Kumar      Maths
Peter       Roy        Science
Stephen     Fleming    Science
```

The following table describes how logical "OR" operator selects a row.

Column1 Satisfied?	Column2 Satisfied?	Row Selected
YES	YES	YES
YES	NO	YES
NO	YES	YES
NO	NO	NO

"AND" Logical Operator:

If you want to select rows that must satisfy all the given conditions, you can use the logical operator, AND.

For Example: To find the names of the students between the age 10 to 15 years, the query would be like:

```
SELECT first_name, last_name, age
FROM student_details
WHERE age >= 10 AND age <= 15;
```

The output would be something like,

```
first_name    last_name    age
-------------  -------------  ------
Rahul         Sharma       10
Anajali       Bhagwat      12
Shekar        Gowda        15
```

The following table describes how logical "AND" operator selects a row.

Column1 Satisfied?	Column2 Satisfied?	Row Selected
YES	YES	YES
YES	NO	NO
NO	YES	NO
NO	NO	NO

"NOT" Logical Operator:

If you want to find rows that do not satisfy a condition, you can use the logical operator, NOT. NOT results in the reverse of a condition. That is, if a condition is satisfied, then the row is not returned.

For example: If you want to find out the names of the students who do not play football, the query would be like:

```
SELECT first_name, last_name, games
FROM student_details
WHERE NOT games = 'Football'
```

The output would be something like,

```
first_name   last_name Games
---------------- -------------  -----------
Rahul        Sharma     Cricket
Stephen      Fleming    Cricket
Shekar       Gowda      Badminton
Priya        Chandra    Chess
```

The following table describes how logical "NOT" operator selects a row.

Column1 Satisfied?	NOT Column1 Satisfied?	Row Selected
YES	NO	NO

NO	YES	YES

~~~~~~~~~~~~~~~~~~~
Nested Logical Operators:
~~~~~~~~~~~~~~~~~~~

You can use multiple logical operators in an SQL statement. When you combine the logical operators in a SELECT statement, the order in which the statement is processed is

1) NOT
2) AND
3) OR

For example: If you want to select the names of the students who age is between 10 and 15 years, or those who do not play football, the

SELECT statement would be

SELECT first_name, last_name, age, games
FROM student_details
WHERE age >= 10 **AND** age <= 15
OR NOT games = 'Football'

The output would be something like,

```
first_name last_name Age    games
------------- ------------- -------- ------------
Abdul       Raheem    10     Cricket
Peter       Roy       15     Chess
```

In this case, the filter works as follows:

Condition 1: All the students you do not play football are selected.
Condition 2: All the students whose are aged between 10 and 15 are selected.
Condition 3: Finally the result is, the rows which satisfy atleast one of the above conditions is returned.

NOTE:
The order in which you phrase the condition is important, if the order changes you are likely to get a different result.

~~~~~~~~~~~~~~~~~~~~
SQL Comparison Keywords
~~~~~~~~~~~~~~~~~~~~

There are other comparison keywords available in sql which are used to enhance the search capabilities of a sql query. They are "IN", "BETWEEN...AND", "IS NULL", "LIKE".

Comparision Operators	Description
LIKE	column value is similar to specified character(s).
IN	column value is equal to any one of a specified set of values.
BETWEEN...AND	column value is between two values, including the end values specified in the range.
IS NULL	column value does not exist.

SQL LIKE Operator

The LIKE operator is used to list all rows in a table whose column values match a specified pattern. It is useful when you want to search rows to match a specific pattern, or when you do not know the entire value. For this purpose we use a wildcard character '%'.

For example: To select all the students whose name begins with 'S'

```
SELECT first_name, last_name
FROM student_details
WHERE first_name LIKE 'S%';
```

The output would be similar to:

```
first_name      last_name
-------------   -------------
Stephen         Fleming
Shekar          Gowda
```

The above select statement searches for all the rows where the first letter of the column first_name is 'S' and rest of the letters in the name can be any character.

There is another wildcard character you can use with LIKE operator. It is the underscore character, ' _ ' . In a search string, the underscore signifies a single character.

For example: to display all the names with 'a' second character,

```
SELECT first_name, last_name
FROM student_details
WHERE first_name LIKE '_a%';
```

The output would be similar to:

```
first_name      last_name
-------------   -------------
Rahul           Sharma
```

NOTE:

Each underscore act as a placeholder for only one character. So you can use more than one underscore. Eg: ' __i% '-this has two underscores towards the left, 'S__j%' - this has two underscores between character 'S' and 'i'.

SQL BETWEEN ... AND Operator

The operator BETWEEN and AND, are used to compare data for a range of values.

For Example: to find the names of the students between age 10 to 15 years, the query would be like,

```
SELECT first_name, last_name, age
FROM student_details
WHERE age BETWEEN 10 AND 15;
```

The output would be similar to:

```
first_name      last_name       Age
-------------   -------------   ------
Rahul           Sharma          10
```

```
Anajali     Bhagwat     12
Shekar      Gowda       15
```

SQL IN Operator

The IN operator is used when you want to compare a column with more than one value. It is similar to an OR condition.

For example: If you want to find the names of students who are studying either Maths or Science, the query would be like,

```
SELECT first_name, last_name, subject
FROM student_details
WHERE subject IN ('Maths', 'Science');
```

The output would be similar to:

```
first_name   last_name   Subject
-------------  -------------  ----------
Anajali      Bhagwat     Maths
Shekar       Gowda       Maths
Rahul        Sharma      Science
Stephen      Fleming     Science
```

You can include more subjects in the list **like** ('maths','science','history')

NOTE:

The data used to compare is case sensitive.

**~~~~~~~~~~~~~~~~**
SQL IS NULL Operator
**~~~~~~~~~~~~~~~~**

A column value is NULL if it does not exist. The IS NULL operator is used to display all the rows for columns that do not have a value.

For Example: If you want to find the names of students who do not participate in any games, the query would be as given below

```
SELECT first_name, last_name
FROM student_details
WHERE games IS NULL
```

There would be no output as we have every student participate in a game in the table student_details, else the names of the students who do not participate in any games would be displayed.

**~~~~~~~~~~~**
SQL ORDER BY
**~~~~~~~~~~~**

The ORDER BY clause is used in a SELECT statement to sort results either in ascending or descending order. Oracle sorts query results in ascending order by default.

Syntax for using SQL ORDER BY clause to sort data is:

SELECT column-list
FROM table_name [WHERE condition]
[ORDER BY column1 [, column2, .. columnN] [DESC]];

database table "employee";

id	name	Dept	age	salary	location
100	Ramesh	Electrical	24	25000	Bangalore
101	Hrithik	Electronics	28	35000	Bangalore
102	Harsha	Aeronautics	28	35000	Mysore
103	Soumya	Electronics	22	20000	Bangalore
104	Priya	InfoTech	25	30000	Mangalore

For Example: If you want to sort the employee table by salary of the employee, the sql query would be.

SELECT name, salary FROM employee **ORDER BY** salary;

The output would be like

```
Name           Salary
----------     ----------
Soumya         20000
Ramesh         25000
Priya          30000
Hrithik        35000
Harsha         35000
```

The query first sorts the result according to name and then displays it.
You can also use more than one column in the ORDER BY clause.

If you want to sort the employee table by the name and salary, the query would be like,

SELECT name, salary FROM employee **ORDER BY** name, salary;

The output would be like:

```
Name          Salary
------------- --------
Soumya        20000
Ramesh        25000
Priya         30000
Harsha        35000
Hrithik       35000
```

NOTE:
The columns specified in ORDER BY clause should be one of the columns selected in the SELECT column list.

You can represent the columns in the ORDER BY clause by specifying the position of a column in the SELECT list, instead of writing the column name.

The above query can also be written as given below,

SELECT name, salary FROM employee **ORDER BY** 1, 2;

By default, the ORDER BY Clause sorts data in ascending order. If you want to sort the data in descending order, you must explicitly specify it as shown below.

```
SELECT name, salary
FROM employee
ORDER BY name, salary DESC;
```

The above query sorts only the column 'salary' in descending order and the column 'name' by ascending order.

If you want to select both name and salary in descending order, the query would be as given below.

```
SELECT name, salary
FROM employee
ORDER BY name DESC, salary DESC;
```

How to use expressions in the ORDER BY Clause?

Expressions in the ORDER BY clause of a SELECT statement.

For example:

If you want to display employee name, current salary, and a 20% increase in the salary for only those employees for whom the percentage increase in salary is greater than 30000 and in descending order of the increased price, the SELECT statement can be written as shown below

```
SELECT name, salary, salary*1.2 AS new_salary
FROM employee
WHERE salary*1.2 > 30000
ORDER BY new_salary DESC;
```

The output for the above query is as follows.

```
Name        salary      new_salary
----------  ----------  -------------
Hrithik     35000       37000
Harsha      35000       37000
Priya       30000       36000
```

NOTE:

Aliases defined in the SELECT Statement can be used in ORDER BY Clause.

**~~~~~~~~~~~~~~~**

SQL GROUP BY Clause

**~~~~~~~~~~~~~~~**

The SQL GROUP BY Clause is used along with the group functions to retrieve data grouped according to one or more columns.

For Example: If you want to know the total amount of salary spent on each department, the query would be:

```
SELECT dept,SUM(salary)
FROM employee
GROUP BY dept;
```

The output would be like:

```
Dept              Salary
----------------  --------------
Electrical        25000
Electronics       55000
```

```
Aeronautics    35000
InfoTech       30000
```

NOTE:

The group by clause should contain all the columns in the select list expect those used along with the group functions.

SELECT location,dept,SUM(salary)
FROM employee
GROUP BY location,dept;

The output would be like:

```
Location       Dept             Salary
-------------  ---------------  -----------
Bangalore      Electrical       25000
Bangalore      Electronics      55000
Mysore         Aeronautics      35000
Mangalore      InfoTech         30000
```

~~~~~~~~~~~~~~
SQL HAVING Clause
~~~~~~~~~~~~~~

Having clause is used to filter data based on the group functions. This is similar to WHERE condition but is used with group functions. Group functions cannot be used in WHERE Clause but can be used in HAVING clause.

For Example: If you want to select the department that has total salary paid for its employees more than 25000, the sql query would be like;

SELECT dept,SUM(salary)
FROM employee
GROUP BY dept
HAVING SUM(salary)>25000

The output would be like:

```
Dept          Salary
------------- -------------
Electronics   55000
Aeronautics   35000
InfoTech      30000
```

When WHERE, GROUP BY and HAVING clauses are used together in a SELECT statement, the WHERE clause is processed first, then the rows that are returned after the WHERE clause is executed are grouped based on the GROUP BY clause. Finally, any conditions on the group functions in the HAVING clause are applied to the grouped rows before the final output is displayed.

