LavaTech Technology

Call: 096073 31234
Email: info@lavatechtechnology.com
Website: https://lavatechtechnology.com
Address: Pune, Maharashtra

PUBLISHED BY LAVATECH TECHNOLOGY

lavatechtechnology.com

January 2024

YOU CAN

TOTALLY

DO THIS!

Contents

ORACLE DB PADHO!
JOB KI CHINTA MAAT KARO

1. Oracle RDBMS

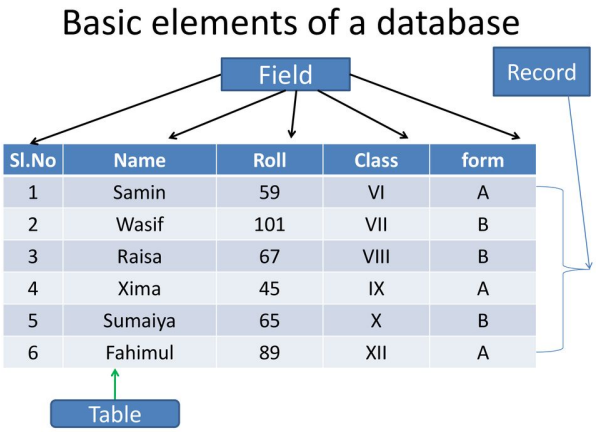
1.0.1 Data, database & DBMS

Data

- Data is collection of information that can be recorded & have meaning.

Database

- Database is collection of interrelated data stored in tables.



DBMS

- DBMS stands for database management system.
- DBMS is software that enables user to create and maintain a database.
- DBMS allows to define, construct and manipulate the database.
- Advantages of DBMS:
  - Data Independence
  - Efficient Data Access
  - Data Integrity and security
  - Data administration
  - Concurrent access and Crash recovery

1.0.2 What is RDBMS?

- RDBMS is a type of DBMS that stores and manages data in tables.
- It is designed using relational model.

Relational Model: It is a framework introduced by E.F. Codd in 1970. It organize and manage data within DBMS as follows:

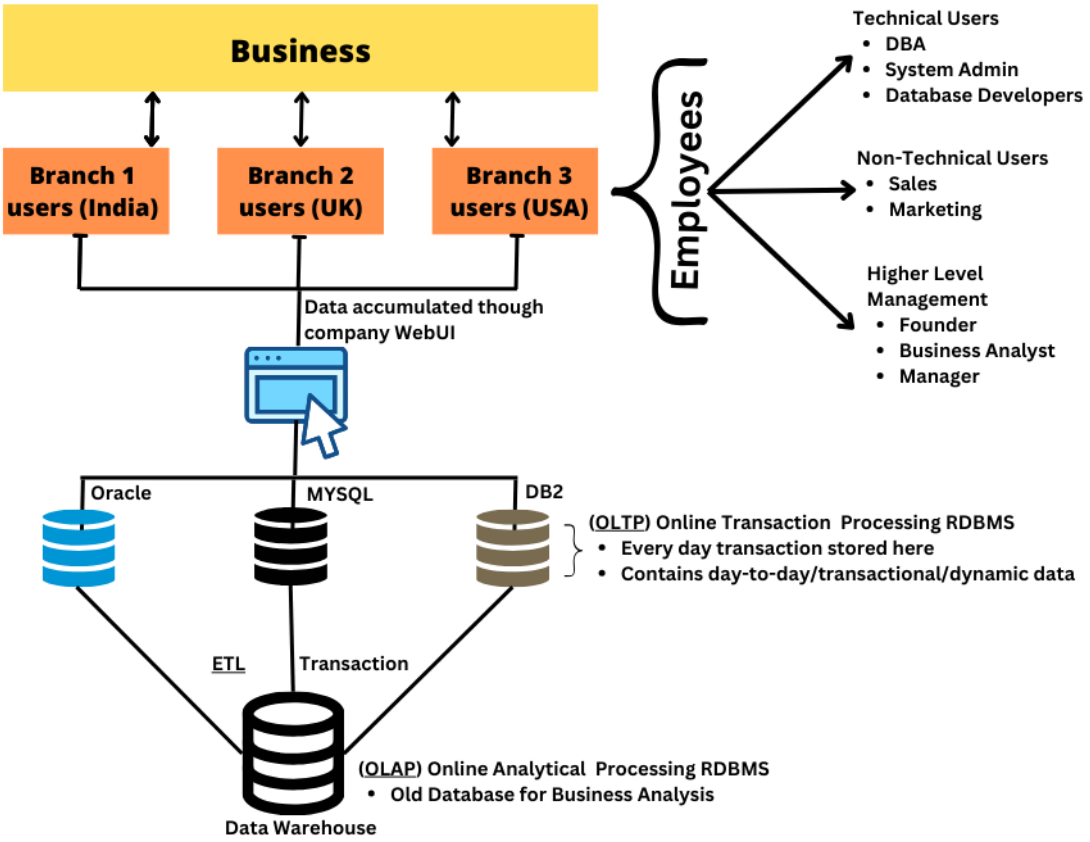
- **Tables:**
  - Tables are relations.
  - Table contains rows (records) and columns (attributes or fields).
  - Each row is unique record.
  - Each column is attribute.
- **Tables have a Primary Key:**
  - A primary key is a column (or set of columns) that uniquely identifies each table record.
  - Primary key ensures records are not duplicate.
- **Foreign Key:**
  - Table relations are established using foreign keys.
  - A foreign key in one table refers to the primary key in another table.

- **Data Integrity:**
  - Defines constraints, such as uniqueness, null values, ensures foreign keys reference valid primary keys etc.
- **Structured Query Language (SQL):**
  - Use SQL (Structured Query Language) for querying and manipulating data.
  - SQL allows users to perform operations like SELECT, INSERT, UPDATE, DELETE etc.
- **Data Independence:**
  - The relational model promotes a separation between physical storage and logical representation of data.
  - Changes to the physical storage do not affect data being accessed or queried.
- **Normalization:**
  - Organize the database tables to minimize redundancy and anomalies.
- **Adherence to Codd's Rules:**
  - Codd formulated 12 rules that makes a system fully compliant with relational model.
- **ACID Transactions**: RDBMS supports ACID (Atomicity, Consistency, Isolation, Durability) properties.

1.0.3 RDBMS v/s DBMS

RDBMS	DBMS
An RDBMS is type of DBMS based on E.F. Codd's relational model	DBMSs supports relational model, hierarchical model , network model, object-oriented model, and key-value.
RDBMS stores data in tables & use SQL for querying data.	DBMS stores data using indexing, hashing, or even simple file systems.
Eg: RDBMSs include MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server, and SQLite.	Eg: All RDBMSs are DBMSs, DBMS also includes NoSQL like MongoDB and Cassandra.

1.0.4 RDBMS in Business



OLTP

- Stands for **O**n**L**ine **T**ransaction **P**rocessing.
- It is database system to manage and process day-to-day data transactions.
- It involve activities characterized by:
  - Process transactions quickly.
  - It involve transaction for insert, update, or delete small amounts of data in database.
  - Break data into smaller, related tables to avoid data duplication.
  - Users expect quick responses for applications like e-commerce, banking systems etc.

Data warehouse

- A data warehouse is a centralized repository for storing large

volumes of data from various sources.

