ALAGAPPA UNIVERSITY

(Accredited with 'A+' Grade by NAAC (with CGPA: 3.64) in the Third Cycle and Graded as category - I University by MHRD-UGC)
(A State University Established by the Government of Tamilnadu)

KARAIKUDI – 630 003

DIRECTORATE OF DISTANCE EDUCATION

B.C.A

Second Year – Third Semester

101 34 / 127 34

RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS) - LAB

Author:
Dr. C.Balakrishnan
Assistant Professor
Alagappa Institute of Skill Development
Alagappa University,
Karaikudi. 630 003.

RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS) - LAB

SYLLABI-BOOK MAPPING TABLE

INTRODUCTION

Background Concepts

RDBMS atands for "Relational Database Management System." An RDBMS is a DBMS designed specifically for relational databases. Therefore, RDBMSes are a subset of DBMSes.

A relational database refers to a database that stores data in a structured format, using rows and columns. This makes it easy to locate and access specific values within the database. It is "relational" because the values within each table are related to each other. Tables may also be related to other tables. The relational structure makes it possible to run queries across multiple tables at once.

While a relational database describes the type of database an RDMBS manages, the RDBMS refers to the database program itself. It is the software that executes queries on the data, including adding, updating, and searching for values. An RDBMS may also provide a visual representation of the data. For example, it may display data in a table like a spreadsheet, allowing you to view and even edit individual values in the table. Some RDMBS programs allow you to create forms that can streamline entering, editing, and deleting data.

Most well-known DBMS applications fall into the RDBMS category. Examples include Oracle Database, MySQL, Microsoft SQL Server, and IBM DB2. Some of these programs support non-relational databases, but they are primarily used for relational database management.

A relational database is a digital database based on the relational model of data, as proposed by E. F. Codd in 1970. A software system used to maintain relational databases is a relational database management system (RDBMS). Many relational database systems have an option of using the standard SQL (Structured Query Language) for querying and maintaining the database.

Oracle

Oracle Corporation is an American multinational computer technology corporation headquartered in Redwood Shores, California. The company sells database software and technology, cloud engineered systems, and enterprise software products—particularly its own brands of database management systems.

An Oracle **database** is a collection of data treated as a unit. The purpose of a database is to store and retrieve related information. A database server is the key to solving the problems of information management. In general, a **server** reliably manages a large amount of data in a multiuser environment so that many users can concurrently access the same data. All this is accomplished while delivering high performance. A database server also prevents unauthorized access and provides efficient solutions for failure recovery.

Oracle database (Oracle DB) is a relational database management system (RDBMS) from the Oracle Corporation. Originally developed in 1977 by Lawrence Ellison and other developers, Oracle DB is one of the most trusted and widely-used relational database engines.

The system is built around a relational database framework in which data objects may be directly accessed by users (or an application front end) through structured query language (SQL). Oracle is a fully scalable relational database architecture and is often used by global enterprises, which manage and process data across wide and local area networks. The Oracle database has its own network component to allow communications across networks.

Oracle DB is also known as Oracle RDBMS and, sometimes, just Oracle.

Oracle Database is the first database designed for enterprise grid computing, the most flexible and cost effective way to manage information and applications. Enterprise grid computing creates large pools of industry-standard, modular storage and servers. With this architecture, each new system can be rapidly provisioned from the pool of components. There is no need for peak workloads, because capacity can be easily added or reallocated from the resource pools as needed.

The database has **logical structures** and **physical structures**. Because the physical and logical structures are separate, the physical storage of data can be managed without affecting the access to logical storage structures.

A key feature of Oracle is that its architecture is split between the logical and the physical. This structure means that for large-scale distributed computing, also known as grid computing, the data location is irrelevant and transparent to the user, allowing for a more modular physical structure that can be added to and altered without affecting the activity of the database, its data or users. The sharing of resources in this way allows for very flexible data networks whose capacity can be adjusted up or down to suit demand, without degradation of service. It also allows for a robust system to be devised as there is no single point at which a failure can bring down the database, as the networked schema of the storage resources means that any failure would be local only.

Oracle DB editions are hierarchically broken down as follows:

- Enterprise Edition: Offers all features, including superior performance and security, and is the most robust
- Standard Edition: Contains base functionality for users that do not require Enterprise Edition's robust package
- Express Edition (XE): The lightweight, free and limited Windows and Linux edition
- Oracle Lite: For mobile devices

Oracle Enterprise Edition:
Enterprise Edition is the full (top of the range) version or the Oracle Database Server. Options like RAC, Partitioning, Spatial, etc. can be purchased separately to enhance the functionality of the database.

Oracle Standard Edition:
Standard Edition is designed for smaller businesses and enterprises. It offers a subset of the features/ functionality implemented in Enterprise Edition. Database options like Data Guard, Partitioning, Spatial, etc. is not available with Standard Edition (from 10g one can use RAC with Standard Edition). Standard Edition can only be licensed on servers with a maximum capacity of four processors.

Oracle Standard Edition One:
Standard Edition One is a low cost, entry-level version of the Oracle Standard Edition database server. Standard Edition One can only be licensed on small servers with a maximum capacity of two processors.

Oracle Personal Edition:
Personal Oracle is a single user version of the database server. It is mostly the same as Enterprise Edition, but doesn't support advanced options like RAC, Streams, XMLDB, etc.

Oracle XE:
Express Edition (XE) is a free, downloadable version of the Oracle database server. Oracle XE can only be used on single processor machines. It can only manage up to 4 GB of data and 1 GB of memory. ISVs can embed XE in 3rd party products and redistribute it freely.

Oracle Lite:
Oracle Light is a database engine that can be used on mobile platforms like cell phones and PDA's.

Oracle Express Edition (XE)

This practical manual uses Oracle Express Edition (XE) for demonstration of code and queries. The main features of Oracle XE are as follows:

- Oracle Database XE is an entry level database available on Windows and Linux operating systems. XE is built with the same code base as Oracle Database 11g Release 2, so scaling XE to other editions can be easily achieved.

- Oracle Database XE is a good starter database for DBAs and developers who need a free database for training and deployment. **Independent Software Vendors (ISVs)** and hardware vendors can freely distribute Oracle Database XE along with their products, thus adding value to their own products.

- Educational institutions can freely use Oracle Database XE for their curriculum.

The Oracle XE provides wide range of facilities as follows:

Developers:

- Connect Oracle Database to your favorite programming languages and dev environments including Java, .NET, Python, Node.js, Go, PHP, C/C++ and more.
- Learn SQL on the world's leading relational database, or experiment with Oracle's native support for JSON documents and spatial & graph data.
- Use free dev tools and IDEs from Oracle including SQL Developer, SQLcl, and SQL Developer Data Modeler.
- Install free Oracle REST Data Services (ORDS) to REST-enable your database.

- For low-code app development, run Oracle APEX on top of ORDS and XE at no extra cost to rapidly build data-centric web apps that look beautiful in mobile and desktop browsers.

DBAs:

Test drive advancements in Oracle Database that make life easier for DBAs and other administrators using free XE. With XE, any administrator can benefit from playing with many of the advanced features of Oracle Database.

- Manage multiple Oracle Databases in one place with Oracle Multitenant pluggable databases.
- Accelerate database queries using table partitions.
- Get more from database storage with data compression.
- Backup your whole database using Oracle RMAN.
- For security and compliance, encrypt data at rest with Transparent Data Encryption, set database audit policies to track data access, and configure Database Vault to prevent unauthorized access by privileged users.

Data Scientists:

Oracle Database provides data scientists with sheer analytic power, and XE has it all. See what that means.

- Dramatically accelerate queries on large data sets using Oracle In-Memory Column Store.
- Reduce complex analysis to concise SQL statements with Advanced Analytics, including Data Mining SQL.
- Build your analysis graphically in Oracle's free Data Miner UI.
- Quickly load data into your database using Data Pump, SQL*Loader, external tables, or SQL Developer.
- If you prefer R programming, Oracle Database supports that too.

Educators:

Teachers and students can freely use XE for database curriculum and instruction. Students can install it on a laptop to work wherever, whenever - rather than being tethered to a computer lab.

- Suite of courseware available from Oracle Academy.
- No licensing costs.
- Easy to install.
- A full-featured database.

Independent Software Vendors (ISVs)

In need of a database with all of the features and a small footprint, then look no further than XE.

- Embed in your software.
- Distribute with your software.
- Install on customer premises for proof of concept.
- Deliver comprehensive prototypes to your prospects.

Everyone

Oracle Database XE is well suited to users large and small. For example:

- Startups working on a limited budget who need to begin development immediately.

- Non-profits and other organizations who want an Oracle Database, and it does not need to be a fully supported edition.
- Software developers needing to demonstrate their apps to customers.
- Anyone who wants a private sandbox for database evaluation, testing, and proof-of-concept projects.

Installation of Oracle Express Edition (XE)

The Oracle Database XE provides an Oracle database and tools for managing the database.

Oracle Database XE supports the following development environments:

- **Oracle SQL Developer**: Oracle SQL Developer is a graphical version of SQL*Plus that gives database developers a convenient way to perform basic tasks. You can connect to any target Oracle Database XE schema using standard Oracle database authentication. Once connected, you can perform operations on objects in the database.

Download and install Oracle SQL Developer from:
http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html

- **Oracle Application Express**: Oracle Database XE includes Oracle Application Express, a rapid web application development tool for the Oracle database. Oracle Application Express is enabled by default in Oracle Database XE.
- **Java**: Java is an open-source programing language that is designed for use in the distributed environment of the Internet. You can use Oracle JDeveloper, which is a free integrated Java development environment with support for the full development life cycle.

Download and install Oracle JDeveloper from:
http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html

- **.NET and Visual Studio**: Visual Studio is an integrated development environment by Microsoft for building .NET applications. .NET is a software framework for Microsoft Windows operating systems.

Download and install Oracle Data Access Components (ODAC) for Windows from:
http://www.oracle.com/technetwork/topics/dotnet/downloads/index.html

- **PHP**

PHP is an open-source server-side embedded scripting language that is designed for Web development and can be embedded in HTML. You can use the following PHP product:

- **PHP:** Download and install from: http://www.php.net

For more information on Oracle Database XE, see the following:

- Oracle Database XE home page on the Oracle Technology Network:

https://www.oracle.com/database/technologies/xe-downloads.html

- Oracle Database XE Documentation Library:

Click the appropriate link on the Oracle Database XE home page on the Oracle Technology Network; or from the system menus, get to **Oracle Database 11g Express Edition** and select **Get Help**, then **Read Documentation**.

- Discussion forum:

Click the appropriate link on the Oracle Database XE home page on the Oracle Technology Network; or from the system menus, get to **Oracle Database 11g Express Edition** and select **Get Help**, then **Go to Online Forum**.

System Requirements

Oracle provides 32-bit (Windows x86) and 64-bit (Windows x64) versions of Oracle Database XE server and client.

The 32-bit database server runs on 32-bit Windows only. The 64-bit database server and client runs on Windows x64 only. See Table, " Oracle Database XE Requirements for Microsoft Windows 64-bit" for supported operating systems. The 32-bit database client runs on both 32-bit Windows and Windows x64.

Table, " Oracle Database XE Requirements for Microsoft Windows 64-bit" provides system requirements for Oracle Database XE for Microsoft Windows 64-bit

Oracle Database XE Requirements for Microsoft Windows 64-bit

Requirement	Value
System architecture	AMD64 and Intel EM64T
Operating system	One of the following 64-bit Microsoft Windows x64 operating systems: • Windows Server 2008 x64 - Standard, Enterprise, Datacenter, Web, and Foundation Editions • Windows Server 2008 R2 x64 - Standard, • Enterprise, Datacenter, Web, and Foundation Editions. • Windows Server 2012 x64 - Standard, Datacenter, Essentials, and Foundation Editions • Windows Server 2012 R2 x64 - Standard, • Datacenter, Essentials, and Foundation Editions • Windows 7 x64 - Professional, Enterprise, and Ultimate Editions. • Windows 8 - Pro and Enterprise Editions • Windows 8.1 - Pro and Enterprise Editions The Server Core option is not supported.
Network protocol	The following protocols are supported: • IPC • Named Pipes • SDP • TCP/IP • TCP/IP with SSL
Disk space	1.5 gigabytes minimum
RAM	256 megabytes minimum, 512 megabytes recommended for Oracle Database XE. The operating system itself may have a higher minimum requirement.

Installing Oracle Database XE

This section covers the following topics:

- Performing a Graphical User Interface Installation of the Server
- Performing a Silent Installation
- Enabling the Control Panel Services for .NET Stored Procedures and Oracle Services for Microsoft Transaction Server
- Making Oracle Database XE Available to Remote Clients

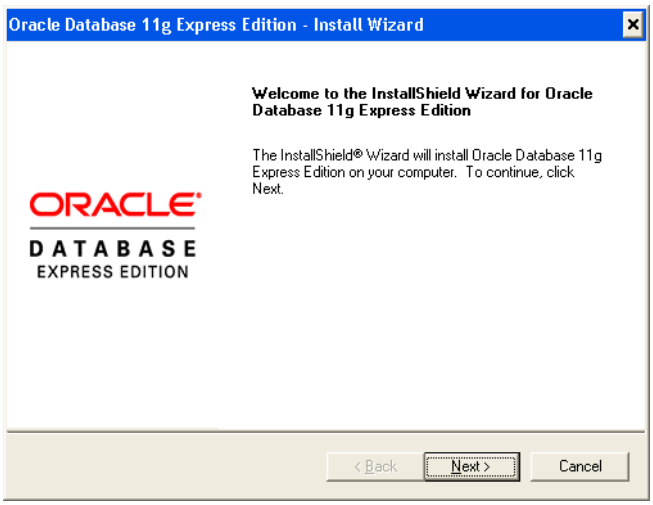
Performing a Graphical User Interface Installation of the Server

Most users will install Oracle Database XE by downloading the installation executable, double-clicking it, and answering graphical user interface prompts as needed.

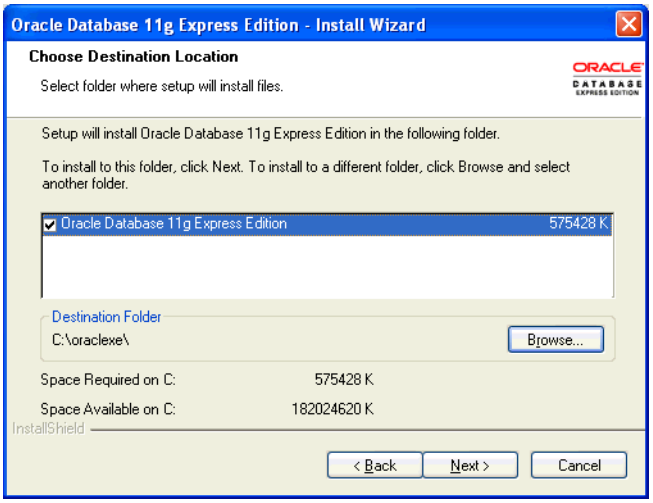
Before attempting to install Oracle Database XE 11.2 uninstall any existing Oracle Database XE or database with the SID XE from the target system.
If you have an existing version of Oracle Database XE, then save your data by exporting it to data files. After you install the new version of Oracle Database XE import this data into the new database. For more information see Section 7, "Importing and Exporting Data between 10.2 XE and 11.2 XE".