~~~~~~~~
SQL Joins
~~~~~~~~

SQL Joins are used to relate information in different tables. A Join condition is a part of the sql query that retrieves rows from two or more tables. A SQL Join condition is used in the SQL WHERE Clause of select, update, delete statements.

The Syntax for joining two tables is:

```
SELECT col1, col2,col3. . . .
FROM table_name1,table_name2
WHERE table_name1.col2 = table_name2.col1;
```

If a sql join condition is omitted or if it is invalid the join operation will result in a Cartesian product. The Cartesian product returns a number of rows equal to the product of all rows in all the tables being joined. For example, if the first table has 20 rows and the second table has 10 rows, the result will be 20 * 10, or 200 rows. This query takes a long time to execute.

Lets use the below two tables to explain the sql join conditions.

database table "product";

product_id	product_name	supplier_name	unit_price
100	Camera	Nikon	300
101	Television	Onida	100
102	Refrigerator	Vediocon	150
103	Ipod	Apple	75
104	Mobile	Nokia	50

database table "order_items";

order_id	product_id	total_units	customer
5100	104	30	Infosys
5101	102	5	Satyam
5102	103	25	Wipro
5103	101	10	TCS

SQL Joins can be classified into Equi join and Non Equi join.

1) SQL Equi joins	**2) SQL Non equi joins**
It is a simple sql join condition which uses the equal sign as the comparison operator. Two types of equi joins are SQL Outer join and SQL Inner join. **For example**: You can get the information about a customer who purchased a product and the quantity of product.	It is a sql join condition which makes use of some comparison operator other than the equal sign like >, <, >=, <=

1) SQL Equi Joins:

An equi-join is further classified into two categories:

a) SQL Inner Join
b) SQL Outer Join

a) SQL Inner Join:

All the rows returned by the sql query satisfy the sql join condition specified.

For example: If you want to display the product information for each order the query will be as given below. Since you are retrieving the data from two tables, you need to identify the common column between these two tables, which is theproduct_id.

The query for this type of sql joins would be like,

```
SELECT order_id, product_name, unit_price, supplier_name, total_units
FROM product, order_items
WHERE order_items.product_id = product.product_id;
```

The columns must be referenced by the table name in the join condition, because product_id is a column in both the tables and needs a way to be identified. This avoids ambiguity in using the columns in the SQL SELECT statement.

The number of join conditions is (n-1), if there are more than two tables joined in a query where 'n' is the number of tables involved. The rule must be true to avoid Cartesian product.

We can also use aliases to reference the column name, then the above query would be like,

```
SELECT o.order_id, p.product_name, p.unit_price, p.supplier_name, o.total_units
FROM product p, order_items o
WHERE o.product_id = p.product_id;
```

b) SQL Outer Join:

This sql join condition returns all rows from both tables which satisfy the join condition along with rows which do not satisfy the join condition from one of the tables. The sql outer join operator in Oracle is (+) and is used on one side of the join condition only.

The syntax differs for different RDBMS implementation. Few of them represent the join conditions as "sql left outer join", "sql right outer join".

If you want to display all the product data along with order items data, with null values displayed for order items if a product has no order item, the sql query for outer join would be as shown below:

```
SELECT p.product_id, p.product_name, o.order_id, o.total_units
FROM order_items o, product p
WHERE o.product_id (+) = p.product_id;
```

The output would be like,

```
 product_id  product_name  order_id  total_units
-------------  -------------  ---------- -------------
    100          Camera
    101        Television     5103         10
    102       Refrigerator    5101          5
    103           Ipod        5102         25
    104          Mobile       5100         30
```

NOTE:

If the (+) operator is used in the left side of the join condition it is equivalent to left outer join. If used on the right side of the join condition it is equivalent to right outer join.

**~~~~~~~~~**
SQL Self Join
**~~~~~~~~~**

A Self Join is a type of sql join which is used to join a table to itself, particularly when the table has a FOREIGN KEY that references its own PRIMARY KEY. It is necessary to ensure that the join statement defines an alias for both copies of the table to avoid column ambiguity.

The below query is an example of a self join,

```
SELECT a.sales_person_id, a.name, a.manager_id, b.sales_person_id, b.name
FROM sales_person a, sales_person b
WHERE a.manager_id = b.sales_person_id;
```

Revising in Detail -- 2) SQL Non Equi Join:

A Non Equi Join is a SQL Join whose condition is established using all comparison operators except the equal (=) operator. Like >=, <=, <, >

For example: If you want to find the names of students who are not studying either Economics, the sql query would be like, (lets use student_details table defined earlier.)

```
SELECT first_name, last_name, subject
FROM student_details
WHERE subject != 'Economics'
```

The output would be something like,

```
first_name  last_name   subject
----------- ----------- -----------
Anajali     Bhagwat     Maths
Shekar      Gowda       Maths
Rahul       Sharma      Science
Stephen     Fleming     Science
```

****

Oracle Built in Functions

There are two types of functions in Oracle.

1) Single Row Functions: Single row or Scalar functions return a value for every row that is processed in a query.

^^
2) Group Functions: These functions group the rows of data based on the values returned by the query. This is discussed in SQL GROUP Functions. The group functions are used to calculate aggregate values like total or average, which return just one total or one average value after processing a group of rows.
^^

1. Single Row Functions

There are four types of single row functions. They are:

1) **Numeric Functions**: These are functions that accept numeric input and return numeric values.

2) **Character or Text Functions**: These are functions that accept character input and can return both character and number values.

3) **Date Functions**: These are functions that take values that are of datatype DATE as input and return values of datatype DATE, except for the MONTHS_BETWEEN function, which returns a number.

4) **Conversion Functions**: These are functions that help us to convert a value in one form to another form. For

Example:

a null value into an actual value, or a value from one datatype to another datatype like NVL, TO_CHAR, TO_NUMBER, TO_DATE etc.

You can combine more than one function together in an expression. This is known as nesting of functions.

**~~~~~~~~~~~~~~~**
1) Numeric Functions:
**~~~~~~~~~~~~~~~**

Numeric functions are used to perform operations on numbers. They accept numeric values as input and return numeric values as output. Few of the Numeric functions are:

Function Name	Return Value
ABS (x)	Absolute value of the number 'x'
CEIL (x)	Integer value that is Greater than or equal to the number 'x'
FLOOR (x)	Integer value that is Less than or equal to the number 'x'
TRUNC (x, y)	Truncates value of number 'x' up to 'y' decimal places
ROUND (x, y)	Rounded off value of the number 'x' up to the number 'y' decimal places

The following examples explains the usage of the above numeric functions

Function Name	Examples	Return Value
ABS (x)	ABS (1) ABS (-1)	1 -1
CEIL (x)	CEIL (2.83) CEIL (2.49) CEIL (-1.6)	3 3 -1
FLOOR (x)	FLOOR (2.83) FLOOR (2.49) FLOOR (-1.6)	2 2 -2
TRUNC (x, y)	ROUND (125.456, 1) ROUND (125.456, 0) ROUND (124.456, -1)	125.4 125 120
ROUND (x, y)	TRUNC (140.234, 2) TRUNC (-54, 1) TRUNC (5.7) TRUNC (142, -1)	140.23 54 5 140

These functions can be used on database columns.

For Example:

Let's consider the product table used in sql joins. We can use ROUND to round off the unit_price to the nearest integer, if any product has prices in fraction.

SELECT ROUND (unit_price) FROM product;

**~~~~~~~~~~~~~~~~~~~~~~**
2) Character or Text Functions:
**~~~~~~~~~~~~~~~~~~~~~~**

Character or text functions are used to manipulate text strings. They accept strings or characters as input and can return both character and number values as output.

Few of the character or text functions are as given below:

Function Name	Return Value
LOWER (string_value)	All the letters in 'string_value' is converted to lowercase.
UPPER (string_value)	All the letters in 'string_value' is converted to uppercase.
INITCAP (string_value)	All the letters in 'string_value' is converted to mixed case.
LTRIM (string_value, trim_text)	All occurrences of 'trim_text' is removed from the left of 'string_value'.
RTRIM (string_value, trim_text)	All occurrences of 'trim_text' is removed from the right of'string_value' .
TRIM (trim_text FROM string_value)	All occurrences of 'trim_text' from the left and right of 'string_value' ,'trim_text' can also be only one character long .
SUBSTR (string_value, m, n)	Returns 'n' number of characters from'string_value' starting from the 'm'position.
LENGTH (string_value)	Number of characters in 'string_value'in returned.
LPAD (string_value, n, pad_value)	Returns 'string_value' left-padded with'pad_value' . The length of the whole string will be of 'n' characters.

RPAD (string_value, n, pad_value)	Returns 'string_value' right-padded with 'pad_value' . The length of the whole string will be of 'n' characters.

For Example:

We can use the above UPPER() text function with the column value as follows.

SELECT UPPER (product_name) FROM product;

The following examples explains the usage of the above character or text functions

Function Name	Examples	Return Value
LOWER(string_value)	LOWER('Good Morning')	good morning
UPPER(string_value)	UPPER('Good Morning')	GOOD MORNING
INITCAP(string_value)	INITCAP('GOOD MORNING')	Good Morning
LTRIM(string_value, trim_text)	LTRIM ('Good Morning', 'Good)	Morning
RTRIM (string_value, trim_text)	RTRIM ('Good Morning', ' Morning')	Good
TRIM (trim_text FROM string_value)	TRIM ('o' FROM 'Good Morning')	Gd Mrning
SUBSTR (string_value, m, n)	SUBSTR ('Good Morning', 6, 7)	Morning
LENGTH (string_value)	LENGTH ('Good Morning')	12
LPAD (string_value, n, pad_value)	LPAD ('Good', 6, '*')	**Good
RPAD (string_value, n, pad_value)	RPAD ('Good', 6, '*')	Good**

~~~~~~~~~~~~~
3) Date Functions:
~~~~~~~~~~~~~

These are functions that take values that are of datatype DATE as input and return values of datatypes DATE, except for the MONTHS_BETWEEN function, which returns a number as output.

Few date functions are as given below.

Function Name	Return Value
ADD_MONTHS (date, n)	Returns a date value after adding 'n'months to the date 'x'.
MONTHS_BETWEEN (x1, x2)	Returns the number of months between dates x1 and x2.
ROUND (x, date_format)	Returns the date 'x' rounded off to the nearest century, year, month, date, hour, minute, or second as specified by the 'date_format'.
TRUNC (x, date_format)	Returns the date 'x' lesser than or equal to the nearest century, year, month, date, hour, minute, or second as specified by the 'date_format'.
NEXT_DAY (x, week_day)	Returns the next date of the 'week_day'on or after the date 'x' occurs.
LAST_DAY (x)	It is used to determine the number of days remaining in a month from the date 'x' specified.
SYSDATE	Returns the systems current date and time.
NEW_TIME (x, zone1, zone2)	Returns the date and time in zone2 if date 'x' represents the time in zone1.

The below table provides the examples for the above functions

Function Name	Examples	Return Value
ADD_MONTHS ( )	ADD_MONTHS ('16-Sep-81', 3)	16-Dec-81
MONTHS_BETWEEN( )	MONTHS_BETWEEN ('16-Sep-81', '16-Dec-81')	3
NEXT_DAY( )	NEXT_DAY ('01-Jun-08', 'Wednesday')	04-JUN-08
LAST_DAY( )	LAST_DAY ('01-Jun-08')	30-Jun-08
NEW_TIME( )	NEW_TIME ('01-Jun-08', 'IST', 'EST')	31-May-08