- Data once entered into data warehouse cannot be modified.
- Data in data warehouse is:
  - Consolidated
  - Historical Data
  - Non-volatile

OLAP

- Stands for **O**n**L**ine **A**nalytical **P**rocessing
- OLAP system are used for:
  - Data analysis
  - Reporting
  - Facilitate decision-making
  - Business intelligence activities

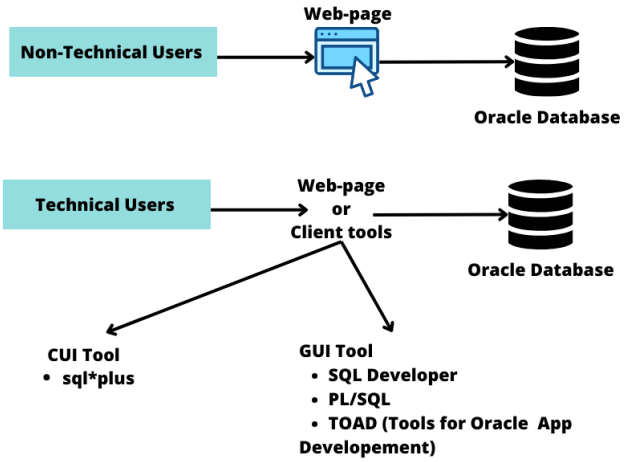
ETL

- ETL stands for Extract, Transform, Load.
- It is iterative process that runs on a scheduled basis to keep the data warehouse updated.
- Data warehousing performs below task:
  - Collect data from various sources
  - Transform it into a suitable format
  - Load it into data warehouse

1.0.5 Oracle Database Installation

- Install Oracle DB
- Install Oracle SQL developer

1.0.6 Oracle DB User



Create a local user in Oracle 21c.

- Login as system DBA:

Syntax:

```
sqlplus sys/<password>@<TNS service name> as sysdba
```

where,

- <password> is the password that you set during oracle installation.
- <TNS service name> where Transparent Network Substrate (TNS) operates for connection to Oracle databases.

Note:

You can find the current database name using below query:

Query:

```
SELECT ora_database_name FROM dual;
```

Eg:

Query:

```
>sqlplus sys/Admin12345@orcl as sysdba
```

Optionally, you can connect using command prompt with username/password:

Query:

```
> sqlplus / as sysdba
```

- Once connected to database, alter the session:

Query:

```
SQL> alter session set "_oracle_script"=true;
```

- Create a local user using below syntax:

Syntax:

```
SQL> create user <username> identified by <password>;
SQL> grant resource,connect,dba to <username>;
```

- Eg:

Query:

```
SQL> create user jack identified by Admin12345;
SQL> grant resource,connect,dba to jack;
```

- Login as local user:

Syntax:

```
sqlplus user/password
```

Eg:

Query:

```
C:\Users\Admin> sqlplus jack/Admin12345
```

- You can check current user using below command:

Query:

```
SQL> show user;
USER is "Jack"
```

- You can switch user using below command:

Query:

```
SQL> conn jack
Enter password:
Connected.
```

- You can check all user details in "ALL_USER" table:

Query:

```
SQL> select username from ALL_USER;
```

- You can clear the screen using below command:

Query:

```
SQL> cl scr;
```

- You can drop user account using below command:

Syntax:

```
drop user username;
or
drop user username CASCADE;
```

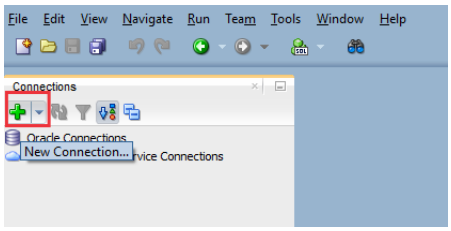
Eg:

Query:

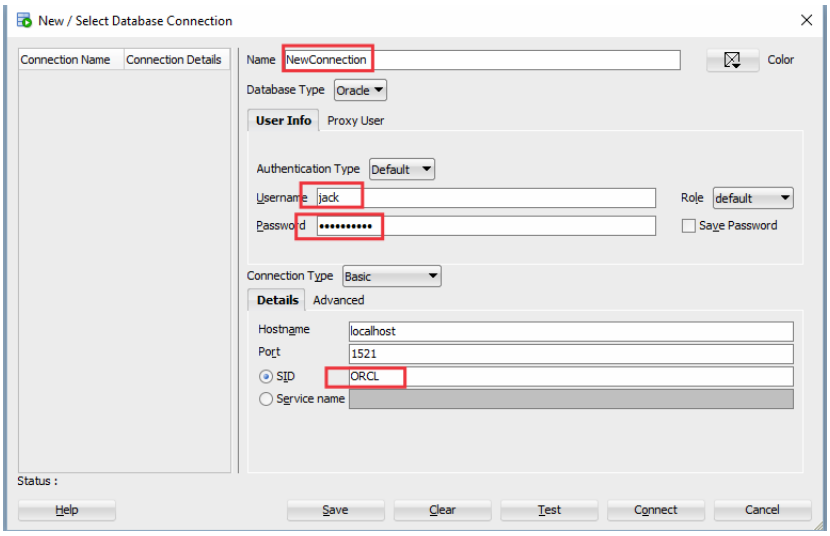
```
SQL> drop user jack;
```

Connecting to Oracle SQL developer

- Start the Oracle SQL developer
- Create a new connection by clicking the "+" symbol as shown in screenshot below:

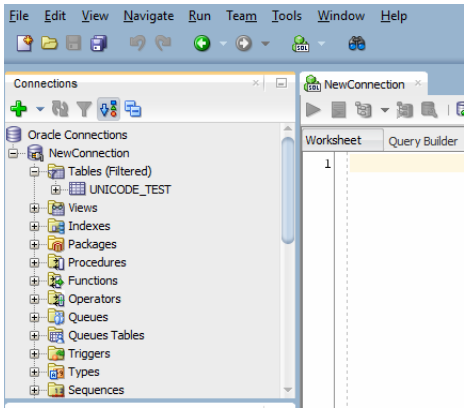


- Enter the connection details as shown and click "Test".



Once the test is successful, select "Connect".

- Enlarge the "NewConnection" dropdown and notice all the tables are displayed:

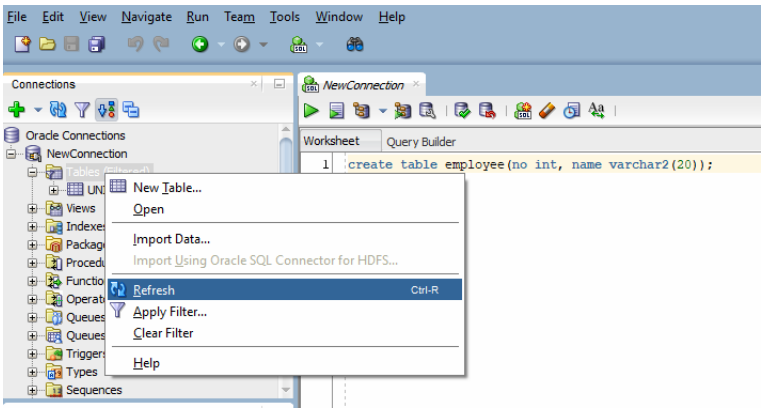


- Create a new table by executing below query in worksheet. Click the green run logo on top of worksheet to execute the query.

Query:

```
create table employee(no int, name varchar2(20));
```

- Refresh the table to see new table entry:



- Optionally you can execute below query to display all tables:

Query:

```
select * from tab;
```

1.0.7 What is SQL?

- SQL stands for **Structured Query Language**
- Pronounced as S.Q.L/Sequel.
- This language is used to communicate with Database.
- SQL require environment (like SQL *Plus) to implement it's statements
- **SQL is non-procedural language** (means contains only statement and no procedure).
- It is case-insensitive language.
- SQL is ANSI(American National Standards Institute) standards.