To perform a graphical user interface installation:

1. Log on to Windows with Administrative privileges.
   You must be part of the Administrators group on Windows to install Oracle Database XE. If you are logged in as a domain user, ensure that you are connected to the network.
2. If the ORACLE_HOME environment variable has been set, then use **System** in the Control Panel to delete it.
3. Go to the following Web site:
   http://www.oracle.com/technetwork/database/express-edition/downloads/in dex.html
4. Click Free Download and follow the instructions to select and download the Microsoft Windows version of Oracle Database XE.
5. After downloading the Oracle Database XE installation executable, setup.exe, double-click it.
   "Oracle Database XE Character and Language Configurations" on page 16 describes these character sets in detail.
6. In the Oracle Database 11g Express Edition - Install Wizard welcome window, click Next.



7. In the License Agreement window, select **I accept the terms in the license agreement** and then click **Next**.
8. In the Choose Destination Location window, either accept the default or click **Browse** to select a different installation directory. (Do not select a directory that has spaces in its name.) Then click **Next**.



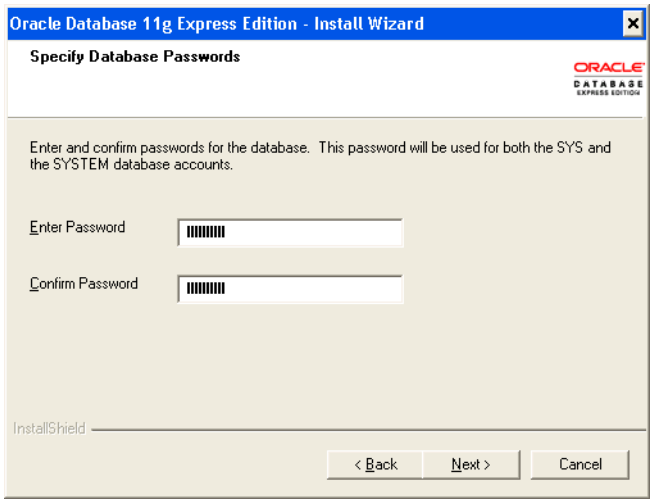
9. If you are prompted for a port number, then specify one.

The following port numbers are the default values:

- 1521: Oracle database listener
- 2030: Oracle Services for Microsoft Transaction Server
- 8080: HTTP port for the Oracle Database XE graphical user interface

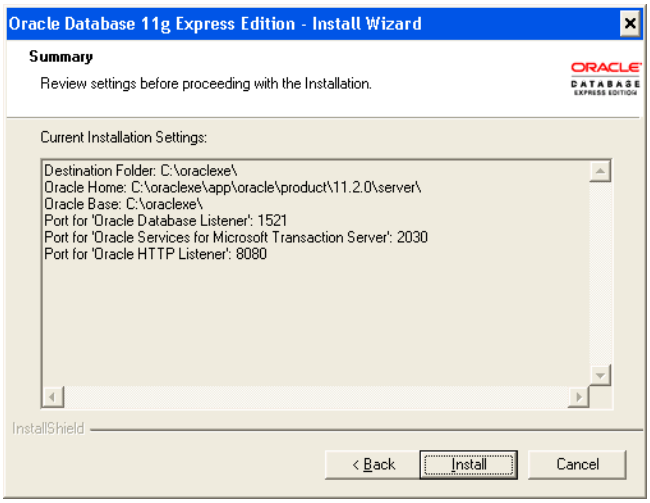
If these port numbers are not currently used, then the installation uses them automatically without prompting you. If they are in use, then you will be prompted to enter an available port number.

10. In the Specify Database Passwords window, enter and confirm the password to use for the SYS and SYSTEM database accounts. Then click Next.



Note: The password for the INTERNAL and ADMIN Oracle Application Express user accounts will be the same as the SYS and SYSTEM administrative user accounts

11. In the Summary window, review the installation settings, and if you are satisfied, click **Install**. Otherwise, click **Back** and modify the settings as necessary.



12. In the InstallShield Wizard Complete window, click **Finish**.

SQL Commends in Oracle Express Edition (XE)

Oracle XE support the following data types:

1. char(size)
2. varchar2(size)
3. date
4. number(p,s)
5. long
6. raw/long raw

Different types of commands in SQL:

A).**DDL commands: -** To create a database objects
B).**DML commands: -** To manipulate data of a database objects
C).**DQL command: -** To retrieve the data from a database.
D).**DCL/DTL commands: -** To control the data of a database…

DDL commands:

1. The Create Table Command: - it defines each column of the table uniquely. Each column has minimum of three attributes, a name , data type and size.

Syntax:
Create table <table name> (<col1> <datatype>(<size>),<col2> <datatype><size>));

Ex:
create table emp(empno number(4) primary key, ename char(10));

2. Modifying the structure of tables.
a)add new columns

Syntax:
Alter table <tablename> add(<new col><datatype(size),<new col>datatype(size));

Ex:
alter table emp add(sal number(7,2));

3. Dropping a column from a table.

Syntax:
Alter table <tablename> drop column <col>;

Ex:
alter table emp drop column sal;

4. Modifying existing columns.

Syntax:
Alter table <tablename> modify(<col><newdatatype>(<newsize>));

Ex:
alter table emp modify(ename varchar2(15));

5. Renaming the tables

Syntax:
Rename <oldtable> to <new table>;

Ex:
rename emp to emp1;

6. truncating the tables.

Syntax:
Truncate table <tablename>;

Ex:
trunc table emp1;

7. Destroying tables.

Syntax:
Drop table <tablename>;

Ex:
drop table emp;

DML commands:

8. Inserting Data into Tables: - once a table is created the most natural thing to do is load this table with data to be manipulated later.

Syntax:

insert into <tablename> (<col1>,<col2>) values(<exp>,<exp>);

9. Delete operations.

a) remove all rows
Syntax:
delete from <tablename>;

b) removal of a specified row/s
Syntax:
delete from <tablename> where <condition>;

10. Updating the contents of a table.

a) updating all rows
Syntax:
Update <tablename> set <col>=<exp>,<col>=<exp>;

b) updating seleted records.
Syntax:
Update <tablename> set <col>=<exp>,<col>=<exp> where <condition>;

11. Types of data constrains.
a) not null constraint at column level.
Syntax:
<col><datatype>(size)not null

b) unique constraint
Syntax:
Unique constraint at column level.
<col><datatype>(size)unique;

c) unique constraint at table level:
Syntax:
Create table tablename(col=format,col=format,unique(<col1>,<col2>);

d) primary key constraint at column level
Syntax:
<col><datatype>(size)primary key;

e) primary key constraint at table level.
Syntax:
Create table tablename(col=format,col=format
primary key(col1>,<col2>);

f) foreign key constraint at column level.
Syntax:
<col><datatype>(size>) references <tablename>[<col>];

g) foreign key constraint at table level
Syntax:
foreign key(<col>[,<col>])references <tablename>[(<col>,<col>)

h) check constraint

check constraint constraint at column level.
Syntax: <col><datatype>(size) check(<logical expression>)

i) check constraint constraint at table level.
Syntax: check(<logical expression>)

DQL Commands:

12. Viewing data in the tables: - once data has been inserted into a table, the next most logical operation would be to view what has been inserted.

a) all rows and all columns
Syntax:
Select <col> to <col n> from tablename;

Select * from tablename;

13. Filtering table data: - while viewing data from a table, it is rare that all the data from table will be required each time. Hence, sql must give us a method of filtering out data that is not required data.

a) Selected columns and all rows:
Syntax:
select <col1>,<col2> from <tablename>;

b) selected rows and all columns:
Syntax:
select * from <tablename> where <condition>;

c) selected columns and selected rows
Syntax:
select <col1>,<col2> from <tablename> where<condition>;

14. Sorting data in a table.
Syntax:
Select * from <tablename> order by <col1>,<col2> <[sortorder]>;

DCL commands:
Oracle provides extensive feature in order to safeguard information stored in its tables from unauthoraised viewing and damage.The rights that allow the user of some or all oracle resources on the server are called privileges.

a) Grant privileges using the GRANT statement

The grant statement provides various types of access to database objects such as tables,views and sequences and so on.

Syntax:
GRANT <object privileges>
ON <objectname>
TO<username>
[WITH GRANT OPTION];

b) Reoke permissions using the REVOKE statement:

The REVOKE statement is used to deny the Grant given on an object.

Syntax:
REVOKE<object privilege>
ON
FROM<user name>;

BLOCK 1: TABLE MANIPULATION

1. Table Creation

The CREATE TABLE statement is used to create a new table in a database.

Query Syntax:

CREATE TABLE *table_name* (
column1 *datatype*,
column2 *datatype*,
column3 *datatype*,
....);

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar2, number, date, etc.).

Query Example:

To create a Table to store the details of a Persons such as, *Id, Name, Address, City, Mobile Number*

CREATE TABLE Persons (
PersonID number(4),
Name varchar2(25),
Address varchar2(75),
City varchar2(50),
Mobile number(10));

SQL> create table persons (personID number(4), Name varchar2(25), address varchar2(75), city varchar2(50), mobile number(10));

SQL> table created.

Try your Own:

- Create a table with student details
- Create a table with customer details
- Create a table with employee details
- Create a table with product details

Inserting rows in Table

Query Syntax (insert one rows):

INSERT INTO table_name VALUES (value_list);

Query Example:

SQL> insert into persons values (1001, 'Rajesh', '35 South St', 'Karaikudi', 9876896512);

1 row created.

SQL>

Query Syntax (insert bulk rows):

INSERT INTO table_name VALUES (value_list);

Query Example:

SQL> insert into persons values (&personID, '&name', '&address', '&city', &mobile);

Enter value for personID : 1002
Enter value for name: Kumar
Enter value for address: 45 Market Road
Enter value for city: Chennai
Enter value for mobile: 8765349210

insert into persons values (&personID, '&name', '&address', '&city', &mobile);

insert into persons values (1002, 'Kumar', ' 45 Market Road', 'Chennai', 8765349210);

1 row created.

SQL>

Note: To add further rows simply use '/' in the SQL prompt. Actually, this '/' command will again execute the previously entered command.

PersonID	Name	Address	City	Mobile
1001	Rajesh	35 South St	Karaikudi	9876896512
1002	Kumar	45 Market Road	Chennai	8765349210
1243	John	371 Thomas Mt.	Chennai	7893451260
1378	Rani	875 Mount Rd.	Chennai	9654712341
1567	Sundari	67 Main Road	Rajapalayam	6387348987

2. Rename a Table

RENAME TABLE allows you to rename an existing table in any schema (except the schema SYS). To rename a table, you must either be the database owner or the table owner.

Query Syntax:

RENAME TABLE *table-Name* TO *new-Table-Name*;

Query Example:

SQL> rename table persons_chennai to persons_city;

3. Creating Table using existing Table / Copying another table:

Query Syntax:

CREATE TABLE *table_name as select attribute_list from table_name [constraint]*);

Query Example:

SQL> create table persons_chennai as select * from persons1 where city='Chennai';

SQL> table created.

4. Dropping a Table:

Query Syntax:

DROP TABLE *table_name*;

Query Example:

SQL> drop table persons1;

Note:

- The DROP TABLE statement is used to drop an existing table in a database.
- The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.
    - Syntax: TRUNCATE TABLE table_name;

Keys and Constraints

- **Primary Key** is a column (or attribute) in a table (or a relation) that uniquely identifies every row (or tuple) through the table. And you can define only one PK on a table.

- **Unique Key** is similar to a Primary Key, but it allows a NULL value. And you can have multiple Unique Keys on a table.

- **NOT NULL** constraint restricts a column from having a NULL value. Once **NOT NULL** constraint is applied to a column, you cannot pass a null value to that column. It enforces a column to contain a proper value.

Table creation with keys:

Query Syntax:

CREATE TABLE *table_name* (
column1 *datatype constrint, ...*);

Query Example:

To create a Table to store the details of a Persons1 such as, *Id, Name, Address, City, Mobile Number* with appropriate constraints

```
CREATE TABLE Persons1 (
  PersonID    number(4) primary key,
  Name        varchar2(25) not null,
  Address     varchar2(75),
  City        varchar2(50),
  Mobile      number(10) unique);
```

From the above syntax, the *PersonID* is declared as 'Primary Key', *Name* is enforced with 'Not Null' and *Mobile* with 'Unique' constraints.

SQL> create table persons (personID number(4) primary key, Name varchar2(25) not null, address varchar2(75), city varchar2(50), mobile number(10) unique);

SQL> table created.

5. Describing Table Definitions

To view the structure of the created table.

SQL> desc persons;

Output:

```
Name                                      Null?    Type
----------------------------------------- -------- ----------------
PERSONID                                           NUMBER(4)
NAME                                               VARCHAR2(25)
ADDRESS                                            VARCHAR2(75)
CITY                                               VARCHAR2(50)
```

MOBILE NUMBER(10)

Notes

6. Modifying Tables

ALTER TABLE Statement

This Oracle tutorial explains how to use the Oracle **ALTER TABLE statement** to add a column, modify a column, drop a column, rename a column or rename a table (with syntax, examples and practice exercises).

Description

The Oracle ALTER TABLE statement is used to add, modify, or drop/delete columns in a table. The Oracle ALTER TABLE statement is also used to rename a table.

Add column in table

Syntax

To ADD A COLUMN in a table, the Oracle ALTER TABLE syntax is:

```
ALTER TABLE table_name
  ADD column_name column_definition;
```

Example

Let's look at an example that shows how to add a column in an Oracle table using the ALTER TABLE statement.

For example:

```
ALTER TABLE customers
  ADD customer_name varchar2(45);
```

This Oracle ALTER TABLE example will add a column called *customer_name* to the *customers* table that is a data type of varchar2(45).

In a more complicated example, you could use the ALTER TABLE statement to add a new column that also has a default value:

```
ALTER TABLE customers
  ADD city varchar2(40) DEFAULT 'Karaikudi';
```

In this example, the column called *city* has been added to the *customers* table with a data type of varchar2(40) and a default value of 'Karaikudi'.

Add multiple columns in table

Syntax

To ADD MULTIPLE COLUMNS to an existing table, the Oracle ALTER TABLE syntax is:

```
ALTER TABLE table_name
  ADD (column_1 column_definition,
      column_2 column_definition,
      ...
      column_n column_definition);
```

Example

Let's look at an example that shows how to add multiple columns in an Oracle table using the ALTER TABLE statement.