~~~~~~~~~~~~~~~~~
4) Conversion Functions:
~~~~~~~~~~~~~~~~~

These are functions that help us to convert a value in one form to another form.

EXAMPLE:

A null value into an actual value, or a value from one datatype to another datatype like NVL, TO_CHAR, TO_NUMBER, TO_DATE.

Few of the conversion functions available in oracle are:

Function Name	Return Value
TO_CHAR (x [,y])	Converts Numeric and Date values to a character string value. It cannot be used for calculations since it is a string value.
TO_DATE (x [, date_format])	Converts a valid Numeric and Character values to a Date value. Date is formatted to the format specified by 'date_format'.
NVL (x, y)	If 'x' is NULL, replace it with 'y'. 'x' and 'y'must be of the same datatype.
DECODE (a, b, c, d, e, default_value)	Checks the value of 'a', if a = b, then returns'c'. If a = d, then returns 'e'. Else, returnsdefault_value.

The below table provides the examples for the above functions

Function Name	Examples	Return Value
TO_CHAR ()	TO_CHAR (3000, '$9999') TO_CHAR (SYSDATE, 'Day, Month YYYY')	$3000 Monday, June 2008
TO_DATE ()	TO_DATE ('01-Jun-08')	01-Jun-08
NVL ()	NVL (null, 1)	1

More Examples

```
Select eno,ename,hiredate from emp;
Select eno,ename, TO_CHAR(HIREDATE,’DD-MM-YY’) from emp;
Select eno,ename, TO_CHAR(HIREDATE,’DD-MM-YYYY’) from emp;
Select SYSDATE from dual; // 18-JUN-12
Select TO_CHAR(SYSDATE,’DD-MONTH-YY’) from dual; // 18-JUN-12
Select TO_CHAR(SYSDATE,’DAY’) from dual; // Monday
Select TO_CHAR(SYSDATE,’YYYY’) from dual; // 2012
Select TO_CHAR(SYSDATE,’MM’) from dual; // 06
Select TO_CHAR(SYSDATE,’DDD’) from dual; // 170---
Select TO_CHAR(SYSDATE,’DD’) from dual; // 18
Select TO_CHAR(SYSDATE,’MON’) from dual; // MONDAY
Select TO_CHAR(SYSDATE,’DY’) from dual; // mon
Select TO_CHAR(SYSDATE,’DD-MM-YY HH:MI:SS’) from dual; //18-06-12 12:40:44
Select * from emp where TO_CHAR(HIREDATE,’YYYY’) = ‘1981’;
Select * from emp where TO_CHAR(HIREDATE,’YYYY’) = ‘1980’;
Select * from emp where TO_CHAR(HIREDATE,’MON’) = ‘DEC’;
```

TO_NUMBER:

```
Ex: Select TO_NUMBER(LTRIM(‘$1400’,’$’)) + 10 from dual; // 1410
```

TO_DATE: This function is used to convert character values to data value.

```
SELECT TO_CHAR(ADD_MONTHS(TO_DATE('01-JAN-2005 19:15:26','DD-MON-YYYY HH24:MI:SS'), 2), 'DD-MON-YYYY HH24:MI:SS') FROM dual;
```

ADD_MONTHS

```
Select ADD_MONTH(’11-JAN-05’,2) from dual;
Select ADD_MONTH((’11-JANUARY-2005 11:45 A.M.’,’DD-MONTH-YYYY  HH:MI A.M’),2) from dual;
Select ADD_MOONTHS(TO_DATE(’11-01-2005 11:45 A.M.’,  ‘DD-MM-YYYY HH:MI A.M.’),2) from dual;
                                                                        //11-MAR-2015
```

Implicit convertion:

```
Ex: Select * from emp where DEPTNO = ‘10’;
    Select sum(sal) from emp where deptno = ‘10’; //8750
    Select sum(sal) from emp where deptno = ‘20’; //10875
    Select sum(sal) from emp where deptno = ‘30’; // 9400
```

****

~~~~~~~~~~~~~~~~~~
2. SQL GROUP Functions
~~~~~~~~~~~~~~~~~~

Group functions are built-in SQL functions that operate on groups of rows and return one value for the entire group. These functions are: **COUNT, MAX, MIN, AVG, SUM, DISTINCT**

SQL COUNT ():

This function returns the number of rows in the table that satisfies the condition specified in the WHERE condition. If the WHERE condition is not specified, then the query returns the total number of rows in the table.

For Example: If you want the number of employees in a particular department, the query would be:

```
SELECT COUNT(*) FROM employee
WHERE dept = 'Electronics';
```

The output would be '2' rows.

If you want the total number of employees in all the department, the query would take the form:

```
SELECT COUNT(*) FROM employee;
```

The output would be '5' rows.

SQL DISTINCT():

This function is used to select the distinct rows.

For Example: If you want to select all distinct department names from employee table, the query would be:

```
SELECT DISTINCT dept FROM employee;
```

To get the count of employees with unique name, the query would be:

```
SELECT COUNT (DISTINCT name) FROM employee;
```

SQL MAX():

This function is used to get the maximum value from a column.

To get the maximum salary drawn by an employee, the query would be:

```
SELECT MAX (salary) FROM employee;
```

SQL MIN():

This function is used to get the minimum value from a column.

To get the minimum salary drawn by an employee, he query would be:

```
SELECT MIN (salary) FROM employee;
```

SQL AVG():

This function is used to get the average value of a numeric column.

To get the average salary, the query would be

```
SELECT AVG (salary) FROM employee;
```

SQL SUM():

This function is used to get the sum of a numeric column

To get the total salary given out to the employees,

SELECT SUM(salary) FROM employee;

GROUP Functions

We have seen before arithmetic functions which operated on numerical value and returned a single value for each single row taken as input. But there may be occasions where one might require several rows to be grouped and the result of the grouped rows might be needed. To handle such situations GROUP functions in SQL helps. There are number of group functions available in SQL. Let us see some of them:

Average:
To calculate the average of a group of values SQL defines the function AVG (column name)

Counting number of values:
If we want to count the number of values in a particular column then we can use the SQL function COUNT (column name)

Maximum:
To find the maximum among the column values SQL function MAX (column name) can be used

Minimum:
To find the minimum among the column values SQL function MIN (column name) can be used

There are many more SQL commands and terminologies in SQL but the above gives only an overview of the basics of SQL.

Group By clause: Group By clause is used to divide rows into several group. So that we can apply group function on each group.

Ex: Select deptno, sum(sal) from emp Group By deptno;

```
Deptno Sal
30          9400
20          10875
10          8750
```

Select deptno,min(sal),max(sal) from emp Group By deptno;

```
Deptno Min   Max
30          950    2850
20          800    3000
10          1300   5000
```

```
Select job, avg(sal) from emp Group By job;
Select job,count(*) from emp Group By job;
Select job,count(job) from emp Group By job;
Select Deptno, sum(Sal), min(Sal), max(Sal), avg(Sal), count(*) from emp Group By deptno;
```

We can use the combination of where clause and Group By clause.
First where clause is executed on the result of where clause Group By clause is apply.

```
Select deptno, sum(sal) from emp where deptno <> 10 Group By deptno;
Select deptno, job, sum(sal) from emp Group By deptno, job;
```

***Rule of group by clause**: All the column in the select of list should use group functions or should by included in group by clause.

```
Select deptno, sum(sal), ename from emp Group By deptno;
  Error: Not a Group By Expression
```

Having clause: (to use Group By clause)

Having clause is used to filter the output from Group By clause.

Ex: Select deptno, sum(sal) from emp Group By deptno having sum(sal) > 9000;

Deptno	Sum(sal)
30	9400
20	10875

*__Order By clause__:
Order By clause is used to arrange the rows in the table.

By default order by clause ascending order.

Null values are arranged in the last.

Examples

Select * from emp Order By sal;
Select * from emp Order By ename;
Select * from emp Order By HIREDATE; //Chronological order 1980…..1985
Select * from emp Order By eno;
Select * from emp Order By job,sal;
Select * from emp Order By sal DESC; //descending order by depending the query

Note: Order by clause should be the last change of the query.
Select deptno, sum(sal) from emp Group By deptno Having sum(sal) > 9000 Order By sum(sal) DESC;
Select deptno, sum(sal) from emp where ename <> 'King' Group By deptno;
Select deptno, sum(sal) from emp where ename <> 'King' Group By deptno Having sum(sal) > 9000;
Select deptno, sum(sal) from emp where ename <> 'King' Group By deptno Having sum(sal) > 9000 Order By sum(sal) DESC;

****

Database Schema Objects:

A **schema** is a collection of logical structures of data, or schema objects. A schema is owned by a database user and has the same name as that user. Each user owns a single schema. Schema objects can be created and manipulated with SQL and include the following types of objects:

Schema objects are the logical structures that directly refer to the database's data. Schema objects include structures like tables, views, and indexes. (There is no relationship between a tablespace and a schema. Objects in the same schema can be in different tablespaces, and a tablespace can hold objects from different schemas.)

Some of the most common schema objects are defined as :
→ Tables
→ Indexes
→ Views
→ Clusters
→ Synonyms
→ Sequence

~~~~~
## Tables:
~~~~~

A table is the basic unit of data storage in an oracle database as it consists of rows and columns and corresponds to a single record. When creating a table you give it a name and define the columns that belong to it. You can specify the width or the precision and scale for certain columns, and some of the columns can contain default values.

Dual Table
The dual table belongs to the sys schema and is created automatically when the data dictionary is created. Everything in oracle has to be in a table even results from a arithmetical expression have to be in a table, a query that retrieves those results needs a table to use and the dual table serves as a catchall table for those's expressions.

- dual is a table that contains a single row.
- The dual table has one VARCHAR2 column named dummy.

- dual contains a single row with the value X.

The structure of the dual table:
SQL> **DESCRIBE dual;**

```
 Name       Null?    Type
 DUMMY               VARCHAR2(1)
```

SQL> **SELECT * FROM dual;**

```
D
-
X
```

Do simple calculation by using dual:

SQL> select 123 * 456 from dual;

```
123*456
----------
56088
```

SQL> select sysdate from dual;

```
SYSDATE
----------
26-10-2009
```

**~~~~~~**

Indexes:

**~~~~~~**

Oracle indexes provides faster access to table rows by storing sorted values in specific columns and using those sorted values to easily lookup the associated table rows. This means that you can lookup data without having to look at more than a small fraction of the total rows within the table,

The trade off is that you get faster retrieval but inserting data is slower as the index needs to be updated, this slower inserting is bad news for OLTP type of databases but in a data warehouse type of database where inserting is at a minimal indexes can be used heavily.

Index Types

Unique/non-Unique	Based on unique column, something like national insurance number. It is better to use unique constraints on a tables columns which means oracle will create a unique index on those columns.
Primary/Secondary	Primary indexes are unique indexes that must always have a value, they cannot be NULL. Secondary indexes are other indexes in the same table that may not be unique.
Composite (concatenated)	Indexes that contain two or more columns from the same table, they are useful for enforcing uniqueness in a tables column where there's no single column that can uniquely identify a row.

Syntax to create Index:

```
CREATE INDEX index_name
ON table_name (column_name1,column_name2...);
```

Syntax to create SQL unique Index:

```
CREATE UNIQUE INDEX index_name
ON table_name (column_name1,column_name2...);
```

index_name is the name of the INDEX.

table_name is the name of the table to which the indexed column belongs.

column_name1, column_name2.. is the list of columns which make up the INDEX.

In Oracle there are two types of SQL index namely, implicit and explicit.

Implicit Indexes:

They are created when a column is explicity defined with PRIMARY KEY, UNIQUE KEY Constraint.

Explicit Indexes:

They are created using the "create index syntax.

NOTE:

1) Even though sql indexes are created to access the rows in the table quickly, they slow down DML operations like INSERT, UPDATE, DELETE on the table, because the indexes and tables both are updated along when a DML operation is performed. So use indexes only on columns which are used to search the table frequently.

2) Is is not required to create indexes on table which have less data.

3) In oracle database you can define up to sixteen (16) columns in an INDEX.

**~~~~**
Views:
**~~~~**

Views are customized presentations of data in one or more tables or other views. A view can also be considered a stored query. Views do not actually contain data. Rather, they derive their data from the tables on which they are based, referred to as the **base tables** of the views.

Like tables, views can be queried, updated, inserted into, and deleted from, with some restrictions. All operations performed on a view actually affect the base tables of the view.

A view is a virtual table consisting of a stored query, it contains **no data**. A view does not exist, basically its a definition defined within the data dictionary, lots of the DBA_ are views. There are a number of reason why we need views

→ Reduce complexity
→ Improve security
→ Increase convenience
→ Rename table columns
→ Customize the data for the users
→ Protect data integrity

Creating	create view test_view as select employee_id, first_name, last_name from employee where manger_id = 122;
Removing	drop view test_view;
Compile	alter view test compile;
Using a view	select * from test_view; Note: The sql statement defined by the view will be run.

The Syntax to create a sql view is

```
CREATE VIEW view_name
AS
SELECT column_list
FROM table_name [WHERE condition];
```

view_name is the name of the VIEW.

The SELECT statement is used to define the columns and rows that you want to display in the view.

For Example: to create a view on the product table the sql query would be like

```
CREATE VIEW view_product
AS
SELECT product_id,product_name
FROM product;
```

**~~~~~~~**

Synonyms:

**~~~~~~~**

Synonyms are aliases for any objects within the database (Objects like table, view, materialized view, sequence, procedure, function, package, type, Java class schema object, user-defined object type, or another synonym) they are used to make life easier for users, they hide the objects identity and can be either public or private. public are accessible to all users, and private synonyms are part on an individual users schema, so the user has to grant privilege right to other users. Synonyms can be created for

→ tables
→ views (including materialized views)
→ stored code (packages and procedures)

Synonyms can create both public and private synonyms.

A **public synonym** is owned by the special user group named PUBLIC and is accessible to every user in a database.

A **private synonym** is contained in the schema of a specific user and available only to the user and to grantees for the underlying object.

Synonyms themselves are not securable. When you grant object privileges on a synonym, you are really granting privileges on the underlying object, and the synonym is acting only as an alias for the object in the GRANT statement.

Creating Synonyms

Grant Permissions on Synonyms

→ To create a private synonym in your **own schema**, you must have the **CREATE SYNONYM** privilege.
→ To create a private synonym in **another user's schema**, you must have the **CREATE ANY SYNONYM** privilege.
→ To create a **public synonym**, you must have the **CREATE PUBLIC SYNONYM** system privilege.

Create a synonym using the CREATE SYNONYM statement. The underlying schema object need not exist, nor do you need privileges to access the object for the CREATE SYNONYM statement to succeed.

It is possible to refer a table with a different name and this done by using CREATE SYNONYM. It is possible to create synonym for tables as well as views.

SYNTAX:

```
CREATE synonym name FOR username.tablename;
```

Example:
The following statement creates a public synonym named public_emp on the emp table contained in the schema of jward:

CREATE PUBLIC SYNONYM public_emp FOR jward.emp

(When you create a synonym for a remote procedure or function, you must qualify the remote object with its schema name. Alternatively, you can create a local public synonym on the database where the remote object resides, in which case the database link must be included in all subsequent calls to the procedure or function).

Creating public	create public synonym employees for test.employees; Note: any user can use the synonym above
Creating private	create synonym addresses for hr.locations;
Removing	drop synonym addresses;

To check the synonyms information by using queries:

```
SQL>desc user_synonyms
SQL> desc dba_synonyms
```

```
SELECT * FROM user_synonyms
WHERE synonym_name LIKE 'abd%';
```

The following statement drops the private synonym named emp:

DROP SYNONYM emp;

The following statement drops the public synonym named public_emp:

DROP PUBLIC SYNONYM public_emp;

**~~~~~~~**

Sequences

~~~~~~~~~

Oracle has a automatic sequence generator that can produce a unique set of numbers, normally the numbers are used for primary keys. It is possible to cache the sequence numbers making the numbers ready and available in memory which improves performance, however if the system was to crash those sequences numbers are lost forever so be careful if the application requires no loss of sequence numbers.

There are a number of options that can be used when creating a sequence

Creating	create sequence employee_id_seq start with 1000 increment by 1 nomaxvalue nocycle; start with - you can choose any number you like to start with increment by - you can increment by any number nomaxvalue - just keep on going nocycle - you can recycle the list
Removing	drop sequence employee_id_seq;
caching (default 20)	alter sequence employee_id_seq cache 100; Note: remember you will lose the cache values during a system crash.

Using	select employee_id_seq.nextval from dual;
	select employee_id_seq.curval from dual;
	
	Note: If using "curval" in you get error message “is not yet defined in this session” must use nextval first.

Useful Views	
~~~~~~~~~

DBA_SEQUENCES	describes all sequences in the database
DBA_CATALOG	lists all indexes, tables, views, clusters, synonyms, and sequences in the database

~~~~~~ Clusters ~~~~~~

A **cluster** is a schema object that contains data from one or more tables, all of which have one or more columns in common. Oracle Database stores together all the rows from all the tables that share the same cluster key.

For information on existing clusters, query the USER_CLUSTERS, ALL_CLUSTERS, and DBA_CLUSTERS data dictionary views.

To create a cluster in your own schema, you must have CREATE CLUSTER system privilege. To create a cluster in another user's schema, you must have CREATE ANY CLUSTER system privilege. Also, the owner of the schema to contain the cluster must have either space quota on the tablespace containing the cluster or the UNLIMITED TABLESPACE system privilege.

****

SQL Subquery concepts

Subquery or Inner query or Nested query is a query in a query. A subquery is usually added in the WHERE Clause of the sql statement. Most of the time, a subquery is used when you know how to search for a value using a SELECT statement, but do not know the exact value.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries are an alternate way of returning data from multiple tables.

Subqueries can be used with the following sql statements along with the comparision operators like =, <, >, >=, <= etc.

SELECT
INSERT
UPDATE
DELETE

For Example:

1) Usually, a subquery should return only one record, but sometimes it can also return multiple records when used with operators like IN, NOT IN in the where clause. The query would be like,

```
SELECT first_name, last_name, subject
FROM student_details
WHERE games NOT IN ('Cricket', 'Football');
```

The output would be similar to:

```
 first_name   last_name    subject
------------- -------------  ----------
   Shekar       Gowda     Badminton
    Priya       Chandra      Chess
```

2) Lets consider the student_details table which we have used earlier. If you know the name of the students who are studying science subject, you can get their id's by using this query below,

```
SELECT id, first_name
FROM student_details
WHERE first_name IN ('Rahul', 'Stephen');
```

But, if you do not know their names, then to get their id's you need to write the query in this manner,

```
SELECT id, first_name
FROM student_details
WHERE first_name IN (SELECT first_name
FROM student_details
WHERE subject= 'Science');
```

Output:

```
   id     first_name
 -------- -------------
   100       Rahul
   102      Stephen
```

In the above sql statement, first the inner query is processed first and then the outer query is processed.

3) Subquery can be used with INSERT statement to add rows of data from one or more tables to another table. Lets try to group all the students who study Maths in a table 'maths_group'.

```
INSERT INTO maths_group(id, name)
SELECT id, first_name || ' ' || last_name
FROM student_details WHERE subject= 'Maths'
```

4) A subquery can be used in the SELECT statement as follows. Lets use the product and order_items table defined in the sql_joins section.

```
select p.product_name, p.supplier_name, (select order_id from order_items where product_id = 101) as order_id
from product p where p.product_id = 101
```

```
  product_name   supplier_name   order_id
 ------------------ ------------------ ----------
    Television          Onida          5103
```

Correlated Subquery

A query is called correlated subquery when both the inner query and the outer query are interdependent. For every row processed by the inner query, the outer query is processed as well. The inner query depends on the outer query before it can be processed.

```
SELECT p.product_name FROM product p
WHERE p.product_id = (SELECT o.product_id FROM order_items o
WHERE o.product_id = p.product_id);
```

NOTE:
1) You can nest as many queries you want but it is recommended not to nest more than 16 subqueries in oracle.
2) If a subquery is not dependent on the outer query it is called a non-correlated subquery.

DCL commands are used to enforce database security in a multiple user database environment. Two types of DCL commands are GRANT and REVOTE. Only Database Administrator's or owner's of the database object can provide/remove privileges on a databse object.

****

Review - More Concept about Sub-Queries

There are basically three types of subqueries are:

- ✓ Single Row Subqueries
- ✓ Multiple Row Subqueries
- ✓ Multiple Column Subqueries
- ✓ Things to Remember:

• Subqueries are queries nested inside other queries, marked off with parentheses.
• The result of inner query will pass to outer query for the preparation of final result.
• ORDER BY clause is not supported for Nested Queries.
• You cannot use Between Operator.
• Subqueries will always return only a single value for the outer query.
A sub query must be put in the right hand of the comparison operator.
A query can contain more than one sub-query.

**Syntax: SELECT <column, ..>
FROM <table>
WHERE expression operator
(SELECT <column, ...>
FROM <table>
WHERE <condition)
Single - Row Subqueries**

The single-row subquery returns one row. A special case is the scalar subquery, which returns a single row with one column. Scalar subqueries are acceptable (and often very useful) in virtually any situation where you could use a literal value, a constant, or an expression.

The single row query uses any operator in the query .i.e. (=, <=, >= <>, <, >). If any of the operators in the preceding table are used with a subquery that returns more than one row, the query will fail.

• Suppose you want to find out the ename, job, sal of the employees whose salaries are less than that of an employee whose empno= 7369 from obviously EMP table. Now you need to perform two queries in order to get the desired result. In the first query:

1) We will find out the salary of the employee whose empno=7369. The query is as under:

```
SQL> SELECT SAL FROM EMP WHERE EMPNO=7876;
```

After execution it will return the salary of a given empno and it will be 1100.