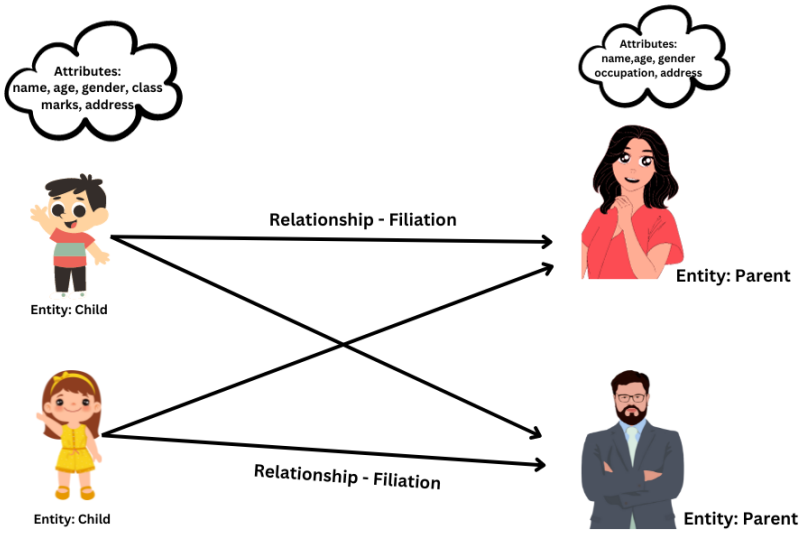
Small steps every day.....

2. ERD

2.1 ERD

2.1.1 Entity Relationship Diagram (ERD)

- An ERD is a visual representation of the data model depicting entities and their relationship with each other.
- ERD components:
  - **Entities**:
    - Entities represents real world object having some data.
    - Eg: Customer, Employee
  - **Attributes**:
    - Attributes are characteristics of entities.
    - Eg: name, age, phone-number
  - **Relationships**:
    - Relationships define how entities are connected to each other.
    - Eg: Child and Parent have filiation relationship



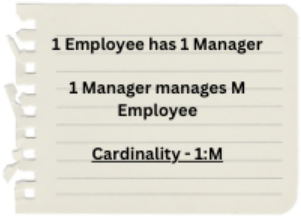
– **Cardinality**:
  * Cardinality specify how many instances of one entity are associated with instances of another entity in a relationship.
  * 3 types of cardinalities:

One-to-One	One-to-Many	Many-to-Many
1 Country - 1 Capital	1 Customer - M Order	M Book - M Author
1 Student - 1 Library Card	1 Class - M Student	M Student - M Tuition
1 Student - 1 Address	1 Manager - M Employee	M Teacher - M Subject
1 Employee - 1 Department	1 Chef - M Dish	M Patient - M Doctor

- **Primary Key**:
  - A primary key is an attribute that uniquely identifies an entity.
  - It ensures that there are no duplicate records.
- **Foreign Key**:
  - A foreign key is an attribute in one entity that refers to the primary key of another entity.
  - It establishes a link between the two entities.

Employee ID (Primary Key)	Employee Name	Manager ID (Foreign Key)
101	Raman	201
102	Raj	201
106	Ravi	202

Employee Table



Manager ID (Primary Key)	Manager Name	Address
201	Mr. Verma	Pune
202	Mr. Sharma	Mumbai

Manager Table

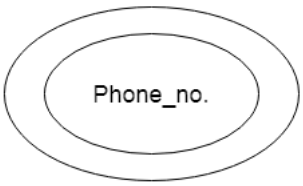
– **Diagram Symbols**: Peter Chen invented Chen ERD notation. Symbols used to represent ERD are:

* **Entity:**

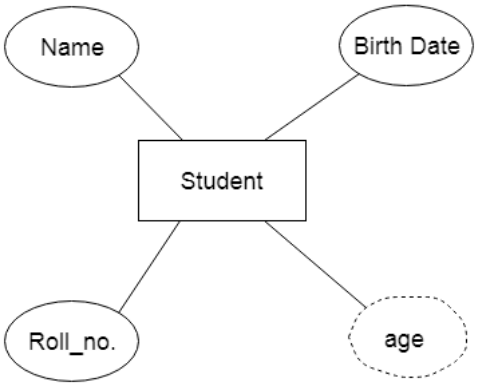
* **Attribute:**

* **Multi-valued attribute:** An attribute can have more than one value.

Eg: A student can have more than one phone number.

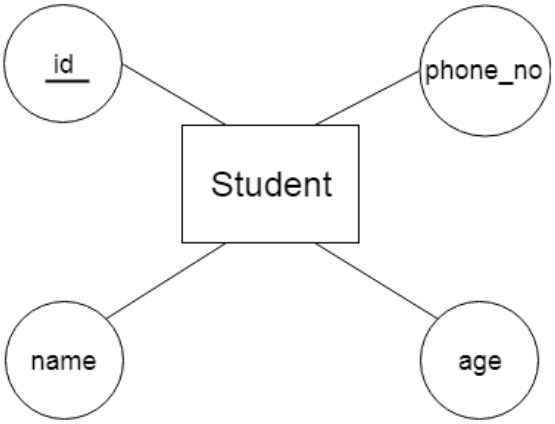


* **Derived Attribute:** An attribute that can be derived from other attribute is known as a derived attribute.
Eg: A person's age can be derived from Date of birth.



* **Primary Key attribute:**

Eg: Id is unqiue for every student



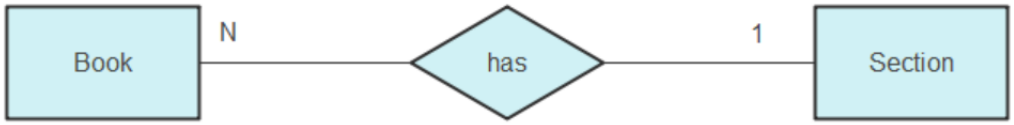
* **Relationship:**

* **Cardinality:**

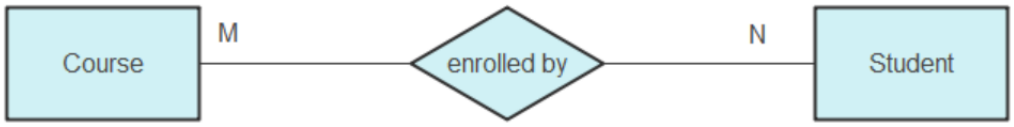
one-to-one (1:1)

one-to-many (1:N)

many-to-one (N:1)



many-to-many (M:N)



2.1.2 Relationship types

- **One to One (1:1)**
  - Each record in one entity (table) is associated with exactly one record in another entity, and vice versa.
  - Eg: Person and Passport is 1:1 relationship

Person table

Fields	Type
person_id	Primary Key
name	
date_of_birth	

Passport table

Fields	Type
passport_id	Primary Key
passport_number	
expiration_date	
person_id	Foreign Key from Person table

- **One to Many (1:M)**
  - A single record in one entity (table) can be associated with multiple records in another entity.
  - Eg: One manager manages multiple employees

Manager table

Fields	Type
id	Primary key
name	
department	

Employee table

Fields	Type
id	Primary key
name	
department	
manager_id	Foreign key

- **Many to Many (M:M)**
  - Each record in one entity can be related to multiple records in the second entity, and vice versa.
  - An intermediary table (or junction table) contains foreign keys from both entities involved in the relationship, creating a link between them.
  - Eg: One student can enroll for multiple course and 1 course can have multiple students.