For example:

```
ALTER TABLE customers
  ADD (customer_name varchar2(45),
      city varchar2(40) DEFAULT 'Karaikudi');
```

This Oracle ALTER TABLE example will add two columns, *customer_name* as a varchar2(45) field and *city* as a varchar2(40) field with a default value of 'Karaikudi' to the *customers* table.

Modify column in table

Syntax

To MODIFY A COLUMN in an existing table, the Oracle ALTER TABLE syntax is:

```
ALTER TABLE table_name
  MODIFY column_name column_type;
```

Example

Let's look at an example that shows how to modify a column in an Oracle table using the ALTER TABLE statement.

For example:

```
ALTER TABLE customers
  MODIFY customer_name varchar2(100) NOT NULL;
```

This Oracle ALTER TABLE example will modify the column called *customer_name* to be a data type of varchar2(100) and force the column to not allow null values.

In a more complicated example, you could use the ALTER TABLE statement to add a default value as well as modify the column definition:

```
ALTER TABLE customers
  MODIFY city varchar2(75) DEFAULT 'Karaikudi' NOT NULL;
```

In this example, the ALTER TABLE statement would modify the column called *city* to be a data type of varchar2(75), the default value would be set to 'Karaikudi' and the column would be set to not allow null values.

Modify Multiple columns in table

Syntax

To MODIFY MULTIPLE COLUMNS in an existing table, the Oracle ALTER TABLE syntax is:

```
ALTER TABLE table_name
  MODIFY (column_1 column_type,
        column_2 column_type,          ...
        column_n column_type);
```

Example

Let's look at an example that shows how to modify multiple columns in an Oracle table using the ALTER TABLE statement.

For example:

```
ALTER TABLE customers
  MODIFY (customer_name varchar2(100) NOT NULL,
        city varchar2(75) DEFAULT 'Karaikudi' NOT NULL);
```

This Oracle ALTER TABLE example will modify both the *customer_name* and *city* columns. The *customer_name* column will be set to a varchar2(100) data type and not allow null values. The *city* column will be set to a varchar2(75) data type, its default value will be set to 'Karaikudi', and the column will not allow null values.

Drop column in table

Syntax

To DROP A COLUMN in an existing table, the Oracle ALTER TABLE syntax is:

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

Example

Let's look at an example that shows how to drop a column in an Oracle table using the ALTER TABLE statement.

For example:

```
ALTER TABLE customers
```

```
DROP COLUMN customer_name;
```

This Oracle ALTER TABLE example will drop the column called *customer_name* from the table called *customers*.

Rename column in table

Syntax

Starting in Oracle 9i Release 2, you can now rename a column.

To RENAME A COLUMN in an existing table, the Oracle ALTER TABLE syntax is:

```
ALTER TABLE table_name
  RENAME COLUMN old_name TO new_name;
```

Example

Let's look at an example that shows how to rename a column in an Oracle table using the ALTER TABLE statement.

For example:

```
ALTER TABLE customers
  RENAME COLUMN customer_name TO cname;
```

This Oracle ALTER TABLE example will rename the column called *customer_name* to *cname*.

Rename table

Syntax

To RENAME A TABLE, the Oracle ALTER TABLE syntax is:

```
ALTER TABLE table_name
  RENAME TO new_table_name;
```

Example

Let's look at an example that shows how to rename a table in Oracle using the ALTER TABLE statement.

For example:

```
ALTER TABLE customers
  RENAME TO contacts;
```

This Oracle ALTER TABLE example will rename the *customers* table to *contacts*.

Practice Exercise #1:

Based on the *departments* table below, rename the *departments* table to *depts*.

```
CREATE TABLE departments
( department_id number(10) NOT NULL,
  department_name varchar2(50) NOT NULL,
  CONSTRAINT departments_pk PRIMARY KEY (department_id)
);
```

Solution for Practice Exercise #1:

The following Oracle ALTER TABLE statement would rename the *departments* table to *depts*:

```
ALTER TABLE departments
  RENAME TO depts;
```

Practice Exercise #2:

Based on the *employees* table below, add a column called *bonus* that is a number(6) datatype.

```
CREATE TABLE employees
( employee_number number(10) NOT NULL,
  employee_name varchar2(50) NOT NULL,
  department_id number(10),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number));
```

Solution for Practice Exercise #2:

The following Oracle ALTER TABLE statement would add a *bonus* column to the *employees* table:

```
ALTER TABLE employees
  ADD bonus number(6);
```

Practice Exercise #3:

Based on the *customers* table below, add two columns - one column called *contact_name* that is a varchar2(50) datatype and one column called *last_contacted* that is a date datatype.

```
CREATE TABLE customers
( customer_id number(10) NOT NULL,
  customer_name varchar2(50) NOT NULL,
  address varchar2(50),
  city varchar2(50),
  state varchar2(25),
  zip_code varchar2(10),
  CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);
```

Solution for Practice Exercise #3:

The following Oracle ALTER TABLE statement would add the *contact_name* and *last_contacted* columns to the *customers* table:

```
ALTER TABLE customers
  ADD (contact_name varchar2(50),
      last_contacted date);
```

Practice Exercise #4:

Based on the *employees* table below, change the *employee_name* column to a varchar2(75) datatype.

```
CREATE TABLE employees
( employee_number number(10) NOT NULL,
  employee_name varchar2(50) NOT NULL,
  department_id number(10),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number));
```

Solution for Practice Exercise #4:

The following Oracle ALTER TABLE statement would change the datatype for the *employee_name* column to varchar2(75):

```
ALTER TABLE employees
  MODIFY employee_name varchar2(75);
```

Practice Exercise #5:

Based on the *customers* table below, change the *customer_name* column to NOT allow null values and change the *state* column to a varchar2(2) datatype.

```
CREATE TABLE customers
( customer_id number(10) NOT NULL,
  customer_name varchar2(50),
  address varchar2(50),
  city varchar2(50),
  state varchar2(25),
  zip_code varchar2(10),
  CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);
```

Solution for Practice Exercise #5:

The following Oracle ALTER TABLE statement would modify the *customer_name* and *state* columns accordingly in the *customers* table:

```
ALTER TABLE customers
  MODIFY (customer_name varchar2(50) NOT NULL,
        state varchar2(2));
```

Practice Exercise #6:

Based on the *employees* table below, drop the *salary* column.

```
CREATE TABLE employees
( employee_number number(10) NOT NULL,
  employee_name varchar2(50) NOT NULL,
  department_id number(10),
  salary number(6),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number)
);
```

Solution for Practice Exercise #6:

The following Oracle ALTER TABLE statement would drop the *salary* column from the *employees* table:

```
ALTER TABLE employees
  DROP COLUMN salary;
```

Practice Exercise #7:

Based on the *departments* table below, rename the *department_name* column to *dept_name*.

```
CREATE TABLE departments
( department_id number(10) NOT NULL,
  department_name varchar2(50) NOT NULL,
  CONSTRAINT departments_pk PRIMARY KEY (department_id)
);
```

Solution for Practice Exercise #7:

The following Oracle ALTER TABLE statement would rename the *department_name* column to *dept_name* in the *departments* table:

```
ALTER TABLE departments
  RENAME COLUMN department_name TO dept_name;
```

7. Joining Tables

This Oracle tutorial explains how to use **JOINS** (inner and outer) in Oracle with syntax, visual illustrations, and examples.

Description

Oracle JOINS are used to retrieve data from multiple tables. An Oracle JOIN is performed whenever two or more tables are joined in a SQL statement.

There are 4 different types of Oracle joins:

- Oracle INNER JOIN (or sometimes called simple join)
- Oracle LEFT OUTER JOIN (or sometimes called LEFT JOIN)
- Oracle RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)
- Oracle FULL OUTER JOIN (or sometimes called FULL JOIN)

So let's discuss Oracle JOIN syntax, look at visual illustrations of Oracle JOINS, and explore Oracle JOIN examples.

INNER JOIN (simple join)

Chances are, you've already written a statement that uses an Oracle INNER JOIN. It is the most common type of join. Oracle INNER JOINS return all rows from multiple tables where the join condition is met.

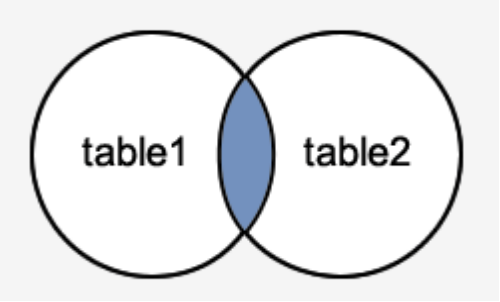
Syntax

The syntax for the INNER JOIN in Oracle/PLSQL is:

```
SELECT columns
FROM table1
INNER JOIN table2 ON table1.column = table2.column;
```

Visual Illustration

In this visual diagram, the Oracle INNER JOIN returns the shaded area:



The Oracle INNER JOIN would return the records where *table1* and *table2* intersect.

Example

Here is an example of an Oracle INNER JOIN:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
INNER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

This Oracle INNER JOIN example would return all rows from the suppliers and orders tables where there is a matching supplier_id value in both the suppliers and orders tables.

Let's look at some data to explain how the INNER JOINS work:

We have a table called *suppliers* with two fields (supplier_id and supplier_name). It contains the following data:

supplier_id	supplier_name
10000	IBM
10001	Hewlett Packard

supplier_id	supplier_name
10002	Microsoft
10003	NVIDIA

We have another table called *orders* with three fields (order_id, supplier_id, and order_date). It contains the following data:

order_id	supplier_id	order_date
500125	10000	2003/05/12
500126	10001	2003/05/13
500127	10004	2003/05/14

If we run the Oracle SELECT statement (that contains an INNER JOIN) below:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
INNER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

supplier_id	name	order_date
10000	IBM	2003/05/12
10001	Hewlett Packard	2003/05/13

The rows for *Microsoft* and *NVIDIA* from the supplier table would be omitted, since the supplier_id's 10002 and 10003 do not exist in both tables. The row for 500127 (order_id) from the orders table would be omitted, since the supplier_id 10004 does not exist in the suppliers table.

LEFT OUTER JOIN

Another type of join is called an Oracle LEFT OUTER JOIN. This type of join returns all rows from the LEFT-hand table specified in the ON condition and **only** those rows from the other table where the joined fields are equal (join condition is met).

Syntax

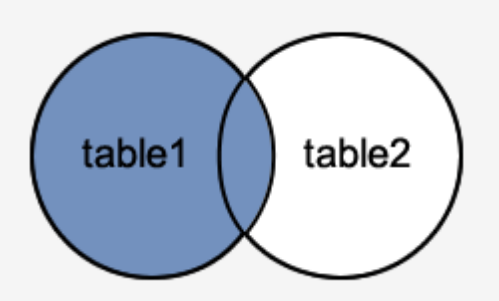
The syntax for the Oracle **LEFT OUTER JOIN** is:

```
SELECT columns
FROM table1 LEFT [OUTER] JOIN table2
ON table1.column = table2.column;
```

In some databases, the LEFT OUTER JOIN keywords are replaced with LEFT JOIN.

Visual Illustration

In this visual diagram, the Oracle LEFT OUTER JOIN returns the shaded area:



The Oracle LEFT OUTER JOIN would return the all records from *table1* and only those records from *table2* that intersect with *table1*.

Example

Here is an example of an Oracle LEFT OUTER JOIN:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
LEFT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

This LEFT OUTER JOIN example would return all rows from the suppliers table and only those rows from the orders table where the joined fields are equal.

If a supplier_id value in the suppliers table does not exist in the orders table, all fields in the orders table will display as <null> in the result set.

Let's look at some data to explain how LEFT OUTER JOINS work:

We have a table called *suppliers* with two fields (supplier_id and supplier_name). It contains the following data:

supplier_id	supplier_name
10000	IBM
10001	Hewlett Packard
10002	Microsoft
10003	NVIDIA

We have a second table called *orders* with three fields (order_id, supplier_id, and order_date). It contains the following data:

order_id	supplier_id	order_date
500125	10000	2003/05/12

order_id	supplier_id	order_date
500126	10001	2003/05/13

If we run the SELECT statement (that contains a LEFT OUTER JOIN) below:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
LEFT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

supplier_id	supplier_name	order_date
10000	IBM	2003/05/12
10001	Hewlett Packard	2003/05/13
10002	Microsoft	<null>
10003	NVIDIA	<null>

The rows for *Microsoft* and *NVIDIA* would be included because a LEFT OUTER JOIN was used. However, you will notice that the order_date field for those records contains a <null> value.

RIGHT OUTER JOIN

Another type of join is called an Oracle RIGHT OUTER JOIN. This type of join returns all rows from the RIGHT-hand table specified in the ON condition and **only** those rows from the other table where the joined fields are equal (join condition is met).

Syntax

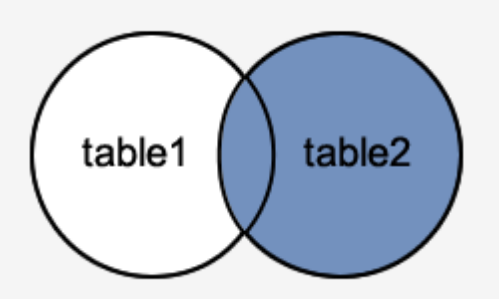
The syntax for the Oracle **RIGHT OUTER JOIN** is:

```
SELECT columns
FROM table1
RIGHT [OUTER] JOIN table2
ON table1.column = table2.column;
```

In some databases, the RIGHT OUTER JOIN keywords are replaced with RIGHT JOIN.

Visual Illustration

In this visual diagram, the Oracle RIGHT OUTER JOIN returns the shaded area:



The Oracle RIGHT OUTER JOIN would return the all records from *table2* and only those records from *table1* that intersect with *table2*.

Example

Here is an example of an Oracle RIGHT OUTER JOIN:

```
SELECT orders.order_id, orders.order_date, suppliers.supplier_name
FROM suppliers
RIGHT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

This RIGHT OUTER JOIN example would return all rows from the orders table and only those rows from the suppliers table where the joined fields are equal.

If a supplier_id value in the orders table does not exist in the suppliers table, all fields in the suppliers table will display as <null> in the result set.

Let's look at some data to explain how RIGHT OUTER JOINS work:

We have a table called *suppliers* with two fields (supplier_id and supplier_name). It contains the following data:

supplier_id	supplier_name
10000	Apple
10001	Google

We have a second table called *orders* with three fields (order_id, supplier_id, and order_date). It contains the following data:

order_id	supplier_id	order_date
500125	10000	2013/08/12
500126	10001	2013/08/13
500127	10002	2013/08/14

If we run the SELECT statement (that contains a RIGHT OUTER JOIN) below:

```
SELECT orders.order_id, orders.order_date, suppliers.supplier_name
FROM suppliers
RIGHT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

order_id	order_date	supplier_name
500125	2013/08/12	Apple
500126	2013/08/13	Google
500127	2013/08/14	<null>

The row for *500127* (order_id) would be included because a RIGHT OUTER JOIN was used. However, you will notice that the supplier_name field for that record contains a <null> value.