2) Now in the second query we will apply the condition from which we will find the ename, job, and sal. We will use the query as under

```
SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE SAL< 1100;
```

After execution we will get two records. The above two queries can be used as a single query by using the concept of SUBQUERY. It will combine the result into a single query as under:

```
SQL) SELECT ENAME, JOB, SAL FROM EMP WHERE SAL <
       (SELECT SAL FROM EMP WHERE EMPHO=7876);
```

After execution you can see that the output will remain the same. First of all the inner query will be performed and the result returned by it will be used by the outer query to return the final result.
Display the Ename, Job, Sal from EMP working in the LOC 'CHICAGO' and salary is less than the salary of whose empno=7876.

```
SQL> SELECT ENAME, JoB, SAL FROM EMP WHERE DEPINO=
        (SELECT DEPTNO FROM DEPT "WHERE LOC=' CHICAGO ' )
        AND SAL < (SELECT SAL FROM EMP WHERE EMPNO=7876);
```

Group Functions in the Subquery

You can also use the aggregate functions in the subquery because it will produce only single result.
To retrieve the details of the employee holding the minimum salary.

```
SQL) SELECT ENAME,JOB,SAL FROM EMP WHERE SAL=
        (SELECT MIN(SAL) FROM EMP);
```

To retrieve the details of employees whose salary is greater than the minimum salary of the employees working in dept number 20 and also he is not working in dept number 20.

```
SQL) SELECT ENAME, SAL FROM EMP WHERE SAL>
        (SELECT MIN (SAL) FROM EMP WHERE DEPTNO=30)
        AND DEPTNO<>30;
```

The following statement returns all departments in where the minimum salary is more than the minimum salary in the department 20.

```
SQL> SELECT DEPTNO, MIN (SAL) FROM EMP
GROUP BY DEPTNO
HAVING MIN (SAL)> (SELECT MIN (SAL)
FROM EMP WHERE DEPTNO=20);
```

Display the detail of department whose manager's Ecode='7698'.

```
SQL> SELECT * FROM DEPT
WHERE DEPTNO= (SELECT DISTINCT DEPTNO FROM EMP
WHERE MGR=7698);
```

On execution the inner query will give the deptno whose manager's ecode='7698'. Without distinct clause the inner query would have return more than one row as there are number of employees whose manager ecode='7698'.

Multiple-row subqueries

Multiple-row subqueries return sets of rows. These queries are commonly used to generate result sets that will be passed to a DML or SELECT statement for further processing. Both single-row and multiple-row subqueries will be evaluated once, before the parent query is run.

Since it returns multiple values, the query must use the set comparison operators (IN, ALL, ANY). If you use a multi row sub query with the equals comparison operators, the database will return an error if more than one row is returned. The operators in the following table can use multiple-row subqueries:

Symbol	**Meaning**
IN	equal to any member in a list
ANY	returns rows that match any value on a list
ALL	returns rows that match all the values in a list

IN Operator

The IN operator return TRUE if the comparison value is contained in the list; in this case, the results of the subquery.
II The following statement finds the employees whose salary is the same as the minimum salary of the employees in some department.

```
SQL) SELECT ENAHE.SAL FROM EMP
WHERE SAL IN (SELECT MIN (SAL)
FROM EMP GROUP BY DEPTNO);
```

On execution first of all the inner query will return the minimum salary of all the departments and return to outer query. After getting the result from inner query it will compare the result of inner query and return the number of employees

• List all the employees Name and Sal working at the location 'DALLAS'.

```
SQL> SELECI ENAME, SAL, JOB FROM EMP WHERE DEPTNO IN
         (SELECT DEPTNO FROM DEPT WHERE LOC ='DAlIAS');
```

Any Operator

The ANY operators work with the equal operators. The ANY operator returns TRUE if the comparison value matches any of the values in the list.

• Display the employees whose salary is more than the maximum salary of the employees in any department.

```
SQL> SELECT ENAME, SAL FROM EMP
```

```
        WHERE SAL) ANY (SELECT MAX (SAL) FROM EMP
        GROUP BY DEPTNO);
```

Display the employees whose salary is greater than any 'manager' and he is not a 'manager'.

```
SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE SAL>
        ANY (SELECT SAL FROM EMP WHERE JOB='MANAGER')
        AND JOB <>'MANAGER';
```

All Operator

The ALL operator returns TRUE only if the comparison value matches all the values in the list. It compares a value to every value returned by a query. The condition evaluates to true if subquery doesn't yield any row.

Display the employee detail with salary less than those whose job is 'manager'.

```
SQL) SELECT empno, ename, job, sal FROM EMP
       WHERE sal < all (SELECT sal FROM EMP
       WHERE job = 'MANAGER')
       AND job <> 'MANAGER';
```

On execution the query will first select the salary of employees who are 'managers', then it give the result to outer query and it will display the empno, ename, job, salary of those employees who have less salary than any 'manager'.

Multiple – Column Subquery

A subquery that compares more than one column between the parent query and subquery is called the multiple column subqueries. In multiple-column subqueries, rows in the subquery results are evaluated in the main query in pair-wise comparison. That is, column-to-column comparison and row-to-row comparison.
• List the employees that makes the same salary as other employee with empno=752l with the same job also.

```
SQL> SELECT ENAME, JOB, SAL, EMPNO FROM EMP
        WHERE (JOB, SAL) IN (SELECT JOB, SAL FROM EMP WHERE EMPNO=7521);
```

On execution it is clear that first of all it will select the job, sal from employee table whose empno is 7521. then it handover the result to outer query and because now we will use two columns for comparison so after comparing each row with the empno 7521 it will return the desired result.

Correlated Subquery

A correlated subquery has a more complex method of execution than single- and multiple-row subqueries and is potentially much more powerful. If a subquery references columns in the parent query, then its result will be dependent on the parent query. This makes it impossible to evaluate the subquery before evaluating the parent query.

Consider this statement, which lists all employees who earn less than the average salary:

```
SQL> Select Last_name from employees
        Where salary < (select avg (salary) from employees);
```

The single-row subquery need only be executed once, and its result substituted into the parent query.
Select the highest paid employee from the list of each department.

```
SQL) SELECT ENAME, SAL, DEPTNO FROM EMP E1 WHERE SAL =
       (SELECT MAX (SAL) FROM EMP WHERE DEPTNO=E1.DEPTNO) ORDER BY DEPTNO;
```

The subquery will get executed for each row returned in the parent key. We used the alias for the name inside the subquery.

The flow of execution is as follows:

Start at the first row of the EMPLOYEES table.
Read the DEPARTMENT_ID and SALARY of the current row.
Run the subquery using the DEPARTMENT_ID from step 2.
Compare the result of step 3 with the SALARY from step 2, and return the row if the SALARY is less than the result.
• Advance to the next row in the EMPLOYEES table.
Repeat from step 2.

A single-row or multiple-row subquery is evaluated once, before evaluating the outer query; a correlated subquery must be evaluated once for every row in the outer query. A correlated subquery can be single- or multiple-row, if the comparison operator is appropriate.

Subqueries with the INSERT Statement:

The basic syntax is as follows:

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
          SELECT [ *|column1 [, column2 ]
          FROM table1 [, table2 ]
          [ WHERE VALUE OPERATOR ]
```

Example:
Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy complete CUSTOMERS table into CUSTOMERS_BKP, following is the syntax:

```
SQL> INSERT INTO CUSTOMERS_BKP
     SELECT * FROM CUSTOMERS
     WHERE ID IN (SELECT ID
                  FROM CUSTOMERS) ;
```

Subqueries with the UPDATE Statement:

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows:

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

Example:
Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table.
Following example updates SALARY by 0.25 times in CUSTOMERS table for all the customers whose AGE is greater than or equal to 27:

```
SQL> UPDATE CUSTOMERS
     SET SALARY = SALARY * 0.25
     WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
                   WHERE AGE >= 27 );
```

Subqueries with the DELETE Statement:

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows:

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

Example:
Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table.

Following example deletes records from CUSTOMERS table for all the customers whose AGE is greater than or equal to 27:

```
SQL> DELETE FROM CUSTOMERS
     WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
                   WHERE AGE > 27 );
```

****

SQL Tuning or SQL Optimization

Sql Statements are used to retrieve data from the database. We can get same results by writing different sql queries. But use of the best query is important when performance is considered. So you need to sql query tuning based on the requirement. Here is the list of queries which we use reqularly and how these sql queries can be optimized for better performance.

SQL*PLUS: Batch Mode

Use your favorite editor to create a file with extension '.sql' containing your SQL command(s).

For instance, use 'emacs' to create a file, 'exCreate.sql' containing the following:

```
SET TERMOUT ON

PROMPT Create Table Student

SET TERMOUT OFF

SET FEEDBACK ON

-- Drop the old table before create. -- In line comment

DROP TABLE Student;

REMARK Create a table Student

CREATE TABLE Student
( Name  VARCHAR2(30),
StudentNumber NUMBER(4) NOT NULL,
Class NUMBER(4),
Major VARCHAR2(4),
Primary key (StudentNumber)
);

/* Insert data into the Student table */

INSERT INTO Student VALUES ('Smith', 17, 1, 'COSC');
INSERT INTO Student VALUES ('Brown', 8, 2, 'COSC');
INSERT INTO Student VALUES ('Senior Answer1', 421, 5, 'COSC');
INSERT INTO Student VALUES ('Dick Davidson', 110, 1, 'COSC');
INSERT INTO Student VALUES ('Babara Benson', 28, 2, 'ECSE');
INSERT INTO Student VALUES ('Charlie Cooper', 21, 2, 'DCSC');
INSERT INTO Student VALUES ('Katherine Ashly', 138, 1,'COSC');
INSERT INTO Student VALUES ('Benjamin Bayer', 430, 5, 'EPW');
INSERT INTO Student VALUES ('Senior Crew', 492, 5, 'COSC');
INSERT INTO Student VALUES ('John', 362, 3, 'CIVI');
INSERT INTO Student VALUES ('Proc', 123, 1, NULL);

COMMIT;
```

From the batch file, you can run it by typing at prompt:

```
@[filename] or

start [filename]
```

Note that the file extension '.sql' can be omitted.

```
SQL> @exCreate
Create Table Student
```

```
SQL>
or
SQL> START exCreate
Create Table Student
SQL>
```

****

SQL Tuning/SQL Optimization Techniques:

1) The sql query becomes faster if you use the actual columns names in SELECT statement instead of than '*'.
For Example:

Write the query as

```
SELECT id, first_name, last_name, age, subject FROM student_details;
```

Instead of:

```
SELECT * FROM student_details;
```

2) HAVING clause is used to filter the rows after all the rows are selected. It is just like a filter. Do not use HAVING clause for any other purposes.
For Example: Write the query as

```
SELECT subject, count(subject)
FROM student_details
WHERE subject != 'Science'
AND subject != 'Maths'
GROUP BY subject;
```

Instead of:

```
SELECT subject, count(subject)
FROM student_details
GROUP BY subject
HAVING subject!= 'Vancouver' AND subject!= 'Toronto';
```

3) Sometimes you may have more than one subqueries in your main query. Try to minimize the number of subquery block in your query.
For Example: Write the query as

```
SELECT name
FROM employee
WHERE (salary, age ) = (SELECT MAX (salary), MAX (age)
FROM employee_details)
AND dept = 'Electronics';
```

Instead of:

```
SELECT name
FROM employee
WHERE salary = (SELECT MAX(salary) FROM employee_details)
AND age = (SELECT MAX(age) FROM employee_details)
AND emp_dept = 'Electronics';
```

4) Use operator EXISTS, IN and table joins appropriately in your query.
a) Usually IN has the slowest performance.
b) IN is efficient when most of the filter criteria is in the sub-query.
c) EXISTS is efficient when most of the filter criteria is in the main query.

For Example: Write the query as

```
Select * from product p
where EXISTS (select * from order_items o
where o.product_id = p.product_id)
```

Instead of:

```
Select * from product p
where product_id IN
(select product_id from order_items
```

5) Use EXISTS instead of DISTINCT when using joins which involves tables having one-to-many relationship.
For Example: Write the query as

```
SELECT d.dept_id, d.dept
FROM dept d
WHERE EXISTS ( SELECT 'X' FROM employee e WHERE e.dept = d.dept);
```

Instead of:

```
SELECT DISTINCT d.dept_id, d.dept
FROM dept d,employee e
WHERE e.dept = e.dept;
```

6) Try to use UNION ALL in place of UNION.
For Example: Write the query as

```
SELECT id, first_name
FROM student_details_class10
UNION ALL
SELECT id, first_name
FROM sports_team;
```

Instead of:

```
SELECT id, first_name, subject
FROM student_details_class10
UNION
SELECT id, first_name
FROM sports_team;
```

7) Be careful while using conditions in WHERE clause.
For Example: Write the query as

```
SELECT id, first_name, age FROM student_details WHERE age > 10;
```

Instead of:

```
SELECT id, first_name, age FROM student_details WHERE age != 10;
```

Write the query as

```
SELECT id, first_name, age
FROM student_details
WHERE first_name LIKE 'Chan%';
```

Instead of:

```
SELECT id, first_name, age
FROM student_details
WHERE SUBSTR(first_name,1,3) = 'Cha';
```

Write the query as

```
SELECT id, first_name, age
FROM student_details
WHERE first_name LIKE NVL ( :name, '%');
```

Instead of:

```
SELECT id, first_name, age
FROM student_details
WHERE first_name = NVL ( :name, first_name);
```

Write the query as

```
SELECT product_id, product_name
FROM product
WHERE unit_price BETWEEN MAX(unit_price) and MIN(unit_price)
```

Instead of:

```
SELECT product_id, product_name
FROM product
WHERE unit_price >= MAX(unit_price)
and unit_price <= MIN(unit_price)
```

Write the query as

```
SELECT id, name, salary
FROM employee
WHERE dept = 'Electronics'
AND location = 'Bangalore';
```

Instead of:

```
SELECT id, name, salary
FROM employee
WHERE dept || location= 'ElectronicsBangalore';
```

Use non-column expression on one side of the query because it will be processed earlier.

Write the query as

```
SELECT id, name, salary
FROM employee
WHERE salary < 25000;
```

Instead of:

```
SELECT id, name, salary
FROM employee
WHERE salary + 10000 < 35000;
```

Write the query as

```
SELECT id, first_name, age
FROM student_details
WHERE age > 10;
```

Instead of:

```
SELECT id, first_name, age
FROM student_details
WHERE age NOT = 10;
```

8) Use DECODE to avoid the scanning of same rows or joining the same table repetitively. DECODE can also be made used in place of GROUP BY or ORDER BY clause.

Example:

Write the query as

```
SELECT id FROM employee
WHERE name LIKE 'Ramesh%'
and location = 'Bangalore';
```

Instead of:

```
SELECT DECODE(location,'Bangalore',id,NULL) id FROM employee
WHERE name LIKE 'Ramesh%';
```

9) To store large binary objects, first place them in the file system and add the file path in the database.

10) To write queries which provide efficient performance follow the general SQL standard rules.

a) Use single case for all SQL verbs
b) Begin all SQL verbs on a new line
c) Separate all words with a single space
d) Right or left aligning verbs within the initial SQL verb.

SQL Tables for Exercises

Que: 1 Create the following tables:

Table Name: **Client_master**
Description : **Use to store information about clients.**

Column Name	Data Type	Size	Attributes
Client no	Varchar	6	Primary Key / First letter must start with 'C'.
Name	Varchar	20	Not Null
Address"!	Varchar	30	
Address2	Varchar	30	
City	Varchar	15	
State	Varchar	15	
Pincode	Numeric	6	
Bal due	Numeric	10,2	

Table Name: **product_master**
Description : **Use to store information about products.**

Column Name	Data Type	Size	Attributes
Product no	Varchar	6	Primary Key / first letter must start with 'P'
Description	Varchar	5	Not Null
Profit_percent	Numeric	5,2	Not Null

Unit measure	Varchar	10	Not Null
Qty on hand	Numeric	8	Not Null
Reorder Ivl	Numeric	8	Not Null
Sell price	Numeric	8,2	Not Null, Cannot be 0.
Cost price	Numeric	8,2	Not Null, Cannot be 0.

Table Name: **salesman_master**
Description : **Use to store information about salesman working in the company**

Column Name	Data Type	Size	Attributes
Salesman no	Varchar	6	Primary Key / First letter must start with 'S'
Salesman name	Varchar	20	Not Null
Addressl	Varchar	30	Not Null
Address2	Varchar	30	
City	Varchar	20	
Pincode	Varchar	6	
State	Varchar	20	
Sal amt	Numeric	8,2	Not Null, cannot be 0
Tgt to get	Numeric	6,2	Not Null, cannot be 0
Ytd sales	Numeric	6,2	Not Null
Remarks	Varchar	60	

Table Name: **sales_order**
Description : **Use to store information about order**

Column Name	Data Type	Size	Attributes
S order no	Varchar	6	Primary Key / First letter must start with 'O'
S order date	Date		
Client no	Varchar	6	Foreign Key references client_no of client_master table
Dely addr	Varchar	25	
Salesman_no	Varchar	6	Foreign Key references salesman no of salesman master table.
Dely type	Char	1	Delivery : part (P) /full (F), Default 'F'
Billed yn	Char	1	
Dely date	Date		Cannot be less than s order date
Order status	Varchar	10	Values('In Process', 'Fulfilled', 'BackOrder'.'Canceled')

Table Name: **sales_order_details**
Description : **Use to store information about products ordered.**

Column Name	Data Type	Size	Attributes
S_order_no	Varchar	6	Primary Key / Foreign Key references s_order_no of sales order table.
Product_no	Varchar	6	Primary Key / Foreign Key references product_no of product_master table.
Qty_ordered	Numeric	8	
Qty_disp	Numeric	8	
Product rate	Numeric	10,2	

Table Name: **ChallanJHeader**
Description : **Use to store information about challans made for the order.**

Column Name	Data Type	Size	Attributes
Challan no	Varchar	6	Primary Key / first two letters must start with 'CH'.
S order no	Varchar	6	Foreign key references s order no of sales order table.
Challan date	Date		Not Null
Billed_yn	Char	1	Values('YYN'), Default 'N'.

Que: 2 Insert the following data into their respective tables using the SQL insert statement:

1. Data for client master table:

Client_no	Name	City	Pincode	State	Bal_due
C00001	Ivan Bayross	Bombay	400054	Maharashtra	15000
C00002	Vandana Saitwal	Madras	780001	Tamil Nadu	0
C00003	Pramada Jaguste	Bombay	400057	Maharashtra	5000
C00004	Basu Navindgi	Bombay	400056	Maharashtra	0
C00005	Ravi Sreedharan	Delhi	100001		2000
C00006	Rukmini	Bombay	400050	Maharashtra	0

2. Data for Product Master table:

Product_no	Description	Profit	UOM	Qty_on_hand	Recorded_lvt	Sell_price	Cost_price
P00001	1 .44 Floppies	5	Piece	100	20	525	500
P03453	Monitors	6	Piece	10	3	12000	11280
P06734	Mouse	5	Piece	20	5	1050	1000
P07865	1 .22 Floppies	5	Piece	100	20	525	500
P07868	Keyboards	2	Piece	10	3	3150	3050
P07885	CD Drive	2.5	Piece	10	3	5250	5100
P07965	540 HDD	4	Piece	10	3	8400	8000
P07975	1 .44 Drive	5	Piece	10	3	1050	1000
P08865	1 .22 Drive	5	Piece	2	3	1050	1000

3. Data for Salesman master table:

Salesman No	Salesman Name	Add1	Add2	City	Pincode	State	Sal Amt	Tgt_to Get	Ytd Sales	Remarks
S00001	Kiran	A/14	Worli	Bombay	400002	MAH	3000	100	50	Good
S00002	Manish	65	Nariman	Bombay	400001	MAH	3000	200	100	Good
S00003	Ravi	P-7	Bandra	Bombay	400032	MAH	3000	200	100	Good
S00004	Ashish	A/5	Juhu	Bombay	400044	MAH	3000	200	150	Good

4. Data for sales order table:

S_order_no	S_order_date	Client_no	Delay	Bill YN	Salesman No	Delay_date	Order_Status
O1 9001	12-jan-96	C00001	F	N	S00001	20-jan-96	IP
019002	25-jan-96	C00002	p	N	S00002	27-jan-96	C
O46865	18-feb-96	C00003	F	Y	S00003	20-feb-96	F
O1 9003	03-apr-96	C00001	F	Y	S00001	07-apr-96	F
046866	20-may-96	C00004	P	N	S00002	22-may-96	C
O1 0008	24-may-96	C00005	F	N	S00004	26-may-96	IP

5. Data for sales order details table:

Order_ID	Product_ID	Qty_ordered	Qty_dsg	Product_rate
O1 9001	P00001	4	4	525
019001	P07965	2	1	8400
019001	P07885	2	1	5250
O19002	P00001	10	0	525
046865	P07868	3	3	3150
O46865	P07885	3	1	5250
O46865	P00001	10	10	525
046865	P03453	4	4	1050
O19003	P03453	2	2	1050
019003	P06734	1	1	12000

046866	P07965	1	0	8400
O46866	P07975	1	0	1050
010008	P00001	10	5	525
O1 0008	P07975	5	3	1050

6. Data for challan header table:

Challan no	S order no	Challan date	Billed
CH9001	O1 9001	12-dec-95	Y
CH6865	O46865	12-nov-95	Y
CH3965	010008	12-oct-95	Y

7. Data for challan details table:

Challan_no	Product_no	Qty_disp
CH9001	P00001	4
CH9001	P07965	1
CH9001	P07885	1
CH6865	P07868	3
CH6865	P03453	4
CH6865	P00001	10
CH3965	P00001	5
CH3965	P07975	2

Perform the following queries on the basis of these tables.

Single Table Retrieval:

1. Find out the names of all the clients.
2. Print the entire client-master table.
3. Retrieve the list of names and the cities of all the clients.
4. List the various products available from the product_master table.
5. Find the name of all clients having 'a' as the second letter in their names.
6. Find out the clients who stay in a city whose second letter is 'a'.
7. Find the list of all clients who stay in 'Bombay' or city 'Delhi' or city 'Madras'.
8. List all the clients who are located in Bombay.
9. Print the list of clients whose bal_due are greater than value 10000.
10. Print the information from sales_order table of orders placed in the month of January.
11. Display the order information for client_no 'cOOOOl 'and 'c00002'.
12. Find the products with description as '1.44 Drive' and '1.22 Drive'.
13. Find the products whose selling price is greater than 2000 and less then or equal to 5000.
14. Find the products whose selling price is more than 1500 and also find the new selling price as orignal price * 15.
15. Rename the new column in the above query as new_price.
16. Find the products whose cost price is less then 1500.
17. List the products in sorted order of their description.
18. Calculate the square root of the price of the each product.
19. Divide the cost of product '540 HDD' by difference between its price and 100.
20. List the names, city and state of clients not in the state of 'Maharashtra'.
21. List the products_no, description, sell_price of products whose description begin with letter 'M'.
22. 22.. List of all orders that were canceled in the month of March.

Set functions and concatenation:

1. Count the total number of orders.
2. Calculate the average price of all the products.
3. Calculate the minimum price of products.
4. Determine the maximum and minimum product prices. Rename the title as max_price and min_price respectively.
5. Count the numbers of products having price greater than or equal to 1500.
6. Find all products whose qty_on_hand is less than recorder level.

7. Print the information of client_master, product_master, sales_order, table in the following format for all the records: - {cust_name} has placed order {order_no} on {s_order_date}.

Having and Group By:

1. Print the description and total qty sold for each product.
2. Find the value of each product sold.
3. Calculate the average qty sold for each client that has a maximum order value of 15000.00
4. Find out the total sales amount receivable for the month of Jan. it will be the sum total of all the billed orders for the month.
5. Print the information of product_master, order_detail table in the following format for all the records:- {descriptionjworth RS.{total sales for the products} was sold.
6. Print the information of product_master,order_detail table in the following format for all the records:- {descriptionjworth Rs. {total sales for the product} was ordered in the month of {s_order_dfate in the month format}.

Joins and Correlation:

1. Find out the products which has been sold to 'ivan bayroos'.
2. Find out the products and their quantities that will have to delivered in the current month.
3. Find the product_no and description of moving products.
4. Find the names of the clients who have purchased 'CD Drive'.
5. List the product_no and s_order_no of customers having qty_ordered less than 5 from the order details Table for the product '1.44 Floppies'.
6. Find the products and their quantities for the orders placed by Vandana Saitwal' and 'Ivan Bayross'.
7. Find the products and their quantities for the orders placed by client_no 'C00001' and 'C00002'.

Nested Queries:

find the product_no and description of non-moving products.
Find the customers name, address"!, address2, city and pincode for the client who has placed order no "O19001[1]
Find the client name who have placed orders before the month of may,96.
Find out if product "1.44 Drive" is ordered by any client and print client_no, name to whom it was sold.
Find the name of clients who have placed ordered worth Rs. 10000 or more.

Query Using Date:

1. Display the order no and day on which client placed their order.
2. Display the month (in alphabets) and date when the order must be delivered.
3. Display s_order_date in the format "dd-month-yy" e.g. 12-February-96.
4. Find the date, fifteen days after todays date.
5. Find the no of days elapsed between todays date and the delivery date of the orders placed by the clients.

Table updations:

1. Change the s_order_Date of client_no 'C00001' to 24/07/96.
2. Change the selling price of '1.44 Drive' to Rs. 1150.00
3. Delete the record with order no 'O19001' from the order table.
4. Delete all the records having delivery date before 10-Jul-96.
5. Change the city of client_no 'C00005' to 'Bombay'.
6. Change the delivery date of order no 'O10008' to 16-08-96.
7. Change the bal_due of client_no 'C00001' to 1000.
8. Change the cost price of '1.22 Floppy Drive' to Rs. 950.00.

Lab Session

(You can do something extraordinary, and something that a lot of people can't do. And if you have the opportunity to work on your gifts, it seems like a crime not to. I mean, it's just weakness to quit because something becomes too hard..)

Login to Linux/Unix:

User Name: root (or) any normal user
Password: *********

Open the terminal by clicking the right button which gets appear dropdown menu

Select 'Open Terminal'
- OR -
Go to --- Application --> Accessories --> Terminal