Students table:

Fields	Type
student_id	Primary Key
student_name	
date_of_birth	

Courses table:

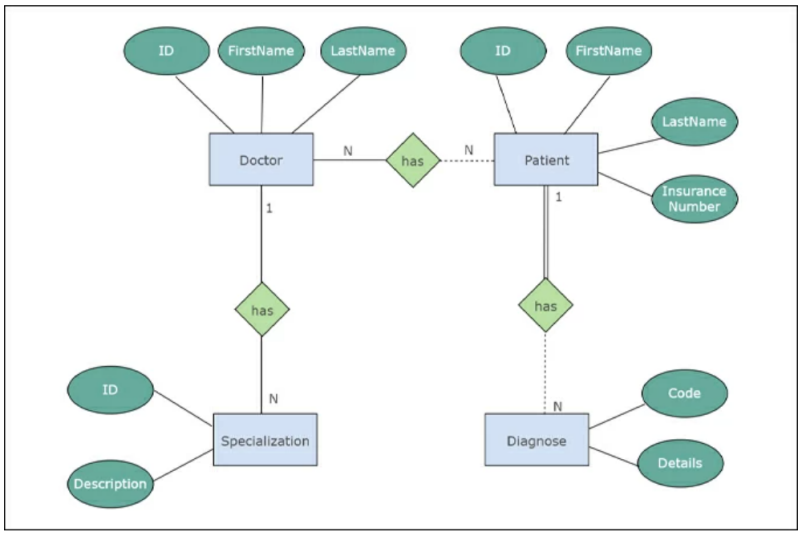
Fields	Type
course_id	Primary Key
course_name	
instructor	

Enrollment table (Junction table):

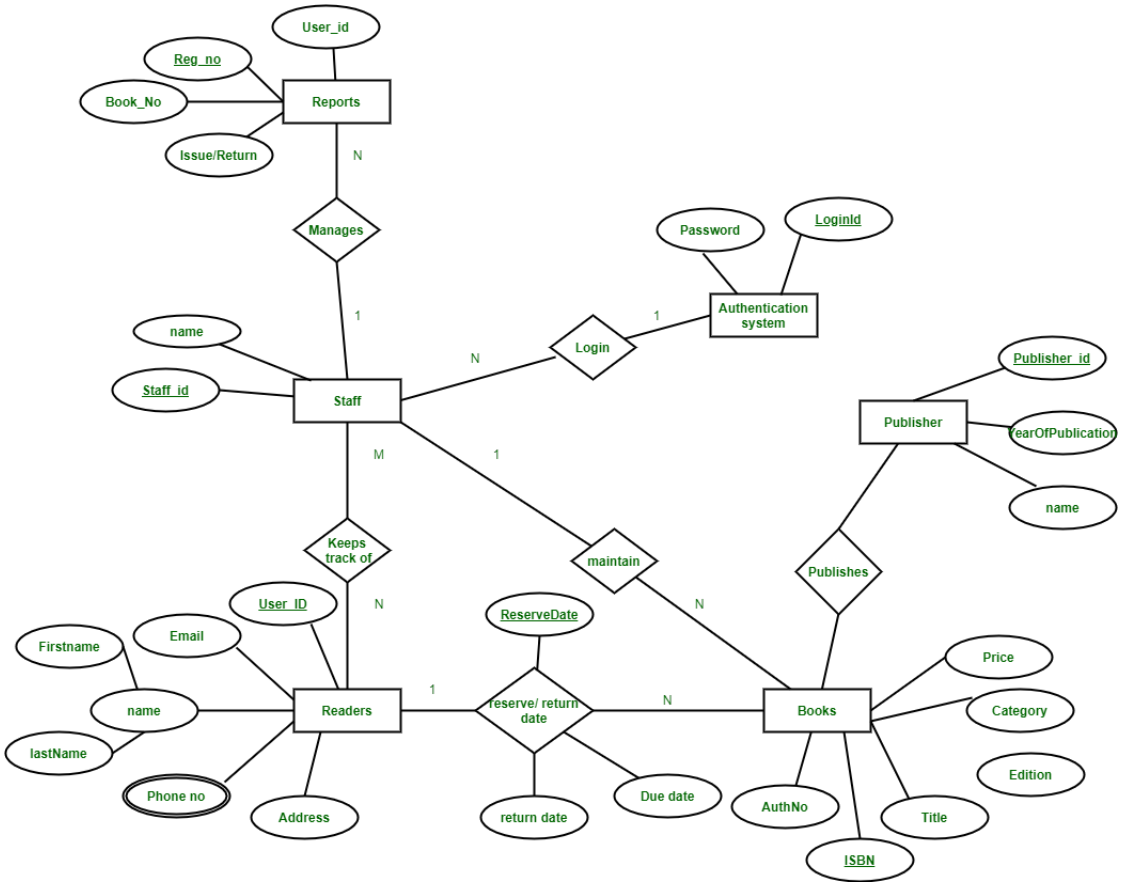
Fields	Type
enrollment_id	Primary Key
student_id	Foreign Key
course_id	Foreign Key

2.1.3 ERD example

Example 1: Docter-Patient ERD



Example 2: Library system ERD



Don't fear failure.

Not failure, but low aim is the crime.

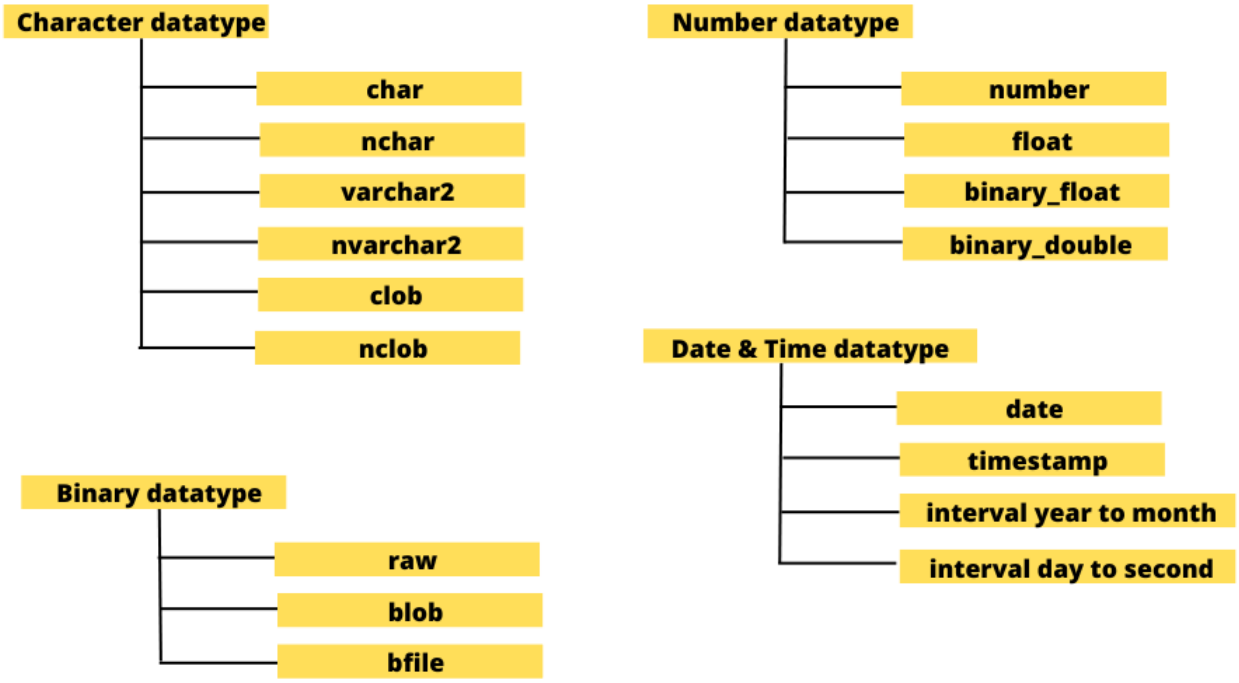
3. Datatypes in Oracle DB

3.0.1 Identifiers

An identifier refers to a name used to identify database objects such as tables, columns, indexes, views, procedures, and other database elements.