FULL OUTER JOIN

Another type of join is called an Oracle FULL OUTER JOIN. This type of join returns all rows from the LEFT-hand table and RIGHT-hand table with nulls in place where the join condition is not met.

Syntax

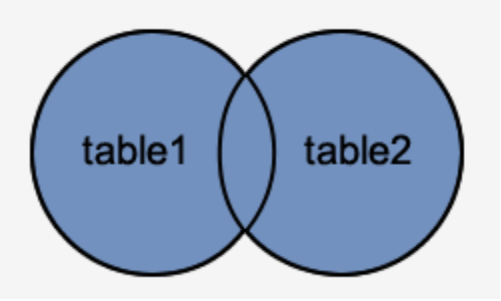
The syntax for the Oracle **FULL OUTER JOIN** is:

```
SELECT columns
FROM table1
FULL [OUTER] JOIN table2
ON table1.column = table2.column;
```

In some databases, the FULL OUTER JOIN keywords are replaced with FULL JOIN.

Visual Illustration

In this visual diagram, the Oracle FULL OUTER JOIN returns the shaded area:



The Oracle FULL OUTER JOIN would return the all records from both *table1* and *table2*.

Example
Here is an example of an Oracle FULL OUTER JOIN:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers FULL OUTER JOIN orders ON suppliers.supplier_id =
orders.supplier_id;
```

This FULL OUTER JOIN example would return all rows from the suppliers table and all rows from the orders table and whenever the join condition is not met, <nulls> would be extended to those fields in the result set.

If a supplier_id value in the suppliers table does not exist in the orders table, all fields in the orders table will display as <null> in the result set. If a supplier_id value in the orders table does not exist in the suppliers table, all fields in the suppliers table will display as <null> in the result set.

Let's look at some data to explain how FULL OUTER JOINS work:

We have a table called *suppliers* with two fields (supplier_id and supplier_name). It contains the following data:

supplier_id	supplier_name
10000	IBM
10001	Hewlett Packard
10002	Microsoft
10003	NVIDIA

We have a second table called *orders* with three fields (order_id, supplier_id, and order_date). It contains the following data:

order_id	supplier_id	order_date
500125	10000	2013/08/12
500126	10001	2013/08/13
500127	10004	2013/08/14

If we run the SELECT statement (that contains a FULL OUTER JOIN) below:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
FULL OUTER JOIN orders ON suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

supplier_id	supplier_name	order_date
10000	IBM	2013/08/12
10001	Hewlett Packard	2013/08/13

supplier_id	supplier_name	order_date
10002	Microsoft	<null>
10003	NVIDIA	<null>
<null>	<null>	2013/08/14

The rows for *Microsoft* and *NVIDIA* would be included because a FULL OUTER JOIN was used. However, you will notice that the order_date field for those records contains a <null> value.

The row for supplier_id 10004 would be also included because a FULL OUTER JOIN was used. However, you will notice that the supplier_id and supplier_name field for those records contain a <null> value.

8. Number and Date Functions

Numeric Functions are used to perform operations on numbers and return numbers.

Following are the numeric functions defined in SQL:

1. **ABS():** It returns the absolute value of a number.
   Syntax: SELECT ABS(-243.5);
   Output: 243.5

2. **ACOS():** It returns the cosine of a number.
   Syntax: SELECT ACOS(0.25);
   Output: 1.318116071652818

3. **ASIN():** It returns the arc sine of a number.
   Syntax: SELECT ASIN(0.25);
   Output: 0.25268025514207865

4. **ATAN():** It returns the arc tangent of a number.
   Syntax: SELECT ATAN(2.5);
   Output: 1.1902899496825317

5. **CEIL():** It returns the smallest integer value that is greater than or equal to a number.
   Syntax: SELECT CEIL(25.75);
   Output: 26

6. **CEILING():** It returns the smallest integer value that is greater than or equal to a number.
   Syntax: SELECT CEILING(25.75);
   Output: 26

7. **COS():** It returns the cosine of a number.
   Syntax: SELECT COS(30);
   Output: 0.15425144988758405

8. **COT():** It returns the cotangent of a number.
   Syntax: SELECT COT(6);
   Output: -3.436353004180128

9. **DEGREES():** It converts a radian value into degrees.
   Syntax: SELECT DEGREES(1.5);
   Output: 85.94366926962348

10. **DIV():** It is used for integer division.
    Syntax: SELECT 10 DIV 5;
    Output: 2

11. **EXP():** It returns e raised to the power of number.
    Syntax: SELECT EXP(1);
    Output: 2.718281828459045

12. **FLOOR():** It returns the largest integer value that is less than or equal to a number.
    Syntax: SELECT FLOOR(25.75);
    Output: 25

13. **GREATEST():** It returns the greatest value in a list of expressions.
    Syntax: SELECT GREATEST(30, 2, 36, 81, 125);
    Output: 125

14. **LEAST():** It returns the smallest value in a list of expressions.
    Syntax: SELECT LEAST(30, 2, 36, 81, 125);
    Output: 2

15. **LN():** It returns the natural logarithm of a number.
    Syntax: SELECT LN(2);
    Output: 0.6931471805599453

16. **LOG10():** It returns the base-10 logarithm of a number.
    Syntax: SELECT LOG(2);
    Output: 0.6931471805599453

17. **LOG2():** It returns the base-2 logarithm of a number.
    Syntax: SELECT LOG2(6);
    Output: 2.584962500721156

18. **MOD():** It returns the remainder of n divided by m.
    Syntax: SELECT MOD(18, 4);
    Output: 2

19. **PI():** It returns the value of PI displayed with 6 decimal places.
    Syntax: SELECT PI();
    Output: 3.141593

20. **POW():** It returns m raised to the nth power.
    Syntax: SELECT POW(4, 2);
    Output: 16

21. **RADIANS():** It converts a value in degrees to radians.
    Syntax: SELECT RADIANS(180);
    Output: 3.141592653589793

22. **RAND():** It returns a random number.

Syntax: SELECT RAND();
Output: 0.33623238684258644

23. **ROUND():** It returns a number rounded to a certain number of decimal places.
Syntax: SELECT ROUND(5.553);
Output: 6

24. **SIGN():** It returns a value indicating the sign of a number.
Syntax: SELECT SIGN(255.5);
Output: 1

25. **SIN():** It returns the sine of a number.
Syntax: SELECT SIN(2);
Output: 0.9092974268256817

26. **SQRT():** It returns the square root of a number.
Syntax: SELECT SQRT(25);
Output: 5

27. **TAN():** It returns the tangent of a number.
Syntax: SELECT TAN(1.75);
Output: -5.52037992250933

28. **ATAN2():** It returns the arctangent of the x and y coordinates, as an angle and expressed in radians.
Syntax: SELECT ATAN2(7);
Output: 1.42889927219073

29. **TRUNCATE():** It returns 7.53635 truncated to 2 places right of the decimal point.
Syntax: SELECT TRUNCATE(7.53635, 2);
Output: 7.53

Date Functions

Function	Example	Result	Description
ADD_MONTHS	ADD_MONTHS(DATE '2016-02-29', 1)	31-MAR-16	Add a number of months (n) to a date and return the same day which is n of months away.
CURRENT_DATE	SELECT CURRENT_DATE FROM dual	06-AUG-2017 19:43:44	Return the current date and time in the session time zone
CURRENT_TIMESTAMP	SELECT CURRENT_TIMESTAMP FROM dual	06-AUG-17 08.26.52.742000000 PM -**07:00**	Return the current date and time with time zone in the session time zone
DBTIMEZONE	SELECT DBTIMEZONE FROM dual;	-07:00	Get the current database time zone
EXTRACT	EXTRACT(YEAR FROM SYSDATE)	2017	Extract a value of a date time field e.g., YEAR,

Function	Example	Result	Description
			MONTH, DAY, … from a date time value.
FROM_TZ	FROM_TZ(TIMESTAMP '2017-08-08 08:09:10', '-09:00')	08-AUG-17 08.09.10.000000000 AM -07:00	Convert a timestamp and a time zone to a TIMESTAMP WITH TIME ZONE value
LAST_DAY	LAST_DAY(DATE '2016-02-01')	29-FEB-16	Gets the last day of the month of a specified date.
LOCALTIMESTAMP	SELECT LOCALTIMESTAMP FROM dual	06-AUG-17 08.26.52.742000000 PM	Return a TIMESTAMP value that represents the current date and time in the session time zone.
MONTHS_BETWEEN	MONTHS_BETWEEN(DATE '2017-07-01', DATE '2017-01-01')	6	Return the number of months between two dates.
NEW_TIME	NEW_TIME(TO_DATE('08-07-2017 01:30:45', 'MM-DD-YYYY HH24:MI:SS'), 'AST', 'PST')	06-AUG-2017 21:30:45	Convert a date in one time zone to another
NEXT_DAY	NEXT_DAY(DATE '2000-01-01', 'SUNDAY')	02-JAN-00	Get the first weekday that is later than a specified date.
ROUND	ROUND(DATE '2017-07-16', 'MM')	01-AUG-17	Return a date rounded to a specific unit of measure.
SESSIONTIMEZONE	SELECT SESSIONTIMEZONE FROM dual;	-07:00	Get the session time zone
SYSDATE	SYSDATE	01-AUG-17	Return the current system date and time of the operating system where the Oracle Database resides.
SYSTIMESTAMP	SELECT SYSTIMESTAMP FROM dual;	01-AUG-17 01.33.57.929000000 PM -07:00	Return the system date and time that includes fractional seconds and time zone.

Function	Example	Result	Description
TO_CHAR	TO_CHAR(DATE'2017-01-01', 'DL')	Sunday, January 01, 2017	Convert a DATE or an INTERVAL value to a character string in a specified format.
TO_DATE	TO_DATE('01 Jan 2017', 'DD MON YYYY')	01-JAN-17	Convert a date which is in the character string to a DATE value.
TRUNC	TRUNC(DATE '2017-07-16', 'MM')	01-JUL-17	Return a date truncated to a specific unit of measure.
TZ_OFFSET	TZ_OFFSET('Europe/London')	+01:00	Get time zone offset of a time zone name from UTC

BLOCK 2 : SQL QUERIES AND SUB QUERIES

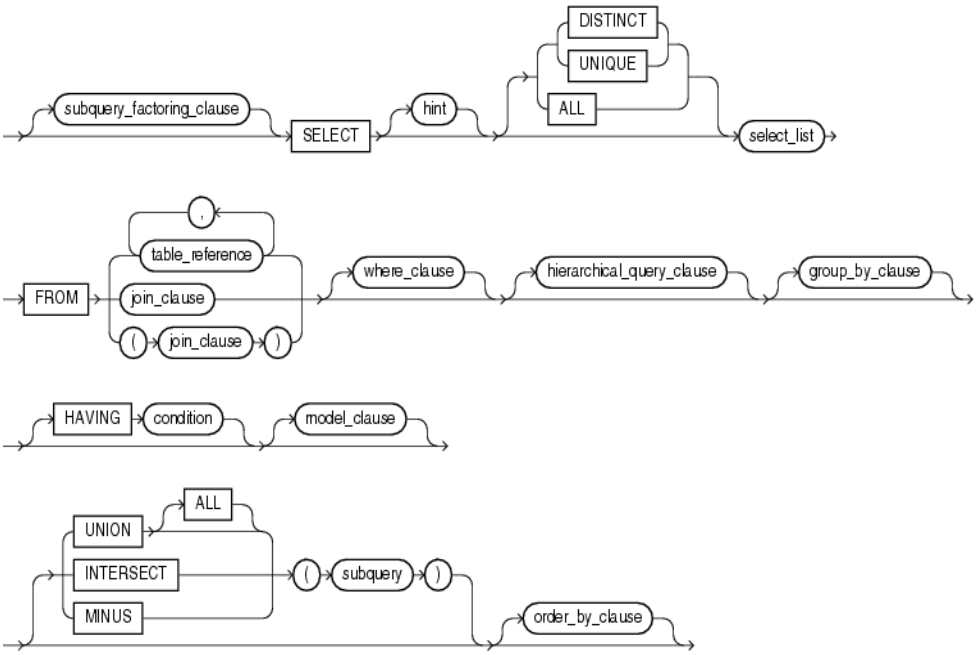
Notes

A **query** is an operation that retrieves data from one or more tables or views. In this reference, a top-level SELECT statement is called a **query**, and a query nested within another SQL statement is called a **subquery**.

This section describes some types of queries and subqueries and how to use them. The top level of the syntax is shown in this chapter. Please refer to SELECT for the full syntax of all the clauses and the semantics of this statement.

***select*::=**

***subquery*::=**



In Oracle, a subquery is a query within a query. You can create subqueries within your SQL statements. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

A subquery is best defined as a query within a query. Subqueries enable you to write queries that select data rows for criteria that are actually developed while the query is executing at run time. More formally, it is the use of a SELECT statement inside one of the clauses of another SELECT statement.

Notes

In fact, a subquery can be contained inside another subquery, which is inside another subquery, and so forth. A subquery can also be nested inside INSERT, UPDATE, and DELETE statements. Subqueries must be enclosed within parentheses.

A subquery can be used any place where an expression is allowed providing it returns a single value. This means that a subquery that returns a single value can also be listed as an object in a FROM clause listing. This is termed an inline view because when a subquery is used as part of a FROM clause, it is treated like a virtual table or view. Subquery can be placed either in FROM clause, WHERE clause or HAVING clause of the main query.

Oracle allows a maximum nesting of 255 subquery levels in a WHERE clause. There is no limit for nesting subqueries expressed in a FROM clause.In practice, the limit of 255 levels is not really a limit at all because it is rare to encounter subqueries nested beyond three or four levels.

A subquery SELECT statement is very similar to the SELECT statement used to begin a regular or outer query.The complete syntax of a subquery is:

```
( SELECT [DISTINCT] subquery_select_parameter
  FROM {table_name | view_name}
          {table_name | view_name} ...
  [WHERE search_conditions]
  [GROUP BY column_name [,column_name ] ...]
  [HAVING search_conditions] )
```

WHERE clause

Most often, the subquery will be found in the WHERE clause. These subqueries are also called nested subqueries.

For example:

```
SELECT *
FROM all_tables tabs
WHERE tabs.table_name IN (SELECT cols.table_name
                  FROM all_tab_columns cols
                  WHERE cols.column_name = 'SUPPLIER_ID');
```

Limitation: Oracle allows up to 255 levels of subqueries in the WHERE clause.

FROM clause

A subquery can also be found in the FROM clause. These are called **inline views**.

For example:

```
SELECT suppliers.name, subquery1.total_amt
FROM suppliers,
 (SELECT supplier_id, SUM(orders.amount) AS total_amt
  FROM orders
  GROUP BY supplier_id) subquery1
```

```
WHERE subquery1.supplier_id = suppliers.supplier_id;
```

In this example, we've created a subquery in the FROM clause as follows:

```
(SELECT supplier_id, SUM(orders.amount) AS total_amt
 FROM orders
 GROUP BY supplier_id) subquery1
```

This subquery has been aliased with the name *subquery1*. This will be the name used to reference this subquery or any of its fields.

SELECT clause

A subquery can also be found in the SELECT clause.
For example:

```
SELECT tbls.owner, tbls.table_name,
  (SELECT COUNT(column_name) AS total_columns
   FROM all_tab_columns cols
   WHERE cols.owner = tbls.owner
   AND cols.table_name = tbls.table_name) subquery2
FROM all_tables tbls;
```

In this example, we've created a subquery in the SELECT clause as follows:

```
(SELECT COUNT(column_name) AS total_columns
 FROM all_tab_columns cols
 WHERE cols.owner = tbls.owner
 AND cols.table_name = tbls.table_name) subquery2
```

The subquery has been aliased with the name *subquery2*. This will be the name used to reference this subquery or any of its fields.

The trick to placing a subquery in the select clause is that the subquery must return a single value. This is why an aggregate function such as SUM function, COUNT function, MIN function, or MAX function is commonly used in the subquery.

Types of Subqueries

- **Single Row Sub Query:** Sub query which returns single row output. They mark the usage of single row comparison operators, when used in WHERE conditions.
- **Multiple row sub query:** Sub query returning multiple row output. They make use of multiple row comparison operators like IN, ANY, ALL. There can be sub queries returning multiple columns also.
- **Correlated Sub Query:** Correlated subqueries depend on data provided by the outer query. This type of subquery also includes subqueries that use the EXISTS operator to test the existence of data rows satisfying specified criteria.

Single Row Sub Query

A single-row subquery is used when the outer query's results are based on a single, unknown value. Although this query type is formally called "single-row," the name implies that the query returns multiple columns-but only one row of results.

Notes

However, a single-row subquery can return only one row of results consisting of only one column to the outer query.

In the below SELECT query, inner SQL returns only one row i.e. the minimum salary for the company. It in turn uses this value to compare salary of all the employees and displays only those, whose salary is equal to minimum salary.