```
[root@machine1 ~]#
[root@machine1 ~]#
```

1. Checking the databases (the file 'oratab file' will get appear only when you already installed Oracle db binaries and database/s) :

[root@machine1 ~]# **cat /etc/oratab**

```
#Backup file is  /u01/app/11.2.0/grid/srvm/admin/oratab.bak.machine1 line added by Agent
#
# This file is used by ORACLE utilities.  It is created by root.sh
# and updated by either Database Configuration Assistant while creating
# a database or ASM Configuration Assistant while creating ASM instance.

# A colon, ':', is used as the field terminator.  A new line terminates
# the entry.  Lines beginning with a pound sign, '#', are comments.
#
# Entries are of the form:
#   $ORACLE_SID:$ORACLE_HOME:<N|Y>:
#
# The first and second fields are the system identifier and home
# directory of the database respectively.  The third filed indicates
# to the dbstart utility that the database should , "Y", or should not,
# "N", be brought up at system boot time.
#
# Multiple entries with the same $ORACLE_SID are not allowed.
#
#
+ASM:/u01/app/11.2.0/grid:N
asmdb:/u01/app/oracle/product/11.2.0/dbhome_1:N
devdb:/u01/app/oracle/product/11.2.0/dbhome_1:N       # line added by Agent
prodb:/u01/app/oracle/product/11.2.0/dbhome_1:Y      # line added by Agent
testdb:/u01/app/oracle/product/11.2.0/dbhome_1:N      # line added by Agent
```

2. Switch to Oracle database User:

```
# su - oracle
$
```

3. To check whether any database/s are running or not

$ **ps -ef | grep pmon or smon**

```
oracle   8484   1  0 16:47 ?      00:00:02 ora_pmon_devdb
oracle   5678   1  0 16:47 ?      00:00:02 ora_pmon_prodb
```

Note: If any database which appears in the oratab file & it was already closed .. . then

4. To set an environment variable related to database or instance name

$ **export ORACLE_SID=peterdb**

5. Open the sql prompt

```
$ sqlplus / as sysdba
SQL>
```

6. Starting up the database now:

SQL> **startup**

7. Check the current User

SQL> **show user** ----> sys

8. Check the following database name, users by using dictionary views:

```
SQL> select name from v$database;
SQL> select * from all_users;
```

9. Go to any user: Example: scott

SQL> **connect scott/******

10. To check the tables in the entire current user:

```
SQL> select * from tab
```

Sub Languages

DDL (Data Defination Langauge)
create, alter, drop, truncate, remane

DML (Data Manipulation Language)
insert, update, delete

TCL (Transaction Control Language)
commit,rollback,savepoint

DCL (Data Control Language)
grant, revoke

```
create table t1 (
tno number(2),
tname varchar2(10)
);

insert into table t1 values (1,'peter','12-jan-2002')
insert into table t1 values (2,'ravi','22-mar-1999')
insert into table t1 values (3, 'ahmed','04-jun-2013')
commit;

insert into table t1 values (&tno,'&tname','&tdate')

select * from t1;
select tno from t1;
select tname from t1;
select * from t1 where tno=2;

update t1 set tno=123 where tname='peter';
update t1 set tname='mohd' where tno=3;

delete from t1 where tno-123;
delete from t1;
rollback;
```

(as it will go to last commit point where it marks to the last transaction)

```
drop table t1;

truncate table t1;
```

alter table old_table_name rename to new_table_name

```
alter table p1 add pdate date;

alter table p1 add comments varchar2(15);

alter table p1 drop column pdate;

alter table p1 modify comments(12);
```

To create table with Key Constraints(NN,UK,PK)

```
create table school
(
stno number(3), constraint stno_pk primary key(stno),
stname varchar2(10) not null,
course varchar2(12) not null,
batch varchar2(10) not null,
fees number(9,2) not null,
mobile number(12) unique
)
select owner,constraint_name,constraint_type from user_constraints
alter table school
add constraint course_chk
check (course in ('dba','sql','php','java'))
/


create table course
(
course varchar2(12),
stno number(3) constraint stno_fk  references school(stno)
)
/
```

Practice Session
===============

Table1
-----
university
 columns ---> uid,cid,cname,cdept,cadm,cfees,cjoindate

Table2
------
college
 column --> cid,cname,cloc, uid

```
create table university
(
uno number(3), constraint uno_pk primary key(uno),
uname varchar2(9) not null,
collegename varchar2(8) not null,
sectionname varchar2(12) not null,
collegefees number(7,2),
cjoindate date
)

sql> alter table university add constraint collegefees_chk check(collegefees between 1000 and 8000)

sql> alter table university add constraint sectionname_chk check(sectionname in ('english','maths','physics'))

create table college
(
cid number(3) unique,
collegename varchar2(12),constraint collegename_pk primary key(collegename),
joindate date
)
alter table university add constraint collegename_fk foreign key(collegename) references college(collegename)

insert into college values (&cid,'&collegename','&joindate');
/
commit;

insert into university values (&uno,'&uaname','&collegename','&sectionname',&collegefees)
/
commit;
```

****

Simple Project
==============
Prepare the different tables which should be references from one table to another by using SQL.

Master Tables
----
1. Student Table
2. Visa Table
3. Passport Table

Transaction Tables

```
--------------------------
  1. Aptech Table
  2. Attendance Table
  3. Course Table

****************************************************************************

create table t1 (tno number(3),tname varchar2(10), tdate date);
Table created.

SQL> insert into t1 values (1,'suman','12-april-2015');
1 row created.

SQL> select * from t1;
       TNO TNAME      TDATE
---------- ---------- ---------
         1 suman      12-APR-15
--------------------------------------------------------------------

SQL> select sysdate from dual;

SYSDATE
---------
12-MAY-16

SQL> select 1+5 from dual;
       1+5
----------
         6

SQL> select 5*5 from dual;
       5*5
----------
        25

SQL> select sum(12+23+4+56+78) from dual;
SUM(12+23+4+56+78)
------------------
               173
SQL>
SQL> desc dual

====================================================================

SQL> set pages 100
SQL> set lines 130

SQL> create table  e1 as select * from emp;
SQL

SQL> select empno,ename,job,sal from e1;

SQL> select max(sal) from e1;
SQL> select min(sal) from e1;
SQL> select sum(sal) from e1;
SQL> select sum(sal) as total_sal from e1;
SQL> select empno,ename,job,sal from e1 where sal>2000;
SQL> select * from e1 where deptno=20;
SQL> select empno,ename,sal,sal+500 as bonus from emp where deptno=10;
select empno,ename,sal,sal+500 from emp where deptno=10
SQL> select empno,ename,sal from e1 where sal between 1000 and 1500;
SQL> select * from e1 where comm is null;
SQL> select * from e1 where comm is not null;
SQL> select * from e1 where deptno in (10,30);

SQL> select * from e1 where sal<=2000;
SQL> select * from e1 where sal>2500;
SQL> select * from e1 where ename like 'J%';
SQL> select * from e1 where job like 'M%';
SQL> select * from e1 where job not like 'M%';
 select * from e1 where job like '__E%'
SQL> select * from e1 where job like '%K'
SQL> select * from e1 where ename like '%__I';
SQL> select * from e1 where sal>2000 and deptno=10;

SQL> select * from e1  where sal between 1000 and 1500 and comm is not null;
select * from emp where sal>2000 and deptno=100;
select * from emp where sal>2000 or deptno=100;
select * from emp where sal>9000 or deptno=10;