- Identifiers can consist of **letters**, **numbers**, and **underscore character** (_).
- They must **start with a letter**.
- They can be up to **30 characters** in length.
- Identifiers are **case-insensitive**. However, you can enclose identifiers in double quotes to make them case-sensitive. For example, "My_Table" and "my_table" would be treated as different identifiers.
- You **cannot use reserved words as identifiers** without enclosing them in double quotes.
- Common naming conventions is **camelCase or underscores** to separate words in identifiers (e.g., employee_id, customerName).

3.0.2 Datatypes



- **Character Data Types**
  - **char**: Fixed-length character strings
  - For example, if you define a CHAR(10) column and store 'hello' in it, it will occupy 10 bytes of storage, with trailing spaces added to fill the remaining space.
  - **Syntax:**

    char(size)

    where,

    size is optional
  - * **Max size**: 2000 chars/bytes
    * **Min size**: 1 char/byte
    * Eg:

Query:

```
create table sample(grade char, section char(3));
insert into sample values('C', 'II');
select length(section) from sample;
```

Notice the length is 3 characters.

– **nchar**: Store fixed-length unicode character strings.

Syntax:

char(size)

where,

size is optional

* **Max size**: 2000 char/byte
* **Min size**: 1 char/byte
* Eg:

Query:

```
create table sample(name nchar(20));
insert into sample values('α θ');
select length(name) from sample;
```

Notice the length is 20 characters.

Note:

* CHAR stores Unicode characters using UTF-16 encoding, but it's not specifically designed for Unicode data.
* NCHAR is explicitly designed for storing Unicode characters.

– **varchar2**: Variable-length character strings

Syntax:

varchar2(size)

where,

size is not optional

* **Max size**: 4000 characters/bytes
* Eg:

Query:

```
create table sample(name varchar2(30));
insert into sample values('Raman');
```

Notice the length is 5 characters.

– **nvarchar2**: Store variable-length unicode character data.

Syntax:

nvarchar2(size)

where,

size is not optional

* **Max size**: 4000 char/byte
* Eg:

Query:

```
create table sample(name nvarchar2(20));
insert into sample values('α θ');
```

Note:

* VARCHAR2 uses the database's default character set encoding, which might not support all Unicode characters.
* NVARCHAR2 ensuring that it can store characters from any language or script supported by Unicode.

- **CLOB**: stands for **C**haracter **L**arge **O**bject

  Syntax:

  clob

  * Store variable-length large character data using the database's character set, which may not or may be Unicode.
  * **Max size**: 4 GB
  * Eg:

    Query:

    create table sample(para clob);
    insert into sample values('α θ data');

- **NCLOB**: stands for **N**ational **C**haracter **L**arge **O**bject

  Syntax:

  nclob

  * Store variable-length large character data in the Unicode character set supporting multilingual text.
  * **Max size**: 4 GB
  * If you need to store different languages, NCLOB is the appropriate choice to ensure compared to CLOB.
  * Eg:

    Query:

    create table sample(para nclob);
    insert into sample values('α θ data');

- **Number**
  - NUMBER datatype is used to store numeric data, including **integers** and **floating-point** numbers.
  - It is a versatile datatype that can store both fixed-point and floating-point numbers with precision and scale.

Syntax:

NUMBER(precision, scale)
where,
Precision (p): Specifies the total number of digits that can be stored in the number. This includes both the digits to the left and right of the decimal point.
Scale: Specifies the number of digits to the right of the decimal point. Scale cannot be greater than precision.

- If you don't specify precision and scale, Oracle defaults to NUMBER(38,0), which means it can store numbers with up to 38 digits, all of which can be to the left of the decimal point.
- Eg:
  * NUMBER: This defines a numeric column with default precision (38) and scale (0), suitable for storing integers.
  * NUMBER(10, 2): This defines a numeric column with a precision of 10 and a scale of 2, meaning it can store numbers with up to 10 digits, 2 of which can be to the right of the decimal point.
  * NUMBER(5): This defines a numeric column with a precision of 5 and a scale of 0, suitable for storing small integers.

Note:

If you add more than 38 digits in Number datatype, Oracle DB can have incorrect behaviour and store incorrect values.

- **Float**
  - FLOAT datatype in Oracle Database is deprecated, and its use is discouraged.
  - Instead, Oracle recommends using the NUMBER datatype for

precise numeric data and the BINARY_FLOAT and BINARY_DOUBLE datatypes for floating-point numbers.

- **Binary_float**
  - It stores single-precision 32-bit floating point number.
  - Max size: 4 bytes
  - It provides a precision of approximately 6 to 7 significant decimal digits.

    Syntax:

    float

  - Eg:

    Query:

    ```
    create table sample(no binary_float);
    insert into sample values(122.2);
    ```

- **Binary_double**
  - It is double-precision 64-bit floating point number.
  - Max size: 8 bytes
  - Double precision provides approximately 15-16 decimal digits of precision.

    Syntax:

    binary_double

  - Eg:

    Query:

    ```
    create table sample(no binary_double);
    insert into sample values(122.2);
    ```

Date & Time

- **Date Datatype**
  - Default date format is **DD-MON-YYYY**
  - Complete date format is **DD-MON-YY HH:MI:SS AM**
    where,
    - **DD**: Day of month
    - **MON**: Month name (e.g: "JAN" for January)
    - **YY**: Last two digits of year (e.g: "21" for 2021)
    - **HH**: Hour in 12-hour format
    - **MI**: Minutes
    - **SS**: Seconds
    - **AM**: Time period, either "AM" or "PM"

    Eg: 26-SEP-23 03:30:00 PM
  - Storage size is 7 bytes.
  - After Oracle 9i, the allowed range is between **01-Jan-4712 BC - 31-Dec-9999**
  - Eg:

    Query:

    ```
    create table sample(joining_date date);
    insert into sample values('04-Mar-1880');
    insert into sample values('04-Mar-21');
    insert into sample values('4-Mar-98');
    ```

  - Date functions:
    - **sysdate**: Returns the current date & time.

      Eg:

      Query:

      ```
      select sysdate from dual;
      insert into sample values(sysdate);
      ```

    - **to_date**: Converts a character to DATE value using a specified format.

Eg:

Query:

```
insert into sample values(to_date('05-Dec-2022 06:34:23 PM', 'DD-MON-YYYY HH:MI:SS PM'));
```

* **extract**: Extracts a date component (e.g., year, month, day) from a DATE value.

Eg:

Query:

```
select extract(year from joining_date), extract(month from joining_date), extract(day from joining_date) from sample;
```

- **Timestamp**
  - Introduced in oracle 9i.
  - Allows date and time.

Syntax:

```
timestamp(p)
timestamp(p) with timezone
timestamp(p) with local timezone
where,
p stands for precision (optional)
```

  - **TIMESTAMP(p):** This syntax defines a TIMESTAMP data type with optional precision 'p', specifying the number of digits in the fractional seconds part. If 'p' is not specified, the default precision is 6.