```
SELECT first_name, salary, department_id
FROM employees
WHERE salary = (SELECT MIN (salary)
                FROM employees);
```

A HAVING clause is used when the group results of a query need to be restricted based on some condition. If a subquery's result must be compared with a group function, you must nest the inner query in the outer query's HAVING clause.

```
SELECT department_id, MIN (salary)
FROM employees
GROUP BY department_id
HAVING MIN (salary)  < (SELECT AVG (salary)
                        FROM employees)
```

Multiple Row Sub Query

Multiple-row subqueries are nested queries that can return more than one row of results to the parent query. Multiple-row subqueries are used most commonly in WHERE and HAVING clauses. Since it returns multiple rows,it must be handled by set comparison operators (IN, ALL, ANY).While IN operator holds the same meaning as discussed in earlier chapter, ANY operator compares a specified value to each value returned by the sub query while ALL compares a value to every value returned by a sub query.

Below query shows the error when single row sub query returns multiple rows.

```
SELECT          first_name, department_id
FROM employees
WHERE department_id = (SELECT department_id
                       FROM employees
                       WHERE LOCATION_ID = 100)
department_id = (select
          *
ERROR at line 4:
ORA-01427: single-row subquery returns more than one row
```

Usage of Multiple Row operators

- [> ALL] More than the highest value returned by the subquery
- [< ALL] Less than the lowest value returned by the subquery
- [< ANY] Less than the highest value returned by the subquery
- [> ANY] More than the lowest value returned by the subquery
- [= ANY] Equal to any value returned by the subquery (same as IN)

Above SQL can be rewritten using IN operator like below.

```
SELECT          first_name, department_id
FROM employees
WHERE department_id IN (SELECT department_id
                        FROM departments
```

```
                WHERE LOCATION_ID = 100)
```

Note in the above query, IN matches department ids returned from the sub query, compares it with that in the main query and returns employee's name who satisfy the condition.

A join would be better solution for above query, but for purpose of illustration, sub query has been used in it.

Correlated Sub Query

As opposed to a regular subquery, where the outer query depends on values provided by the inner query,a correlated subquery is one where the inner query depends on values provided by the outer query. This means that in a correlated subquery,the inner query is executed repeatedly, once for each row that might be selected by the outer query.

Correlated subqueries can produce result tables that answer complex management questions.

Consider the below SELECT query. Unlike the subqueries previously considered, the subquery in this SELECT statement cannot be resolved independently of the main query. Notice that the outer query specifies that rows are selected from the employee table with an alias name of e1. The inner query compares the employee department number column (DepartmentNumber) of the employee table with alias e2 to the same column for the alias table name e1.

```
SELECT EMPLOYEE_ID, salary, department_id
FROM   employees E
WHERE salary > (SELECT AVG(salary)
          FROM   EMP T
          WHERE E.department_id = T.department_id)
```

Multiple Column Sub Query

A multiple-column subquery returns more than one column to the outer query and can be listed in the outer query's FROM, WHERE, or HAVING clause. For example, the below query shows the employee's historical details for the ones whose current salary is in range of 1000 and 2000 and working in department 10 or 20.

```
SELECT first_name, job_id, salary
FROM emp_history
WHERE (salary, department_id) in (SELECT salary, department_id
                                  FROM employees
                                  WHERE salary BETWEEN 1000 and 2000
                                  AND department_id BETWEEN 10 and 20)
ORDER BY first_name;
```

When a multiple-column subquery is used in the outer query's FROM clause, it creates a temporary table that can be referenced by other clauses of the outer query. This temporary table is more formally called an inline view. The subquery's results are treated like any other table in the FROM clause. If the temporary table contains grouped data, the grouped subsets are treated as separate rows of data in a table. Consider the FROM clause in the below query. The inline view formed by the subquery is the data source for the main query.

```
SELECT *
FROM (SELECT salary, department_id
        FROM employees
        WHERE salary BETWEEN 1000 and 2000);
```

Notes

10. Aggregate Functions

Aggregate functions return a single value based on groups of rows, rather than single value for each row. You can use Aggregate functions in select lists and in ORDER BY and HAVING clauses. They are commonly used with the GROUP BY clause in a SELECT statement, where Oracle divides the rows of a queried table or view into groups.

The important Aggregate functions are :

- Avg
- Sum
- Max
- Min
- Count
- Stddev
- Variance

AVG

AVG(ALL /DISTINCT expr)

Returns the average value of expr.

Example

The following query returns the average salary of all employees.

select avg(sal) "Average Salary" from emp;

```
Average Salary
--------------------
2400.40
```

SUM

SUM(ALL/DISTINCT expr)

Returns the sum value of expr.

Example

The following query returns the sum salary of all employees.

select sum(sal) “Total Salary” from emp;

Total Salary

26500

MAX

MAX(ALL/DISTINCT expr)

Returns maximum value of expr.

Example

The following query returns the max salary from the employees.

select max(sal) “Max Salary” from emp;

Maximum Salary

4500

MIN

MIN(ALL/DISTINCT expr)

Returns minimum value of expr.

Example

The following query returns the minimum salary from the employees.

select min(sal) “Min Salary” from emp;

Minimum Salary

1200

COUNT

COUNT(*) OR COUNT(ALL/DISTINCT expr)

Returns the number of rows in the query. If you specify expr then count ignore nulls. If you specify the asterisk (*), this function returns all rows, including duplicates and nulls. COUNT never returns null.

Example

The following query returns the number of employees.

Select count(*) from emp;

COUNT

14

The following query counts the number of employees whose salary is not null.

Select count(sal) from emp;

COUNT

12

STDDEV

STDDEV(ALL/DISTINCT expr)

STDDEV returns sample standard deviation of expr, a set of numbers.

Example

The following query returns the standard deviation of salaries.

select stddev(sal) from emp;

Stddev

1430

VARIANCE

VARIANCE(ALL/DISTINCT expr)

Variance returns the variance of expr.

Example

The following query returns the variance of salaries.

select variance(sal) from emp;

Variance

1430

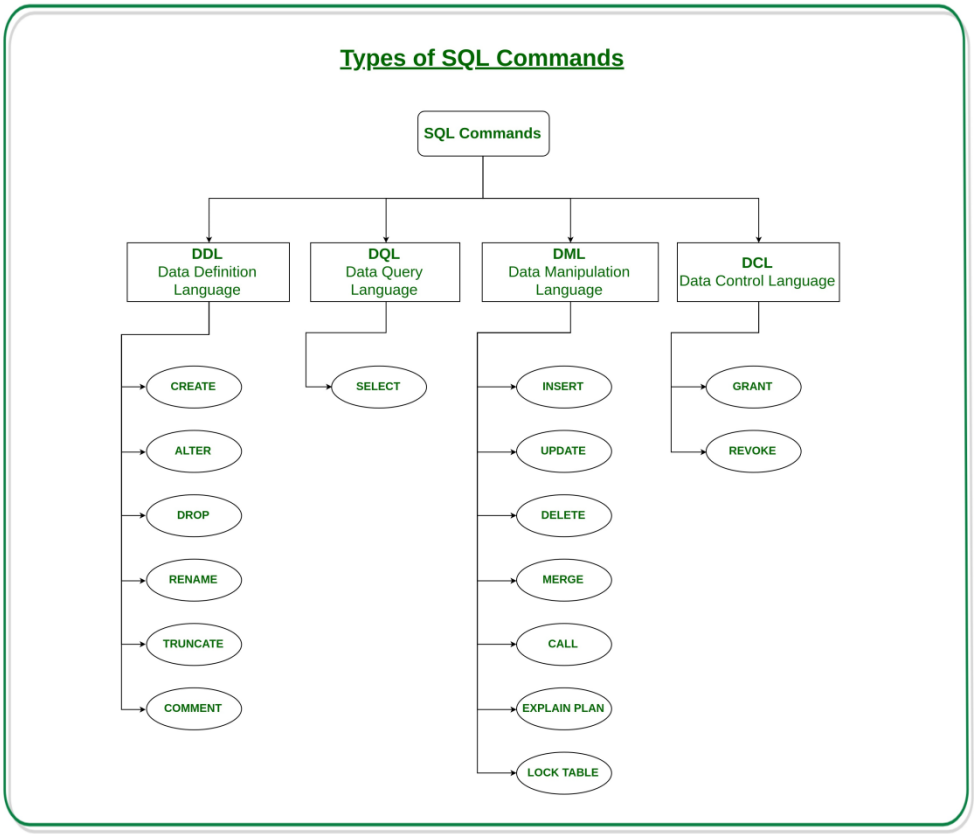
SQL | DDL, DQL, DML, DCL and TCL Commands

Structured Query Language(SQL) as we all know is the database language by the use of which we can perform certain operations on the existing database and also we can use this language to create a database. SQL uses certain commands like Create, Drop, Insert etc. to carry out the required tasks.

These SQL commands are mainly categorized into four categories as:

1. DDL – Data Definition Language
2. DQl – Data Query Language
3. DML – Data Manipulation Language
4. DCL – Data Control Language

Though many resources claim there to be another category of SQL clauses **TCL – Transaction Control Language**. So we will see in detail about TCL as well.



1. **DDL(Data Definition Language) :** DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.

Examples of DDL commands:

- **CREATE** – is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
- **DROP** – is used to delete objects from the database.
- **ALTER**-is used to alter the structure of the database.
- **TRUNCATE**–is used to remove all records from a table, including all spaces allocated for the records are removed.
- **COMMENT** –is used to add comments to the data dictionary.
- **RENAME** –is used to rename an object existing in the database.

2. **DQL (Data Query Language) :**

DML statements are used for performing queries on the data within schema objects. The purpose of DQL Command is to get some schema relation based on the query passed to it.

Notes

Example of DQL:

- **SELECT** – is used to retrieve data from the a database.

3. **DML(Data Manipulation Language) :** The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements.

Examples of DML:

- **INSERT** – is used to insert data into a table.
- **UPDATE** – is used to update existing data within a table.
- **DELETE** – is used to delete records from a database table.

4. **DCL(Data Control Language) :** DCL includes commands such as GRANT and REVOKE which mainly deals with the rights, permissions and other controls of the database system.

Examples of DCL commands:

- **GRANT**-gives user's access privileges to database.
- **REVOKE-**withdraw user's access privileges given by using the GRANT command.

5. **TCL(transaction Control Language) :** TCL commands deals with the transaction within the database.

Examples of TCL commands:

- **COMMIT**– commits a Transaction.
- **ROLLBACK**– rollbacks a transaction in case of any error occurs.
- **SAVEPOINT**–sets a savepoint within a transaction.
- **SET TRANSACTION**–specify characteristics for the transaction.

11. Data Definition Language (DDL) Statements

Data definition language (DDL) statements enable you to perform these tasks:

- Create, alter, and drop schema objects
- Grant and revoke privileges and roles
- Analyze information on a table, index, or cluster
- Establish auditing options
- Add comments to the data dictionary

The CREATE, ALTER, and DROP commands require exclusive access to the specified object. For example, an ALTER TABLE statement fails if another user has an open transaction on the specified table.

The GRANT, REVOKE, ANALYZE, AUDIT, and COMMENT commands do not require exclusive access to the specified object. For example, you can analyze a table while other users are updating the table.

Oracle implicitly commits the current transaction before and after every DDL statement.

Many DDL statements may cause Oracle to recompile or reauthorize schema objects.

DDL Statements are

CREATE :Use to create objects like CREATE TABLE, CREATE FUNCTION, CREATE SYNONYM, CREATE VIEW. Etc.

ALTER :Use to Alter Objects like ALTER TABLE, ALTER USER, ALTER TABLESPACE, ALTER DATABASE. Etc.

DROP :Use to Drop Objects like DROP TABLE, DROP USER, DROP TABLESPACE, DROP FUNCTION. Etc.

REPLACE :Use to Rename table names.

TRUNCATE :Use to truncate (delete all rows) a table.

Create

To create tables, views, synonyms, sequences, functions, procedures, packages etc.

Example

To create a table, you can give the following statement

```
create table emp (empno number(5) primary key,
        name varchar2(20),
        sal number(10,2),
        job varchar2(20),
        mgr  number(5),
        Hiredate  date,
        comm number(10,2));
```

Now Suppose you have emp table now you want to create a TAX table with the following structure and also insert rows of those employees whose salary is above 5000.

Tax	
Empno	Number(5)
Tax	Number(10,2)

To do this we can first create TAX table by defining column names and datatypes and then use INSERT into EMP SELECT …. statement to insert rows from emp table. like given below.

```
create table tax (empno number(5), tax number(10,2));

insert into tax select empno,(sal-5000)*0.40
        from emp where sal > 5000;
```

Instead of executing the above two statements the same result can be achieved by giving a single CREATE TABLE AS statement.