SQL> select * from e1 where hiredate='28-SEP-81';
SQL> select * from e1 where job='manager' and sal>3000;
select * from e1 where job='manager' or sal>3000
SQL> select * from e1 where sal>1000 and job=lower('manager');
 select * from e1 where sal>1000 or job='MANAGER'
 select * from e1 where sal>10000 or job='MANAGER'

select lower(ename),upper(ename),initcap(ename) from emp;

select * from emp where sal>4000;
select * from emp where deptno=10;
select * from emp where job='CLERK';

select * from emp where ename like 'C%'
select * from emp where ename like '%D';
select * from emp where ename like '%N'
select * from emp where ename like '__L%'

select * from emp where sal between 1000 and 1500;
select * from emp where sal not between 1000 and 1500;
select * from emp where deptno in (10,30);
select * from emp where deptno not in (10,30);
select * from emp where comm is null;
select * from emp where comm is not null;

select avg(sal) from emp;
select round(avg(sal)) from emp;
select trunc(avg(sal),2) from emp;

select distinct(ename) from emp;
select empno,ename,sal,job from emp;
select empno as eno,ename as name,sal as salary,job as ejob from emp;

select empno||ename||job from emp;
select empno||' is the '||ename|| ' job of '||job from emp
select empno||' is the '||ename|| ' job of '||job as "Employees Details" from emp;
select empno,ename,job,sal*12 as "annual pay" from emp;
select empno,ename,job,sal*12 "annual pay" from emp;

select empno,ename,job,sal from emp order by empno;
select empno,ename,job,sal from emp order by empno desc;
select empno,ename,job,sal from emp order by empno,ename desc;
select empno,ename,job,sal,hiredate from emp order by hiredate;
select empno,ename,job,sal,hiredate from emp order by hiredate desc;
select empno,ename,job,sal from emp order by 3;
select * from emp where deptno=&deptno;

select max(sal) from emp;
select empno,max(sal) from emp;
        ORA-00937: not a single-group group function
select empno,max(sal) from emp group by empno;

select empno,ename,max(sal) from emp group by empno;
     ORA-00979: not a GROUP BY expression
select empno,ename,max(sal) from emp group by empno,ename;

select deptno,max(sal) from emp group by deptno having max(sal)>1000;
select deptno,max(sal) from emp group by deptno having max(sal)>2000;
select deptno,max(sal) from emp group by deptno having max(sal)>2000 order by deptno;
select deptno,max(sal) from emp group by deptno having max(sal)>2000 order by deptno desc;
```

==
TO_CHAR Conversion
----
select ename,to_char(hiredate,'month DDth,yyyy') as Hiredate from emp where job='MANAGER';
select ename,to_char(hiredate,'dd-month-yyyy') as hiredate from emp where job='MANAGER';
select ename,to_char(hiredate,'dy/mm/yy') from emp where job='MANAGER'
select ename,to_char(hiredate,'day/mm/yy') from emp where job='MANAGER'
select ename,to_char(hiredate,'yyyy-mon-dd') from emp where job='MANAGER'
select to_char(sysdate,'hh24:mi:ss') Time from dual; select to_char(sal,'$99,999') Salary from emp where ename='KING';

TO_NUMBER Conversion
----

select empno,ename, to_number('1000')+sal as new_salary from emp;

select ename,hiredate,add_months(hiredate,2) as "+2 Months" from emp where ename='BLAKE'
select ename,hiredate,add_months(hiredate,4) as "+4 Months" from emp where ename='BLAKE'
select ename,nvl(to_char(mgr),'Noooooo') from emp where mgr is null;
select months_between(to_date('05-02-1995','mm-dd-yyyy'),to_date('01-01-1995','mm-dd-yyyy')) "Months" from dual;

==

select to_char(min(hiredate),'dd-mon-yyyy'), to_char(max(hiredate),'dd-mon-yyyy') from emp;

select min(sal) as "Lowest Salary",max(sal) as "Highest Salary" from emp;

select count(*) from emp where deptno=20;
select count(comm) from emp where comm is null;
select count('COMM') from emp where comm is not null;
select avg(comm) from emp;
select avg(nvl(comm,0)) from emp;
==

SQL Join Queries

select e.ename,e.job,e.deptno,d.dname,d.loc
from emp e,dept d
where e.deptno=d.deptno;

select e.ename,e.job,e.deptno,d.dname,d.loc
from emp e,dept d
where e.deptno=d.deptno and ename='KING';

select e.ename,e.job,e.deptno,d.dname,d.loc
from emp e,dept d
where e.deptno=d.deptno or ename='KING';

select e.ename,e.job,e.deptno,d.dname,d.loc
from emp e,dept d
where e.deptno=d.deptno and e.job in ('MANAGER','CLERK');

Non-Equi-Join
----
select e.empno,e.ename,e.sal,s.grade
from emp e, salgrade s
where e.sal between s.losal and s.hisal;

Joining More than Two Tables:
----
using three tables (emp, dept, salgrade)

select e.ename,e.deptno,d.dname,e.sal,s.grade
from emp e, dept d, salgrade s
where e.deptno=d.deptno
and e.sal between s.losal and s.hisal;

Self Join
----
select worker.ename || ' works for '|| manager.ename
as WHO_WORKS_FOR_WHOM

from emp worker, emp manager
where worker.mgr = manager.empno
==

SQL - Sub Queries
----

select ename from emp where sal > (select sal from emp where ename='JONES');

select ename from emp where sal < (select sal from emp where ename='JONES');

Who works in the same department as 'king'
select ename,deptno from emp where deptno > (select deptno from emp where ename = 'KING');

select ename,deptno from emp where deptno > (select deptno from emp where ename <= 'KING');

Who has the same manager as 'Blake'
select ename,mgr from emp where mgr = (select mgr from emp where ename = 'BLAKE');

Who has the same job as employee 7369 and earns a higher salary than employee 7876?
select ename,job from emp where job = (select job from emp where empno=7369) and sal > (select sal from emp where empno=7876);

Display all employees who earn the minimum salary
select empno,ename,job from emp where sal = (select min(sal) from emp);

Using Group Functions in a Multiple-Row Subquery
==

Question-1:

Display all employees who earn the same salary as the minimum salary for each department.

SQL> select ename,sal,deptno from emp where sal in (select min(sal) from emp group by deptno);

Question-2:

Display the employees who were hired on the same date as the logest serving employee in any department.

SQL> elect ename,sal,deptno,to_char(hiredate,'dd-mon-yyyy') Hiredate from emp where hiredate in
(select min(hiredate) from emp group by deptno);

==
Oracle DB Objects (Schema)
=========================

Table:

Baisc unit of storage; composed of rows

View:

Logically reopresents subsets of data from one or mor tables.

Sequence

Generates numeric values

Index:

Improves the performance of some queries

Synonym:

Gives alternative names to objects

Views:

- Logical representation of table/s, the main advantages of using views are to hide table name and/or to hide the table's columns with different names.
Two types of views
----
1. Simple views
   (using only single table with simple query)
2. Complex views
   (using multiple tables with complex queries)

Simple view example

```
  create view vemp1
  as
  select empno,ename,job,sal,deptno from emp
  where deptno in (10,30)

select * from vemp1;

drop view vemp1;

create view vvemp as select empno as eno,ename as name,job as ejob,sal as salary from emp;

 select * from vvemp;

create or replace view vvemp as select empno as eno,ename as name,job as ejob from emp;

select * from vvemp;
```

Complex view example

```
create view emp_dept
as
select e.empno as eno,e.ename as name,e.job,e.deptno,d.dname,d.loc as dlocation
from emp e,dept d
where e.deptno=d.deptno;
```
==

Oracle sql Indexes

```
SQL> create table h1 as select * from emp;

SQL> select * from h1;

SQL> create index h1_index on h1(empno);

SQL> set time on
17:45:51 SQL>
```

Exercise (Home Work)
----

create table with thousands of records insert into it.
```
create table t1 as select * from emp;
insert into t1 select * from t1;
/ / / / / / / / / / / / / / / /  commit;

create table t2 as select * from t1;
create index t2_index on t2(empno);
------------------------------------------------------------
col index_name for a18
col index_type for a18
col table_owner for a18
col table_name for a18
select index_name,index_type,table_owner,table_name from user_indexes;
```

To Drop Index

```
drop index index_name
drop index  h1_index
```

==

DB Sequence

```
SQL>
create sequence seq123
increment by 10
start with 120
maxvalue 999
nocache
nocycle;

SQL> create table test1 (testid number(3),testname varchar2(10),testloc varchar2(10));

SQL> insert into test1 (testid,testname,testloc) values (seq123.nextval,'support',2500);

SQL> commit;

SQL> select * from test1;
  TESTID TESTNAME   TESTLOC
---------- ---------- ----------
     120 support    2500


SQL> alter sequence seq123
  2  increment by 5
  3  maxvalue 999;
```

To drop Sequence

```
SQL> drop sequence seq123;
```

Some other Examples

```
SQL> create table s1 (sno number,sname varchar2(10));
insert into s1 (sno,sname) values (seq123.nextval,'john')
 insert into s1 (sno,sname) values (seq123.nextval,'peter')
insert into s1 (sno,sname) values (seq123.nextval,'abdul')
```

==

Synonyms

It is one of the database object that enable you to call a tableby another name. as we can create synonyms to give an alternative name to a table/view/sequence/procedure/any dbobjects.

To create synonyms for a Table

Note: you should have create synonyms privileges

```
conn / as sysdba
SQL> grant create synonym to scott;
SQL> conn scott/*****

SQL> create synonym employ for scott.emp;
```

To create synonym for a View

```
create synonym ed for scott.emp_dept;
```

To Drop synonym

```
SQL> drop synonym ed;
```

==

User Management & Security

```
sqlplus / as sysdba
```

show user --> **sys**

```
SQL> create user max identified by max (In short)

SQL> create user abab identified by abab default tablespace razatbs temporary tablespace temp quota unlimited on razatbs ;

SQL> conn abab/abab
ERROR:
ORA-01045: user ABAB lacks CREATE SESSION privilege; logon denied

Warning: You are no longer connected to ORACLE.

SQL> show user
USER is ""

SQL> conn / as sysdba
Connected.
SQL>
SQL> grant connect to abab;
Grant succeeded.

SQL> conn abab/abab
Connected.

SQL> select  * from tab;
no rows selected

SQL> create table t1 (tno number);
create table t1 (tno number)
*
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> conn / as sysdba
Connected.

SQL> grant create table to abab;
Grant succeeded.

SQL> conn abab/ababa
ERROR:
ORA-01017: invalid username/password; logon denied
Warning: You are no longer connected to ORACLE.

SQL> conn abab/abab
Connected.

SQL> select * from tab;
no rows selected

SQL> create table t1 (tno number);
Table created.

SQL> create view v1 as select  * from t1;
create view v1 as select  * from t1
       *
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> conn / as sysdba
Connected.

SQL> grant create view to abab;
Grant succeeded.

SQL> conn abab/abab
Connected.

SQL>  create view v1 as select  * from t1;
View created.
```

```
SQL> conn / as sysdba
Connected.

SQL> revoke create table,create view from abab;
Revoke succeeded.

SQL> conn abab
Enter password:
Connected.

SQL> show user
USER is "ABAB"
SQL>
SQL> create table t2 (tno number);
create table t2 (tno number)
*
ERROR at line 1:
ORA-01031: insufficient privileges


SQL> create view v2 as select * from t1;
create view v2 as select * from t1
          *
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> revoke connect from abab;
Revoke succeeded.

SQL> conn abab/abab
ERROR:
ORA-01045: user ABAB lacks CREATE SESSION privilege; logon denied

Warning: You are no longer connected to ORACLE.
SQL>
```

===

```
SQL> desc all_users;
SQL> select * from all_users;
```

Permissions

```
grant connect to halemuser;
grant create table to halemuser;
grant create view to halemuser;

revoke create table from halemuser;
revoke create view from  halemuser;
revoke connect from halemuser;
```

To change password of a user

```
alter user halemuser identified by halem123;
```

To Lock/Unlock the user Account

```
alter user halemuser account unlock;
alter user halemuser account lock;
```

===