Code:

```
CREATE TABLE sample (
    t1 TIMESTAMP(3)
);

insert into sample values('01-Jan-12 10:34:56.123');

insert into sample values('01-Jan-12
10:34:56.123990');

select * from sample;
```

    In this example, the precision is of 3 fractional seconds. Hence, even if you enter more than 3 fractional seconds, it would display only 3 fractional seconds. Adding "AM/PM" is optional.

  - **TIMESTAMP(p) WITH TIME ZONE:** This syntax defines a TIMESTAMP data type with time zone information. It stores

date and time information along with the time zone offset from UTC (+00:00).

Query:

```
CREATE TABLE sample (
    t1 TIMESTAMP with time zone
);

insert into sample values('01-Jan-12 10:34:56.123457');

insert into sample values('01-Jan-12 10:34:56.123457 AM ASIA/TOKYO');

insert into sample values('01-Jan-12 10:34:56.123457 AM +00:00');
select * from sample;
```

In this example, if no timezone is provided, it takes "AM ASIA/CALCUTTA" for first record.

– **TIMESTAMP(p) WITH LOCAL TIME ZONE**: This syntax defines a TIMESTAMP data type with local time zone handling. It automatically converts the time zone to the session time zone when storing and retrieving data. If "AM/PM" is not added, it by default takes "AM".

Query:

```
CREATE TABLE sample (
    t1 TIMESTAMP with local time zone
);

insert into sample values('01-Jan-12 10:34:56.123457
AM');

insert into sample values('01-Jan-12 10:34:56.123457
AM');

insert into sample values('01-Jan-12 10:34:56.123457
AM ASIA/TOKYO'); <- Error

insert into sample values('01-Jan-12 10:34:56.123457
AM +00:00'); <– Error
select * from sample;
```

Difference between normal "TIMESTAMP" and "TIMESTAMP WITH LOCAL TIME ZONE" is that the later, when you retrieve the value, it will be converted to the local time zone of the client application.

Note:

* UTC stands for Coordinated Universal Time.
* It is the primary time standard by which the world regulates clocks and time.
* UTC does not change with the seasons like local time zones do, and it is not affected by daylight saving time changes.

- **Interval Year**
  - Allows time period in terms of years and months

Syntax:

internal year(yp) to month(p)
where,
YP is year precision
p is month precision

  - Default year precision is 2
  - Display format: **YY-MM**

INTERVAL YEAR TO MONTH Literals	Meaning
`INTERVAL '120-3' YEAR(3) TO MONTH`	An interval of 120 years, 3 months; Must specify the leading field precision YEAR(3) because the value of the leading field is greater than the default precision (2 digits).
`INTERVAL '105' YEAR(3)`	An interval of 105 years 0 months.
`INTERVAL '500' MONTH(3)`	An interval of 500 months.
`INTERVAL '9' YEAR`	9 years, which is equivalent to `INTERVAL '9-0' YEAR TO MONTH`
`INTERVAL '40' MONTH`	40 months or 3 years 4 months, which is equivalent to `INTERVAL '3-4' YEAR TO MONTH`
`INTERVAL '180' YEAR`	Invalid interval because '180' has 3 digits which are greater than the default precision (2)

  - Eg:

Query:

```
create table sample(t1 interval year to month);
insert into sample values(interval '3' Year);
insert into sample values(interval '3' month);
insert into sample values('3-4');
insert into sample values('3-0');
insert into sample values('0-4');
```

- **Interval Day**
  - Allows time period is terms of days, hours, minutes, seconds and fraction

Syntax:

interval day(dp) to second(p)
where,
dp stands for day precision
p is fractional part of seconds

  - Default day precision is 2 and default p is 6
  - Display format: DD HH:MI:SS:FF

INTERVAL DAY TO SECOND Literals	Meaning
`INTERVAL '11 10:09:08.555' DAY TO SECOND(3)`	11 days, 10 hours, 09 minutes, 08 seconds, and 555 thousandths of a second.
`INTERVAL '11 10:09' DAY TO MINUTE`	11 days, 10 hours, and 09 minutes.
`INTERVAL '100 10' DAY(3) TO HOUR`	100 days 10 hours.
`INTERVAL '999' DAY(3)`	999 days.
`INTERVAL '09:08:07.6666666' HOUR TO SECOND(7)`	9 hours, 08 minutes, and 7.6666666 seconds.
`INTERVAL '09:30' HOUR TO MINUTE`	9 hours and 30 minutes.
`INTERVAL '8' HOUR`	8 hours.
`INTERVAL '15:30' MINUTE TO SECOND`	15 minutes 30 seconds.
`INTERVAL '30' MINUTE`	30 minutes.
`INTERVAL '5' DAY`	5 days.
`INTERVAL '40' HOUR`	40 hours.
`INTERVAL '15' MINUTE`	15 minutes.
`INTERVAL '250' HOUR(3)`	250 hours.
`INTERVAL '15.6789' SECOND(2,3)`	Rounded to 15.679 seconds. Because the precision is 3, the fractional second '6789' is rounded to '679'

  - Eg:

Query:

```
create table sample(t1 interval day to second);
insert into sample values(INTERVAL '3 04:30:15' DAY TO SECOND);
insert into sample values(Interval '3' hour);
insert into sample values(Interval '3' minute);
```

Binary datatype:

- **Raw**:
  - Store fixed-length binary & hexadecimal (starting Oracle 9i) data.
  - Eg: encryption keys or binary-encoded data.
  - **Dataset** for raw datatype:
    - Binary data can be 0 or 1
    - Hex data is not case-sensitive and can be 0-9 or a-f
    - Both data must be enclosed in ''
    - Max size: 2000 bytes
    - Each character take ½ byte of storage space.

Syntax:

raw(size)
where, **size is not optional**

  - Eg:

Query:

```
create table sample(data raw(20));
insert into sample values('10101011');
insert into sample values('aaff11');
```

- **Blob (Binary large object)**:
  - Store unstructured binary data, such as images, audio files, video files etc.

- Max size: 128 TB

Syntax:

```
blob
```

- Eg: Insert blank entry in place of image

Query:

```
CREATE TABLE image_table(id number, image BLOB);
insert into image_table values(101, EMPTY_BLOB());
```

- Eg: Insert image in blob datatype using **utl_raw.cast_to_raw()**

Query:

```
CREATE TABLE image_table(id number, image BLOB);
insert into image_table values(101, utl_raw.cast_to_raw('C:\Users\Admin\
Pictures\test.png') );

select * from image_table;
```

Notice the output should show **blob** or the image in image column.

To display image path itself:

Here's a basic approach using Oracle SQL as an example, assuming the image path is stored as text within the BLOB data:

Query:

```
SELECT SUBSTR(TO_CHAR(image), INSTR(TO_CHAR(image), 'path=') + 1) FROM image_table;
```

- **Bfile (binary file locator)**:
  - Store references to binary files stored in filesystem outside the database.
  - Data will be stored outside the database.
  - It is read-only by default.