```
create table tax as select empno,(sal-5000)*0.4
  as tax from emp where sal>5000
```

Notes

You can also use CREATE TABLE AS statement to create copies of tables. Like to create a copy EMP table as EMP2 you can give the following statement.

create table emp2 as select * from emp;

To copy tables without rows i.e. to just copy the structure give the following statement

create table emp2 as select * from emp where 1=2;

Temporary Tables (From Oracle Ver. 8i)

It is also possible to create a temporary table. The definition of a temporary table is visible to all sessions, but the data in a temporary table is visible only to the session that inserts the data into the table. You use the CREATE GLOBAL TEMPORARY TABLE statement to create a temporary table. The ON COMMIT keywords indicate if the data in the table is transaction-specific (the default) or session-specific:

- ON COMMIT DELETE ROWS specifies that the temporary table is transaction specific and Oracle truncates the table (delete all rows) after each commit.
- ON COMMIT PRESERVE ROWS specifies that the temporary table is session specific and Oracle truncates the table when you terminate the session.

This example creates a temporary table that is transaction specific:

```
CREATE GLOBAL TEMPORARY TABLE taxable_emp
                (empno number(5),
                ename varchar2(20),
                sal number(10,2),
                tax number(10,2))
    ON COMMIT DELETE ROWS;
```

Indexes can also be created on temporary tables. They are also temporary and the data in the index has the same session or transaction scope as the data in the underlying table.

Alter

Use the ALTER TABLE statement to alter the structure of a table.

Examples:

To add new columns addr, city, pin, ph, fax to employee table you can give the following statement

```
alter table emp add (addr varchar2(20), city varchar2(20),
    pin varchar2(10),ph varchar2(20));
```

To modify the datatype and width of a column. For example we you want to increase the length of the column ename from varchar2(20) to varchar2(30) then give the following command.

alter table emp modify (ename varchar2(30))

To decrease the width of a column the column can be decreased up to largest value it holds.

alter table emp modify (ename varchar2(15));

The above is possible only if you are using Oracle ver 8i and above. In Oracle 8.0 and 7.3 you cannot decrease the column width directly unless the column is empty.

To change the datatype the column must be empty in All Oracle Versions.

To drop columns.

From Oracle Ver. 8i you can drop columns directly it was not possible in previous versions.

For example to drop PIN, CITY columns from emp table.

alter table emp drop column (pin, city);

Remember you cannot drop the column if the table is having only one column.

If the column you want to drop is having primary key constraint on it then you have to give cascade constraint clause.

alter table emp2 drop column (empno) cascade constraints;

To drop columns in previous versions of Oracle8.0 and 7.3. and to change the column name in all Oracle versions do the following.

For example we want to drop pin and city columns and to change SAL column name to SALARY.

Step 1: Create a temporary table with desired columns using subquery.

create table temp as select empno, ename, sal AS salary, addr, ph from emp;

Step 2: Drop the original table.

drop table emp;

Step 3: Rename the temporary table to the original table.

rename temp to emp;

Rename

Use the RENAME statement to rename a table, view, sequence, or private synonym for a table, view, or sequence.

Notes

- Oracle automatically transfers integrity constraints, indexes, and grants on the old object to the new object.
- Oracle invalidates all objects that depend on the renamed object, such as views, synonyms, and stored procedures and functions that refer to a renamed table.

Example

To rename table emp2 to employee2 you can give the following command.

rename emp2 to employee2

Drop

Use the drop statement to drop tables, functions, procedures, packages, views, synonym, sequences, tablespaces etc.

Example

The following command drops table emp2

drop table emp2;

If emp2 table is having primary key constraint, to which other tables refer to, then you have to first drop referential integrity constraint and then drop the table. Or if you want to drop table by dropping the referential constraints then give the following command

drop table emp2 cascade constraints;

Truncate

Use the Truncate statement to delete all the rows from table permanently . It is same as “DELETE FROM <table_name>” except

- Truncate does not generate any rollback data hence, it cannot be roll backed.
- If any delete triggers are defined on the table. Then the triggers are not fired
- It deallocates free extents from the table. So that the free space can be use by other tables.

Example

truncate table emp;

If you do not want free space and keep it with the table. Then specify the REUSE storage clause like this

truncate table emp reuse storage;

12. Data Manipulation Language (DML) Statements

Data manipulation language (DML) statements query and manipulate data in existing schema objects. These statements do not implicitly commit the current transaction.

The following are the DML statements available in Oracle.

- INSERT :Use to Add Rows to existing table.
- UPDATE :Use to Edit Existing Rows in tables.
- DELETE :Use to Delete Rows from tables.

Insert

Use the Insert Statement to Add records to existing Tables.

Examples.

To add a new row to an emp table.

```
Insert into emp values (101,'Sami','G.Manager',
        '8-aug-1998',2000);
```

If you want to add a new row by supplying values for some columns not all the columns then you have to mention the name of the columns in insert statements. For example the following statement inserts row in emp table by supplying values for empno, ename, and sal columns only. The Job and Hiredate columns will be null.

```
Insert into emp (empno,ename,sal) values (102,'Ashi',5000);
```

Suppose you want to add rows from one table to another i.e. suppose we have Old_Emp table and emp table with the following structure

Old_emp	
Column Name	Datatype & Width
Empno	Number(5)
Ename	Varchar2(20)
Sal	Number(10,2)
Tdate	Date

Emp	
Column Name	Datatype & Width
Empno	Number(5)
Ename	Varchar2(20)
Sal	Number(10,2)
Hiredate	Date
Job	Varchar2(20)

Now we want to add rows from old_emp table to emp table. Then you can give the following insert statement

```
Insert into emp (empno, ename, sal)
    select empno, ename, sal from old_emp;
```

Update

Update statement is used to update rows in existing tables which is in your own schema or if you have update privilege on them.

For example to raise the salary by Rs.500 of employee number 104. You can give the following statement.

update emp set sal=sal+500 where empno = 104;

In the above statement if we did not give the where condition then all employees salary will be raised by Rs. 500. That's why always specify proper WHERE condition if don't want to update all employees.

For example We want to change the name of employee no 102 from 'Sami' to 'Mohd Sami' and to raise the salary by 10%. Then the statement will be.

update emp set name='Mohd Sami',
sal=sal+(sal*10/100) where empno=102;

Now we want to raise the salary of all employees by 5%.

update emp set sal=sal+(sal*5/100);

Now to change the names of all employees to uppercase.

update emp set name=upper(name);

Suppose We have a student table with the following structure.

Rollno	Name	Maths	Phy	Chem	Total	Average
101	Sami	99	90	89		
102	Scott	34	77	56		
103	Smith	45	82	43		

Now to compute total which is sum of Maths,Phy and Chem and average.

update student set total=maths+phy+chem,
average=(maths+phy+chem)/3;

Using Sub Query in the Update Set Clause.

Suppose we added the city column in the employee table and now we want to set this column with corresponding city column in department table which is join to employee table on deptno.

update emp set city=(select city from dept
where deptno= emp.deptno);

Delete

Use the DELETE statement to delete the rows from existing tables which are in your schema or if you have DELETE privilege on them.

For example to delete the employee whose empno is 102.

delete from emp where empno=102;

If you don't mention the WHERE condition then all rows will be deleted.

Suppose we want to delete all employees whose salary is above 2000. Then give the following DELETE statement.

delete from emp where salary > 2000;

The following statement has the same effect as the preceding example, but uses a subquery:

```
DELETE FROM (SELECT * FROM emp)
   WHERE sal > 2000;
```

To delete all rows from emp table.

delete from emp;

13. Data Control Language (DCL) using GRANT and REVOKE

Data Control Language (DCL) is used to control privileges in Database. To perform any operation in the database, such as for creating tables, sequences or views, a user needs privileges. Privileges are of two types,

- **System:** This includes permissions for creating session, table, etc and all types of other system privileges.
- **Object:** This includes permissions for any command or query to perform any operation on the database tables.

In DCL we have two commands,

- GRANT: Used to provide any user access privileges or other priviliges for the database.
- REVOKE: Used to take back permissions from any user.

Allow a User to create session

When we create a user in SQL, it is not even allowed to login and create a session until and unless proper permissions/priviliges are granted to the user.
Following command can be used to grant the session creating priviliges.

```
GRANT CREATE SESSION TO username;
```

Allow a User to create table

To allow a user to create tables in the database, we can use the below command,

```
GRANT CREATE TABLE TO username;
```

Notes

Grant all privilege to a User

sysdba is a set of priviliges which has all the permissions in it. So if we want to provide all the privileges to any user, we can simply grant them the sysdba permission.

GRANT sysdba TO username

Grant permission to create any table

Sometimes user is restricted from creating come tables with names which are reserved for system tables. But we can grant privileges to a user to create any table using the below command,

GRANT CREATE ANY TABLE TO username

Grant permission to drop any table

As the title suggests, if you want to allow user to drop any table from the database, then grant this privilege to the user,

GRANT DROP ANY TABLE TO username

To take back Permissions

And, if you want to take back the privileges from any user, use the REVOKE command.

REVOKE CREATE TABLE FROM username

BLOCK 3: INDEX AND VIEW

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns. By default, Oracle creates B-tree indexes.

Create an Index

Syntax

The syntax for creating an index in Oracle is:

```
CREATE [UNIQUE] INDEX index_name
 ON table_name (column1, column2, ... column_n)
 [COMPUTE STATISTICS];
```

Unique

It indicates that the combination of values in the indexed columns must be unique.

index_name

The name to assign to the index.

table_name

The name of the table in which to create the index.

column1, column2, ... column_n

The columns to use in the index.

Compute Statistics

It tells Oracle to collect statistics during the creation of the index. The statistics are then used by the optimizer to choose a "plan of execution" when SQL statements are executed.

Example

Let's look at an example of how to create an index in Oracle/PLSQL.

For example:

```
CREATE INDEX supplier_idx ON supplier (supplier_name);
```

In this example, we've created an index on the supplier table called supplier_idx. It consists of only one field - the supplier_name field.

We could also create an index with more than one field as in the example below:

```
CREATE INDEX supplier_idx ON supplier (supplier_name, city);
```

Notes

We could also choose to collect statistics upon creation of the index as follows:

```
CREATE INDEX supplier_idx
  ON supplier (supplier_name, city)
  COMPUTE STATISTICS;
```

Create a Function-Based Index

In Oracle, you are not restricted to creating indexes on only columns. You can create function-based indexes.

Syntax

The syntax for creating a function-based index in Oracle/PLSQL is:

```
CREATE [UNIQUE] INDEX index_name
  ON table_name (function1, function2, ... function_n)
  [ COMPUTE STATISTICS ];
```

Unique
It indicates that the combination of values in the indexed columns must be unique.

index_name
The name to assign to the index.

table_name
The name of the table in which to create the index.

function1, function2, ... function_n
The functions to use in the index.

Compute Statistics
It tells Oracle to collect statistics during the creation of the index. The statistics are then used by the optimizer to choose a "plan of execution" when SQL statements are executed.

Example

Let's look at an example of how to create a function-based index in Oracle.

For example:

```
CREATE INDEX supplier_idx
  ON supplier (UPPER(supplier_name));
```

In this example, we've created an index based on the uppercase evaluation of the *supplier_name* field.

However, to be sure that the Oracle optimizer uses this index when executing your SQL statements, be sure that UPPER(supplier_name) does not evaluate to a NULL value. To ensure this, add **UPPER(supplier_name) IS NOT NULL** to your WHERE clause as follows:

```
SELECT supplier_id, supplier_name, UPPER(supplier_name)
FROM supplier
WHERE UPPER(supplier_name) IS NOT NULL
ORDER BY UPPER(supplier_name);
```

Rename an Index

Syntax

The syntax for renaming an index in Oracle is:

```
ALTER INDEX index_name
  RENAME TO new_index_name;
```

index_name
The name of the index that you wish to rename.

new_index_name
The new name to assign to the index.

Example

Let's look at an example of how to rename an index in Oracle/PLSQL.

For example:

```
ALTER INDEX supplier_idx
  RENAME TO supplier_index_name;
```

In this example, we're renaming the index called *supplier_idx* to *supplier_index_name*.

Collect Statistics on an Index

If you forgot to collect statistics on the index when you first created it or you want to update the statistics, you can always use the ALTER INDEX command to collect statistics at a later date.
Syntax

The syntax for collecting statistics on an index in Oracle/PLSQL is:

```
ALTER INDEX index_name
  REBUILD COMPUTE STATISTICS;
```

index_name
The index in which to collect statistics.

Example

Let's look at an example of how to collect statistics for an index in Oracle/PLSQL.

For example:

```
ALTER INDEX supplier_idx
  REBUILD COMPUTE STATISTICS;
```

Notes

In this example, we're collecting statistics for the index called supplier_idx.

Drop an Index

Syntax

The syntax for dropping an index in Oracle/PLSQL is:

```
DROP INDEX index_name;
```

index_name
The name of the index to drop.

Example

Let's look at an example of how to drop an index in Oracle/PLSQL.

For example:

```
DROP INDEX supplier_idx;
```

In this example, we're dropping an index called supplier_idx.

15. Views

An Oracle VIEW, in essence, is a virtual table that does not physically exist. Rather, it is created by a query joining one or more tables.

Create VIEW

Syntax

The syntax for the CREATE VIEW Statement in Oracle/PLSQL is:

```
CREATE VIEW view_name AS
  SELECT columns
  FROM tables
  [WHERE conditions];
```

view_name

The name of the Oracle VIEW that you wish to create.

WHERE conditions

Optional. The conditions that must be met for the records to be included in the VIEW.

Example

Here is an example of how to use the Oracle CREATE VIEW:

```
CREATE VIEW sup_orders AS
  SELECT suppliers.supplier_id, orders.quantity, orders.price
  FROM suppliers
```

```
INNER JOIN orders
ON suppliers.supplier_id = orders.supplier_id
WHERE suppliers.supplier_name = 'Microsoft';
```

This Oracle CREATE VIEW example would create a virtual table based on the result set of the SELECT statement. You can now query the Oracle VIEW as follows:


```
SELECT *
FROM sup_orders;
```

Update VIEW

You can modify the definition of an Oracle VIEW without dropping it by using the Oracle CREATE OR REPLACE VIEW Statement.

Syntax

The syntax for the CREATE OR REPLACE VIEW Statement in Oracle/PLSQL is:

```
CREATE OR REPLACE VIEW view_name AS
  SELECT columns
  FROM table
  WHERE conditions;
```

view_name

The name of the Oracle VIEW that you wish to create or replace.

Example

Here is an example of how you would use the Oracle CREATE OR REPLACE VIEW Statement:

```
CREATE or REPLACE VIEW sup_orders AS
  SELECT suppliers.supplier_id, orders.quantity, orders.price
  FROM suppliers
  INNER JOIN orders
  ON suppliers.supplier_id = orders.supplier_id
  WHERE suppliers.supplier_name = 'Apple';
```

This Oracle CREATE OR REPLACE VIEW example would update the definition of the Oracle VIEW called *sup_orders* without dropping it. If the Oracle VIEW did not yet exist, the VIEW would merely be created for the first time.

Drop VIEW

Once an Oracle VIEW has been created, you can drop it with the Oracle DROP VIEW Statement.

Syntax

The syntax for the DROP VIEW Statement in Oracle/PLSQL is:

Notes

```
DROP VIEW view_name;
```

view_name

The name of the view that you wish to drop.

Example

Here is an example of how to use the Oracle DROP VIEW Statement:

```
DROP VIEW sup_orders;
```

This Oracle DROP VIEW example would drop/delete the Oracle VIEW called *sup_orders*.

16. Partitioning the Tables

- Partitioning enhances the performance, manageability, and availability of a wide variety of applications and helps reduce the total cost of ownership for storing large amounts of data.
- Partitioning allows tables, indexes, and index-organized tables to be subdivided into smaller pieces, enabling these database objects to be managed and accessed at a finer level of granularity.
- Oracle provides a rich variety of partitioning strategies and extensions to address every business requirement. Moreover, since it is entirely transparent, partitioning can be applied to almost any application without the need for potentially expensive and time consuming application changes.

Range Partitioning Tables

Range partitioning is useful when you have distinct ranges of data you want to store together. The classic example of this is the use of dates. Partitioning a table using date ranges allows all data of a similar age to be stored in same partition. Once historical data is no longer needed the whole partition can be removed. If the table is indexed correctly search criteria can limit the search to the partitions that hold data of a correct age.