    Syntax:

    bfile

  - Using bfile:
    - Create a directory variable using below query:

      Syntax:

      create or replace directory <var> as <path>;

      As admin user, execute below query:

      Syntax:

      grant read,write on directory <var> to <user>;

    - To access the data stored in a BFILE column, you use the BFILENAME function:

      Syntax:

      create table xyz(fname **bfile**);
      insert into xyz values(**bfilename('<var>','<filename>')**);

– Eg: Create a table with bfile field and set a directory path:

Query:

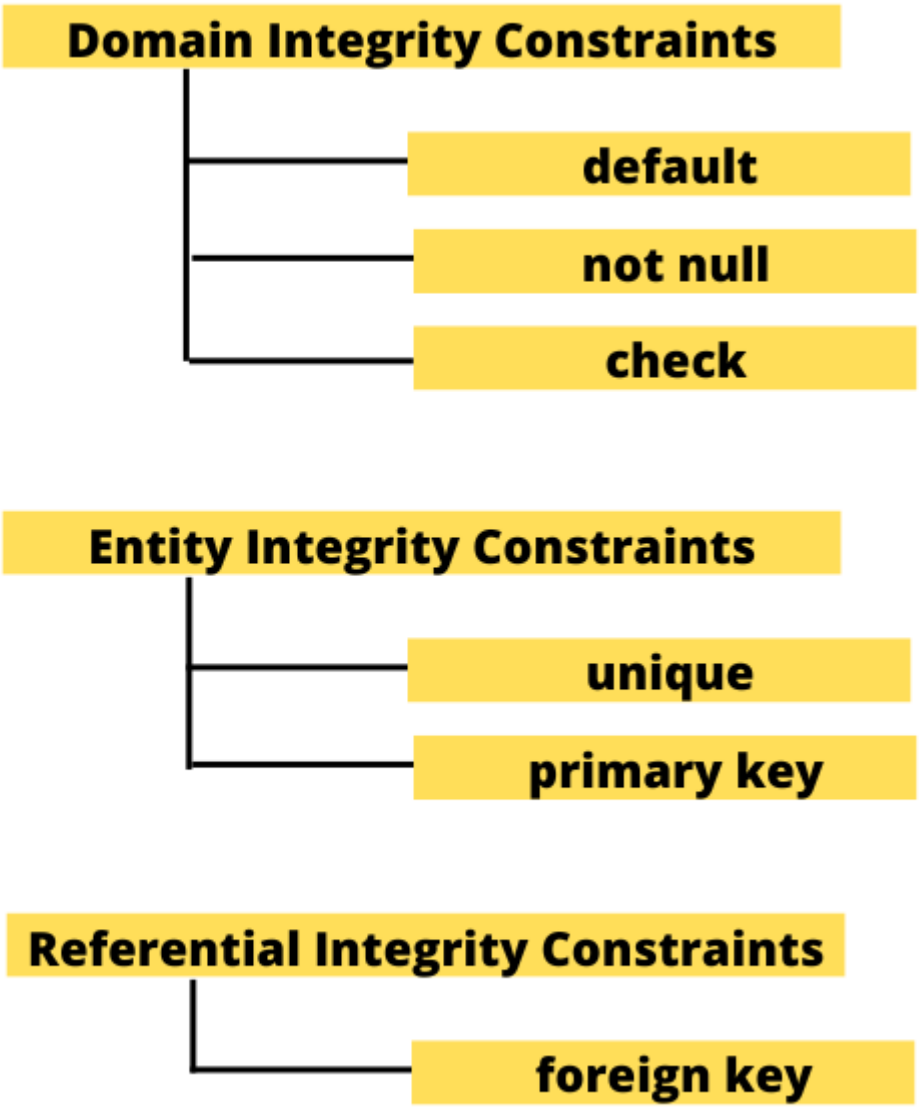
```
create table sample(fname bfile);
create or replace directory dir1 as
'C:\Users\Admin\Pictures\');

– As admin user execute below query
grant read,write on directory dir1 to jack;

insert into sample values(bfilename('dir1','1.png'));
```

3.1 Constraints

- Constraints are rules or conditions enforced on table data.
- It maintains data accuracy, integrity and consistency.
- Types of constraints:



3.1.1 Domain Integrity Constraints

default:

- DEFAULT constraint specify default value for column in table.
- Used when INSERT statement does not provide a value for that column.

Syntax:

```
CREATE TABLE table_name (
    column1 datatype DEFAULT default_value,
    column2 datatype DEFAULT default_value,
    – other columns
);
```

Eg:

Query:

```
CREATE TABLE employees (
    employee_id NUMBER(5),
    first_name VARCHAR2(50) DEFAULT 'John',
    dob date DEFAULT SYSTIMESTAMP,
    joining_date date DEFAULT SYSDATE, );

insert into employees(employee_id, first_name) values (101, 'Ravi');

insert into employees(employee_id, first_name, dob_date, joining_date ) values (101, default, default, default);

CREATE TABLE employees (
    first_name VARCHAR2(50) DEFAULT 'John',
    dob date DEFAULT SYSTIMESTAMP,
    joining_date date DEFAULT SYSDATE,
);
```

```
insert into employees(first_name, dob_date, joining_date )
values (default, default, default);
```

not null:

- NOT NULL ensures a column cannot contain NULL values.

Note:

Default has highest priority than Not NULL, so first define default than not null.

Syntax:

```
CREATE TABLE table_name (
    column_name datatype NOT NULL,
    – other columns
);
or
CREATE TABLE table_name (
    column_name datatype CONSTRAINT constraint_name
NOT NULL,
    – other columns
);
```

Eg:

Query:

```
create table student(
    no number CONSTRAINT c1 not null
);
CREATE TABLE students (
    student_id NUMBER(5) NOT NULL,
    first_name VARCHAR2(50) NOT NULL,
);
```

```
create table students(
    no number DEFAULT 0 NOT NULL,
);
```

check

- CHECK constraint enforces a condition on column or entire table.
- It ensures data entered into the column meets specific criteria or rules.
- Column level check constraints:

Syntax:

```
CREATE TABLE table_name (
column_name datatype CONSTRAINT constraint_name
CHECK (condition),
– other columns
);
or
CREATE TABLE table_name (
column_name datatype CHECK (condition),
– other columns
);
```

- Table level check constraints:

Syntax:

```
CREATE TABLE table_name (
column_name datatype,
CONSTRAINT constraint_name CHECK (condition)
);
```

Eg:

Query:

```
CREATE TABLE orders (
    order_id NUMBER(5),
    order_total NUMBER(10, 2) CHECK (order_total > 0)
);
```

Query:

```
CREATE TABLE emp(
    eno number,
    CONSTRAINT con1 CHECK(eno IN (1,2,3))
);
CREATE TABLE emp1(
    eno number
    DEFAULT 10 CHECK(eno in (10,20,30))
    NOT NULL
);
```

3.1.2 Entity Integrity Constraints

unique

- **Unique** does not allow **duplicate** values
- It allows **NULL** values (Any number of NULLs)
- Can be defined at single or multiple **columns** or at **table level**

Syntax:

```
// Column level
CREATE TABLE table_name (
    column1 datatype UNIQUE,
     column2 datatype CONSTRAINT constraint_name
UNIQUE
);
// Table level
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
CONSTRAINT constraint_name UNIQUE(column1, col-
umn2,..)
);
```

Eg: Column level

Query:

```
CREATE TABLE product(
    id number UNIQUE,
    name varchar(50) UNIQUE
);
CREATE TABLE product(
name varchar(50) CONSTRAINT con1
UNIQUE(column_name)
);
```