```
CREATE TABLE invoices
(invoice_no    NUMBER NOT NULL,
 invoice_date  DATE   NOT NULL,
 comments      VARCHAR2(500))
PARTITION BY RANGE (invoice_date)
(PARTITION invoices_q1 VALUES LESS THAN
(TO_DATE('01/04/2019', 'DD/MM/YYYY')) TABLESPACE users,
 PARTITION invoices_q2 VALUES LESS THAN
(TO_DATE('01/07/2019', 'DD/MM/YYYY')) TABLESPACE users,
 PARTITION invoices_q3 VALUES LESS THAN
(TO_DATE('01/09/2019', 'DD/MM/YYYY')) TABLESPACE users,
 PARTITION invoices_q4 VALUES LESS THAN
(TO_DATE('01/01/2020', 'DD/MM/YYYY')) TABLESPACE users);
```

Hash Partitioning Tables

Hash partitioning is useful when there is no obvious range key, or range partitioning will cause uneven distribution of data. The number of partitions must be a power of 2 (2, 4, 8, 16...) and can be specified by the PARTITIONS...STORE IN clause.

```
CREATE TABLE invoices
(invoice_no    NUMBER NOT NULL,
 invoice_date  DATE   NOT NULL,
 comments      VARCHAR2(500))
PARTITION BY HASH (invoice_no)
PARTITIONS 4
STORE IN (users, users, users, users);
```

Composite Partitioning Tables

Composite partitioning allows range partitions to be hash subpartitioned on a different key. The greater number of partitions increases the possiblities for parallelism and reduces the chances of contention. The following example will range partition the table on invoice_date and subpartitioned these on the invoice_no giving a totol of 32 subpartitions.

```
CREATE TABLE invoices
(invoice_no    NUMBER NOT NULL,
 invoice_date  DATE   NOT NULL,
 comments      VARCHAR2(500))
PARTITION BY RANGE (invoice_date)
SUBPARTITION BY HASH (invoice_no)
SUBPARTITIONS 8
(PARTITION invoices_q1 VALUES LESS THAN
(TO_DATE('01/04/2019', 'DD/MM/YYYY')),
 PARTITION invoices_q2 VALUES LESS THAN
(TO_DATE('01/07/2019', 'DD/MM/YYYY')),
 PARTITION invoices_q3 VALUES LESS THAN
(TO_DATE('01/09/2019', 'DD/MM/YYYY')),
 PARTITION invoices_q4 VALUES LESS THAN
(TO_DATE('01/01/2020', 'DD/MM/YYYY'));
```

17. Locking Tables

The **LOCK TABLE statement** is used to lock tables, table partitions, or table subpartitions.

Syntax

The syntax for the **LOCK TABLE statement** is:

LOCK TABLE tables IN lock_mode MODE [WAIT [, integer] | NOWAIT];
Parameters or Arguments

tables
A comma-delimited list of tables.

lock_mode

It is one of the following values:

lock_mode	Explanation
ROW SHARE	Allows concurrent access to the table, but users are prevented from locking the entire table for exclusive access.
ROW EXCLUSIVE	Allows concurrent access to the table, but users are prevented from locking the entire table with exclusive access and locking the table in share mode.
SHARE UPDATE	Allows concurrent access to the table, but users are prevented from locking the entire table for exclusive access.
SHARE	Allows concurrent queries but users are prevented from updating the locked table.
SHARE ROW EXCLUSIVE	Users can view records in table, but are prevented from updating the table or from locking the table in SHARE mode.
EXCLUSIVE	Allows queries on the locked table, but no other activities.

WAIT
It specifies that the database will wait (up to a certain number of seconds as specified by *integer*) to acquire a DML lock.

NOWAIT
It specifies that the database should not wait for a lock to be released.
Example

Let's look at an example of how to use the LOCK TABLE statement in Oracle.
For example:

```
LOCK TABLE suppliers IN SHARE MODE NOWAIT;
```

This example would lock the suppliers table in *SHARE MODE* and not wait for a lock to be released.

BLOCK 4 : EXCEPTION HANDLING AND PL/SQL

An exception is an error which disrupts the normal flow of program instructions. PL/SQL provides us the exception block which raises the exception thus helping the programmer to find out the fault and resolve it.

There are two types of exceptions defined in PL/SQL

1. User defined exception.
2. System defined exceptions.

Syntax to write an exception

```
WHEN exception THEN
   statement;

DECLARE
declarations section;

BEGIN
        executable command(s);

        EXCEPTION
        WHEN exception1 THEN
        statement1;
        WHEN exception2 THEN
        statement2;
        [WHEN others THEN]
        /* default exception handling code */

END;
```

Note:

When other keyword should be used only at the end of the exception handling block as no exception handling part present later will get executed as the control will exit from the block after executing the WHEN OTHERS.

18.1 System defined exceptions:

These exceptions are predefined in PL/SQL which get raised WHEN certain database rule is violated.

System-defined exceptions are further divided into two categories:

- Named system exceptions.
- Unnamed system exceptions.

Named system exceptions: They have a predefined name by the system like ACCESS_INTO_NULL, DUP_VAL_ON_INDEX, LOGIN_DENIED etc.

Oracle has a standard set of exceptions already named as follows:

Oracle Exception Name	Oracle Error	Explanation
DUP_VAL_ON_INDEX	ORA-00001	You tried to execute an INSERT or UPDATE statement that has created a duplicate value in a field restricted by a unique index.
TIMEOUT_ON_RESOURCE	ORA-00051	You were waiting for a resource and you timed out.
TRANSACTION_BACKED_OUT	ORA-00061	The remote portion of a transaction has rolled back.
INVALID_CURSOR	ORA-01001	You tried to reference a cursor that does not yet exist. This may have happened because you've executed a FETCH cursor or CLOSE cursor before OPENing the cursor.
NOT_LOGGED_ON	ORA-01012	You tried to execute a call to Oracle before logging in.
LOGIN_DENIED	ORA-01017	You tried to log into Oracle with an invalid username/password combination.
NO_DATA_FOUND	ORA-01403	You tried one of the following: You executed a SELECT INTO statement and no rows were returned. You referenced an uninitialized row in a table. You read past the end of file with the UTL_FILE package.
TOO_MANY_ROWS	ORA-01422	You tried to execute a SELECT INTO statement and more than one row was returned.
ZERO_DIVIDE	ORA-01476	You tried to divide a number by zero.
INVALID_NUMBER	ORA-01722	You tried to execute a SQL statement that tried to convert a string to a number, but it was unsuccessful.
STORAGE_ERROR	ORA-06500	You ran out of memory or memory was corrupted.
PROGRAM_ERROR	ORA-06501	This is a generic "Contact Oracle support" message because an internal problem was encountered.
VALUE_ERROR	ORA-06502	You tried to perform an operation and there was a error on a conversion, truncation, or invalid constraining of numeric or character data.
CURSOR_ALREADY_OPEN	ORA-06511	You tried to open a cursor that is already open.

So we will discuss some of the most commonly used exceptions:

Lets create a table marks.

```
create table marks(g_id int , g_name varchar(20), marks int);
```

```
insert into marks values(1, 'Suraj',100);
insert into marks values(2, 'Praveen',97);
insert into marks values(3, 'Jessie', 99);
```

i. NO_DATA_FOUND:

It is raised WHEN a SELECT INTO statement returns *no* rows.

For eg:

```
DECLARE
  temp varchar(20);

BEGIN
  SELECT g_id into temp from geeks where g_name='suresh';

exception
  WHEN no_data_found THEN
    dbms_output.put_line('ERROR');
    dbms_output.put_line('there is no name as');
    dbms_output.put_line(' suresh in marks table');
end;
```

Output:

```
ERROR
there is no name as suresh in marks table
```

ii. TOO_MANY_ROWS:

It is raised WHEN a SELECT INTO statement returns *more* than one row.

```
DECLARE
  temp varchar(20);

BEGIN

    -- raises an exception as SELECT
    -- into trying to return too many rows
      SELECT g_name into temp from geeks;
      dbms_output.put_line(temp);

EXCEPTION
  WHEN too_many_rows THEN
    dbms_output.put_line('error trying to SELECT too many rows');

end;
```

Output:

```
error trying to SELECT too many rows
```

iii. VALUE_ERROR:

This error is raised WHEN a statement is executed that resulted in an arithmetic, numeric, string, conversion, or constraint error. This error mainly results from programmer error or invalid data input.

Notes

```
DECLARE
  temp number;

BEGIN
  SELECT g_name  into temp from geeks where g_name='Suraj';
  dbms_output.put_line('the g_name is '||temp);

EXCEPTION
  WHEN value_error THEN
  dbms_output.put_line('Error');
  dbms_output.put_line('Change data type of temp to varchar(20)');

END;
```

Output:

Error
Change data type of temp to varchar(20)

iv. ZERO_DIVIDE

raises exception WHEN dividing with zero.

```
DECLARE
  a int:=10;
  b int:=0;
  answer int;

BEGIN
  answer:=a/b;
  dbms_output.put_line('the result after division is'||answer);

exception
  WHEN zero_divide THEN
    dbms_output.put_line('dividing by zero please check the values again');
    dbms_output.put_line('the value of a is '||a);
    dbms_output.put_line('the value of b is '||b);
END;
```

Output:

dividing by zero please check the values again
the value of a is 10
the value of b is 0

18.2 User defined exceptions:

This type of users can create their own exceptions according to the need and to raise these exceptions explicitly ***raise*** command is used.

Example:

- Divide non-negative integer x by y such that the result is greater than or equal to 1.

From the given question we can conclude that there exist two exceptions

- Division be zero.
- If result is greater than or equal to 1 means y is less than or equal to x.

```
DECLARE
  x int:=&x; /*taking value at run time*/
  y int:=&y;
  div_r float;
  exp1 EXCEPTION;
  exp2 EXCEPTION;

BEGIN
  IF y=0 then
     raise exp1;

  ELSEIF y > x then
    raise exp2;

  ELSE
    div_r:= x / y;
    dbms_output.put_line('the result is '||div_r);

  END IF;

EXCEPTION
  WHEN exp1 THEN
    dbms_output.put_line('Error');
    dbms_output.put_line('division by zero not allowed');

  WHEN exp2 THEN
    dbms_output.put_line('Error');
    dbms_output.put_line('y is greater than x please check the input');

END;
```

Input 1: x = 20
y = 10

Output: the result is 2
Input 2: x = 20
y = 0

Output:
Error
division by zero not allowed
Input 3: x=20
y = 30

Notes

Output:<.em>
Error
y is greater than x please check the input

RAISE_APPLICATION_ERROR:

It is used to display user-defined error messages with error number whose range is in between -20000 and -20999. When RAISE_APPLICATION_ERROR executes it returns error message and error code which looks ***same as Oracle built-in error.***

Example:

```
DECLARE
   myex EXCEPTION;
   n NUMBER :=10;

BEGIN
   FOR i IN 1..n LOOP
   dbms_output.put_line(i*i);
      IF i*i=36 THEN
      RAISE myex;
      END IF;
   END LOOP;

EXCEPTION
   WHEN myex THEN
      RAISE_APPLICATION_ERROR(-20015, 'Welcome to ALU');

END;
```

Output:

Error report:
ORA-20015: Welcome to ALU
ORA-06512: at line 13

1
4
9
16
25
36

19. Cursors

In Oracle, a **cursor** is a mechanism by which you can assign a name to a SELECT statement and manipulate the information within that SQL statement.

Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time.

There are two types of cursors −

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement.

The following table provides the description of the most used attributes –

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example

We will be using the CUSTOMERS table we had created and used in the previous chapters.

Notes

Select * from customers;

ID	Name	Age	Address	Salary
1	Ramesh	32	Chennai	35000
2	Suresh	26	Trichy	20000
3	Kumar	28	Karaikudi	30000
4	Malik	31	Madurai	40000
5	Suman	27	Karur	54000
6	Chitra	30	Dindigul	45000

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE customers SET salary = salary + 500;
  IF sql%notfound THEN
    dbms_output.put_line('no customers selected');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers selected ');
  END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
6 customers selected

PL/SQL procedure successfully completed.
```

If you check the records in customers table, you will find that the rows have been updated −

Select * from customers;

ID	Name	Age	Address	Salary
1	Ramesh	32	Chennai	35500
2	Suresh	26	Trichy	20500
3	Kumar	28	Karaikudi	30500
4	Malik	31	Madurai	40500
5	Suman	27	Karur	54500
6	Chitra	30	Dindigul	45500

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**.

An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is −

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

```
CURSOR c_customers IS
  SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it.

For example, we will open the above defined cursor as follows −

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors

Notes

```
DECLARE
  c_id customers.id%type;
  c_name customerS.No.ame%type;
  c_addr customers.address%type;
  CURSOR c_customers is
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
  FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

1	Ramesh	Chennai
2	Suresh	Trichy
3	Kumar	Karaikudi
4	Malik	Madurai
5	Suman	Karur
6	Chitra	Dindigul

PL/SQL procedure successfully completed.

20. Triggers

Triggers are stored programs, which are automatically executed or fired when some events occur.

Triggers are, in fact, written to be executed in response to any of the following events −

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes −

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations

- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is −

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
  Declaration-statements
BEGIN
  Executable-statements
EXCEPTION
  Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

```
Select * from customers;
```

Notes

ID	Name	Age	Address	Salary
1	Ramesh	32	Chennai	35000
2	Suresh	26	Trichy	20000
3	Kumar	28	Karaikudi	30000
4	Malik	31	Madurai	40000
5	Suman	27	Karur	54000
6	Chitra	30	Dindigul	45000

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table.

This trigger will display the salary difference between the old values and new values −

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
  sal_diff number;
BEGIN
  sal_diff := :NEW.salary  - :OLD.salary;
  dbms_output.put_line('Old salary: ' || :OLD.salary);
  dbms_output.put_line('New salary: ' || :NEW.salary);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Trigger created.
```

The following points need to be considered here −

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
```

```
VALUES (7, 'Karthick', 22, 'Villupuram', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null.

Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

21. Packages

Packages are schema objects that groups logically related PL/SQL types, variables, and subprograms.

A package will have two mandatory parts −

- Package specification
- Package body or definition

Package Specification

The specification is the interface to the package. It just **DECLARES** the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

The following code snippet shows a package specification having a single procedure.

You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
```

```
   PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Package created.
```

Notes

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from the code outside the package.

The **CREATE PACKAGE BODY** Statement is used for creating the package body.

The following code snippet shows the package body declaration for the ***cust_sal*** package created above for the CUSTOMERS table already been created.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
   PROCEDURE find_sal(c_id customers.id%TYPE) IS
   c_sal customers.salary%TYPE;
   BEGIN
      SELECT salary INTO c_sal
      FROM customers
      WHERE id = c_id;
      dbms_output.put_line('Salary: '|| c_sal);
   END find_sal;
END cust_sal;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Package body created.
```

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax −

```
package_name.element_name;
```

Consider, we already have created the above package in our database schema, the following program uses the ***find_sal*** method of the ***cust_sal*** package −

```
DECLARE
   code customers.id%type := &cc_id;
BEGIN
   cust_sal.find_sal(code);
END;
/
```

When the above code is executed at the SQL prompt, it prompts to enter the customer ID and when you enter an ID, it displays the corresponding salary as follows −

```
Enter value for cc_id: 1
Salary: 3000

PL/SQL procedure successfully completed.
```

Example

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records −

```
Select * from customers;
```

ID	Name	Age	Address	Salary
1	Ramesh	32	Chennai	35000
2	Suresh	26	Trichy	20000
3	Kumar	28	Karaikudi	30000
4	Malik	31	Madurai	40000
5	Suman	27	Karur	54000
6	Chitra	30	Dindigul	45000

The Package Specification

```
CREATE OR REPLACE PACKAGE c_package AS
  -- Adds a customer
  PROCEDURE addCustomer(c_id   customers.id%type,
  c_name  customerS.No.ame%type,
  c_age  customers.age%type,
  c_addr customers.address%type,
  c_sal  customers.salary%type);

  -- Removes a customer
  PROCEDURE delCustomer(c_id  customers.id%TYPE);
  --Lists all customers
  PROCEDURE listCustomer;

END c_package;
/
```

When the above code is executed at the SQL prompt, it creates the above package and displays the following result −

```
Package created.
```

Creating the Package Body

```
CREATE OR REPLACE PACKAGE BODY c_package AS
  PROCEDURE addCustomer(c_id  customers.id%type,
    c_name customerS.No.ame%type,
    c_age  customers.age%type,
    c_addr  customers.address%type,
    c_sal   customers.salary%type)
  IS
```

Notes

```
BEGIN
  INSERT INTO customers (id,name,age,address,salary)
    VALUES(c_id, c_name, c_age, c_addr, c_sal);
END addCustomer;

PROCEDURE delCustomer(c_id   customers.id%type) IS
BEGIN
  DELETE FROM customers
  WHERE id = c_id;
END delCustomer;

PROCEDURE listCustomer IS
CURSOR c_customers is
  SELECT  name FROM customers;
TYPE c_list is TABLE OF customers.Name%type;
name_list c_list := c_list();
counter integer :=0;
BEGIN
  FOR n IN c_customers LOOP
  counter := counter +1;
  name_list.extend;
  name_list(counter) := n.name;
  dbms_output.put_line('Customer('||counter||')'||name_list
       (counter));
  END LOOP;
END listCustomer;

END c_package;
/
```

The above example makes use of the **nested table**. We will discuss the concept of nested table in the next chapter.

When the above code is executed at the SQL prompt, it produces the following result −

```
Package body created.
```

Using the Package

The following program uses the methods declared and defined in the package *c_package*.

```
DECLARE
  code customers.id%type:= 8;
BEGIN
  c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
  c_package.addcustomer(8, 'Subha', 32, 'Pudukkottai', 7500);
  c_package.listcustomer;
  c_package.delcustomer(code);
  c_package.listcustomer;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Customer(1): Ramesh
Customer(2): Suresh
Customer(3): Kumar
Customer(4): Malik
Customer(5): Suman
Customer(6): Chitra
Customer(7): Rajnish
Customer(8): Subha
Customer(1): Ramesh
Customer(2): Suresh
Customer(3): Kumar
Customer(4): Malik
Customer(5): Suman
Customer(6): Chitra
Customer(7): Rajnish

PL/SQL procedure successfully completed
```

22. Functions

A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows −

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
  < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Notes

Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

Select * from customers;

ID	Name	Age	Address	Salary
1	Ramesh	32	Chennai	35000
2	Suresh	26	Trichy	20000
3	Kumar	28	Karaikudi	30000
4	Malik	31	Madurai	40000
5	Suman	27	Karur	54000
6	Chitra	30	Dindigul	45000

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
  total number(2) := 0;
BEGIN
  SELECT count(*) into total
  FROM customers;

  RETURN total;
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

Function created.

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value.

Following program calls the function **totalCustomers** from an anonymous block −

```
DECLARE
  c number(2);
BEGIN
  c := totalCustomers();
```

```
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Total no. of Customers: 6

PL/SQL procedure successfully completed.
```

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
   a number;
   b number;
   c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
   IF x > y THEN
      z:= x;
   ELSE
      Z:= y;
   END IF;
   RETURN z;
END;
BEGIN
   a:= 23;
   b:= 45;
   c := findMax(a, b);
   dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Maximum of (23,45): 45

PL/SQL procedure successfully completed.
```

PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −

Notes

```
n! = n*(n-1)!
  = n*(n-1)*(n-2)!
    ...
  = n*(n-1)*(n-2)*(n-3)... 1
```

The following program calculates the factorial of a given number by calling itself recursively −

```
DECLARE
  num number;
  factorial number;

FUNCTION fact(x number)
RETURN number
IS
  f number;
BEGIN
  IF x=0 THEN
    f := 1;
  ELSE
    f := x * fact(x-1);
  END IF;
RETURN f;
END;

BEGIN
  num:= 6;
  factorial := fact(num);
  dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Factorial 6 is 720

PL/SQL procedure successfully completed.
```

BLOCK 5 : APPLICATION DEVELOPMENT

Library Information System

The following section describes the example of database designed for Library Information System. Here, the General Library considered for design and development.

The relations / tables for Library Information System are as follows:

- Book_details
- Publisher_details
- Author_details
- Member_details
- Borrow_details

The attributes of each relation / table:

Table: Publisher_details

Attribute Name	Type	Size
Publisher_id	Varchar2	5
Publisher_name	Varchar2	30
Publisher_city	Varchar2	20

Table: Author_details

Attribute Name	Type	Size
Author_id	Varchar2	5
Author_name	Varchar2	30
Author_Country	Varchar2	20

Table: Book_details

Attribute Name	Type	Size
Book_id	Varchar2	5
Book_name	Varchar2	30
Prize	Number	6,2
Edition	Number	2
Publisher_id	Varchar2	5
Author_id	Varchar2	5

Table: Member_details

Attribute Name	Type	Size
Member_id	Varchar2	5
Member_name	Varchar2	30
Member_city	Varchar2	20
Member_phone	Number	10
Last_mem_paid	Date	

Notes

Table: Borrow_details

Attribute Name	Type	Size
Book_id	Varchar2	5
Member_id	Varchar2	5
Borrow_date	Date	
Return_date	Date	

Table creation:

Publisher_details

```
Create table Publisher_details (
        Publisher_id    varchar2(5)     primary key,
        Publisher_name  varchar2(30)    not null,
        Publisher_city  varchar2(20)
        );
```

Author_details

```
Create table Author_details (
        Author_id       varchar2(5)     primary key,
        Author_name     varchar2(30)    not null,
        Author_country  varchar2(20)
        );
```

Book_details

```
Create table Book_details (
        Book_id         varchar2(5)     primary key,
        Book_name       varchar2(30)    not null,
        Prize           number(6,2),
        Edition         number(2),
        Publisher_id    varchar2(5)     references Publihser_details(Publihser_id),
        Author_id       varchar2(5)     references Author_details(Author_id)
        );
```

Member_details

```
Create table Member_details (
        Member_id       varchar2(5)     primary key,
        Member_name     varchar2(30)    not null,
        Member_city     varchar2(20),
        Member_phone    number(10)      unique,
        Mem_last_paid   date
        );
```

Borrow_details

```
Create table Borrow_details (
        Member_id   varchar2(5)     references Member_details(Member_id),
        Book_id         varchar2(5)     references Book_details(Book_id),
        Borrow_date      date,
        Return_date      date,
```

);

Insert queries:

Insert into Publisher_details values (‘P1001’, ‘PHI’, ‘New Delhi’);
Insert into Publisher_details values (‘P1245’, ‘Macmillan India’, ‘New Delhi’);
Insert into Publisher_details values (‘P2416’, ‘Emerald Publishing’, ‘Chennai’);

Insert into Author_details values (‘A1241’, ‘C J Date’, ‘USA’);
Insert into Author_details values (‘A2413’, ‘Pressman’, ‘USA’);
Insert into Author_details values (‘A1416’, ‘C Muthu’, ‘India’);

Insert into Book_details values (‘B1011’, ‘Database Systems’, 850.00, 4, ‘P1001’, ‘A1241’);
Insert into Book_details values (‘B2111’, ‘Software Engineering’, 1150.00, 6, ‘P1245’, ‘A2413’);
Insert into Book_details values (‘B1456’, ‘Java Programming’, 650.00, 2, ‘P2416’, ‘A1416’);

Insert into Member_details values (‘M1234’, ‘R. Suresh’, ‘Karaikudi’, 9876567890, ‘03/07/2019’);
Insert into Member_details values (‘M2654’, ‘K. Mohamed’, ‘Devakottai’, 6345789765, ‘12/08/2019’);
Insert into Member_details values (‘M3124’, ‘M. Sunil’, ‘Karaikudi’, 9786956432, ‘03/06/2019’);

Insert into Borrow_details values (‘M1234’, ‘B1011’, ‘13/09/2019’, ‘28/09/2019’);
Insert into Borrow_details values (‘M2654’, ‘B1456’, ‘03/09/2019’, ‘18/09/2019’);
Insert into Borrow_details values (‘M3124’, ‘B2111’, ‘16/08/2019’, ‘30/08/2019’);

Queries:

- Display the members who are from the city ‘Karaikudi’
- Display the details of member, who borrowed book on ‘03/09/2019’
- Display the author details of the book ‘Software Engineering’
- Update the phone number of M. Sunil as 9865782528

Answers:

- Display the members who are from the city ‘Karaikudi’

Select * from member_details where city=’Karaikudi’

Member_id	Member_name	Member_city	Member_phone	Mem_last_paid
M1234	R. Suresh	Karaikudi	9876567890	03/07/2019
M3124	M. Sunil	Karaikudi	9786956432	03/06/2019

- Display the details of member, who borrowed book on ‘03/09/2019’

Notes

Select * from member_details where member_id = (select member_id from borrow_details where borrow_date='03/09/2019');

Member_id	Member_name	Member_city	Member_phone	Mem_last_paid
M2654	K. Mohamed	Devakottai	6345789765	12/08/2019

- Display the author details of the book 'Software Engineering'

Select * from author_details where author_id = (select author_id from borrow_details where book_id = (select book_id from book_details where book_name='Software Engineering));

Author_id	Author_name	Author_country
A2413	Pressman	USA

- Update the phone number of M. Sunil as 9865782528

Update member_details set member_phone = 9865782528 where member_name = 'M.Sunil';

Member_id	Member_name	Member_city	Member_phone	Mem_last_paid
M3124	M. Sunil	Karaikudi	9865782528	03/06/2019

References:

- https://www.techonthenet.com/oracle/index.php
- https://www.oracletutorial.com/
- https://oracle-base.com/articles/8i/
- https://www.tutorialspoint.com/plsql/
- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011
- Rajeeb C. Chatterjee, Learning Oracle SQL and PL/SQL: A Simplified Guide, PHI, 2012

NOTES

MODEL QUESTION PAPER

RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS) - LAB

1. a). Create the following table with 5 rows and perform the SQL operation:

Library (Book_id, Book_name, Barrower_Name, Subs_id, Barrow_date)

- Display the details of Books and its Barrower on a specific date

b). Design and Develop an application for student mark sheet processing.

2. a). Create the following table with 5 rows and perform the SQL operation:

Student (Reg_No, Name, Mark1, Mark2, Mark3)

- Display the details of students who got less than 40 marks in any one of the subjects

b). Design and Develop an application for Electricity bill processing.

3. a). Create the following table with 5 rows and perform the SQL operation:

Shop (item_id, item_name, prize, exp_date, qty)

- Display the details of items with stock is less than 10 in hand

b). Design and Develop an application for Library Information Management System.

4. a). Create the following table with 5 rows and perform the SQL operation:

EB_Bill (cust_id, cust_name, pre_reading, curr_reading, net_usage, amount)

- Update the net_usage by processing curr_reading and pre_reading

b). Develop a PL/SQL block to calculate amount using the following calculations:

- Net_usage * Rs. 1.50 if net_usage up to 100
- Net_usage * Rs. 2.50 if net_usage > 100 and <= 200
- Net_usage * Rs. 3.00 if net_usage > 200

5. a). Create the following table with 5 rows and perform the SQL operation:

SB_Account (cust_id, cust_name, balance)

- Display the customer details whose balance is less than 1000

b). Develop a PL/SQL block to illustrate the usage of cursor for an application of your own choice

6. a). Create the following table with 5 rows and perform the SQL operation:

Library (Book_id, Book_name, Barrower_Name, Subs_id, Barrow_date)

- Display the details of Books available with a particular barrower

b). Design and Develop an application for pay roll processing.

7. a). Create the following table with 5 rows and perform the SQL operation:

product (product_id, name, manuf_date, unit_prize)

- Display the details of products under Rs. 500 as its unit_prize

b). Develop a PL/SQL block to illustrate the usage of trigger for an application of your own choice

8. a). Create the following table with 5 rows and perform the SQL operation:

Train_booking (train_id, train_name, journey_date, seats_empty)

- Display the details of trains with empty seats > 250

b). Design and Develop an application for Gas booking.