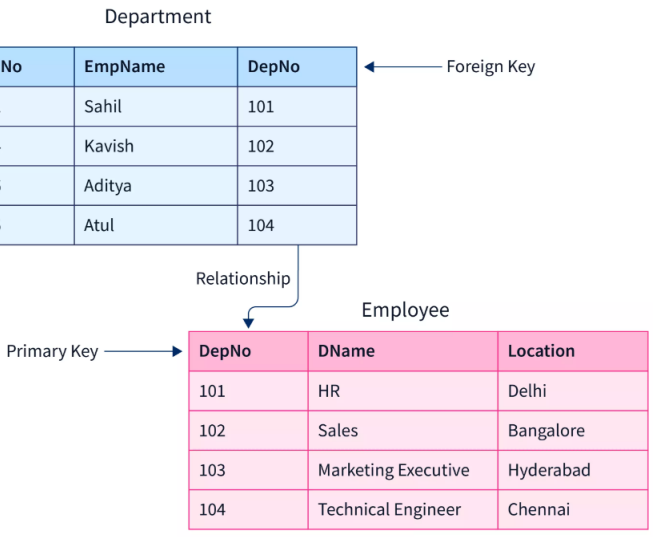
Eg: Table level

Query:

```
create table product1(
    no number,
    constraint constraint_name unique_no unique(no)
);
create table product2(
    no number,
    name varchar2(20),
    unique(no,name)
);
```

Primary key

- A primary key(PK) constraint can:
  - Enforces data uniqueness.
  - Prevent duplicate or NULL values in column
- Other tables can reference the PK column(s) as foreign keys.
- PK on multiple columns is called composite primary key.
- Oracle database automatically creates a unique index on the primary key column(s).



Syntax:

```
// Single column primary key
CREATE TABLE example (
    column1 datatype PRIMARY KEY,
    column2 datatype,
);
// Multiple column primary key
CREATE TABLE example (
    column1 datatype,
    column2 datatype,
    CONSTRAINT pk_example PRIMARY KEY(column1,
column2...)
);
```

Eg:

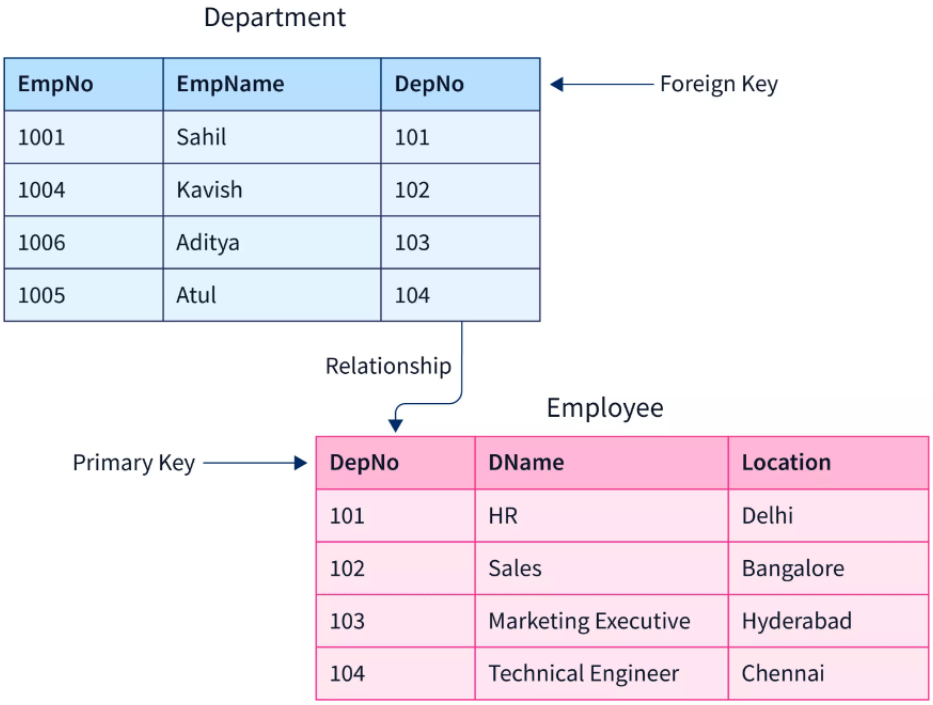
Query:

```
create table product(
    no number primary key,
    name varchar2(30)
);
CREATE TABLE sample(
    no number,
    name varchar2(30),
    CONSTRAINT pk PRIMARY KEY(no, name)
);
```

3.1.3 Referential Integrity Constraints

foreign key

- A foreign key establishes link between two tables.
- It ensures one table's column(foreign key) correspond to another table's primary/unique key(referenced key).
- Reference column may be present in same/different table.
- It allows:
  - Duplicate values
  - NULL values
- It can be defined on single/multiple column (compsite key).
- Can be defined at table level or column level.



Syntax:

```
// Table level
CREATE TABLE child_table (
    column1 datatype,
    column2 datatype,
    referenced_column datatype,
    CONSTRAINT fk_name FOREIGN KEY
    (referenced_column)
    REFERENCES parent_table (primary_key_column)
);
// Column level
CREATE TABLE child_table (
    column1 datatype,
    column2 datatype,
    referenced_column datatype CONSTRAINT fk_name
    REFERENCES parent_table(primary_key_column)
);
```

Eg:

Query:

```
CREATE TABLE product(
    pno number PRIMARY KEY,
    pname varchar(20),
);
CREATE TABLE productOrder(
    no number,
    pno number CONSTRAINT fk
    REFERENCES product(pno)
);
```

```
CREATE TABLE productOrder(
    no number,
    CONSTRAINT fk FOREIGN KEY(pno)
    REFERENCES product(pno)
);
```

ON DELETE SET NULL

- Specify a foreign key column should be set NULL when corresponding row in parent table is deleted.

Syntax:

```
CREATE TABLE child_table (
    column1 datatype,
    parent_reference datatype,
    CONSTRAINT fk_parent_reference
    FOREIGN KEY (parent_reference)
    REFERENCES parent_table(parent_id)
    ON DELETE SET NULL
);
```

- Eg:

Query:

```
CREATE TABLE product(
    pno number PRIMARY KEY,
    pname varchar(20)
);
```

```
create table productOrder(
    id number,
    pno number,
    odate date,
    constraint fk foreign key(pno)
    references product(pno)
    on delete set null
);
insert into product values(101,'cadbury');
insert into product values(102,'5-star');
insert into product values(103,'kitkat');
insert into productOrder values(201,101,SYSDATE);
insert into productOrder values(202,101,SYSDATE);
insert into productOrder values(203,103,SYSDATE);
insert into productOrder values(204,102,SYSDATE);
delete from product where pno=101;
```

Query:

```
select * from productOrder;
ID        PNO        ODATE
201        -        06-OCT-23
202        -        06-OCT-23
203        103        06-OCT-23
204        102        06-OCT-23
```

ON DELETE CASCADE

- Specify when a row in the parent table is deleted, all related rows in the child table should be deleted.

Syntax:

```
CREATE TABLE child_table (
    column1 datatype,
    parent_reference datatype,
    CONSTRAINT fk_parent_reference
    FOREIGN KEY (parent_reference)
    REFERENCES parent_table(parent_id)
    ON DELETE CASCADE
);
```

- Eg:

Query:

```
CREATE TABLE product(
    pno number PRIMARY KEY,
    pname varchar(20)
);
create table productOrder(
    id number,
    pno number,
    odate date,
    constraint fk foreign key(pno)
    references product(pno)
    on delete cascade
);
insert into product values(101,'cadbury');
insert into product values(102,'5-star');
insert into product values(103,'kitkat');
insert into productOrder values(201,101,SYSDATE);
```

```
insert into productOrder values(202,101,SYSDATE);
insert into productOrder values(203,103,SYSDATE);
insert into productOrder values(204,102,SYSDATE);
delete from product where pno=101;
```

Query:

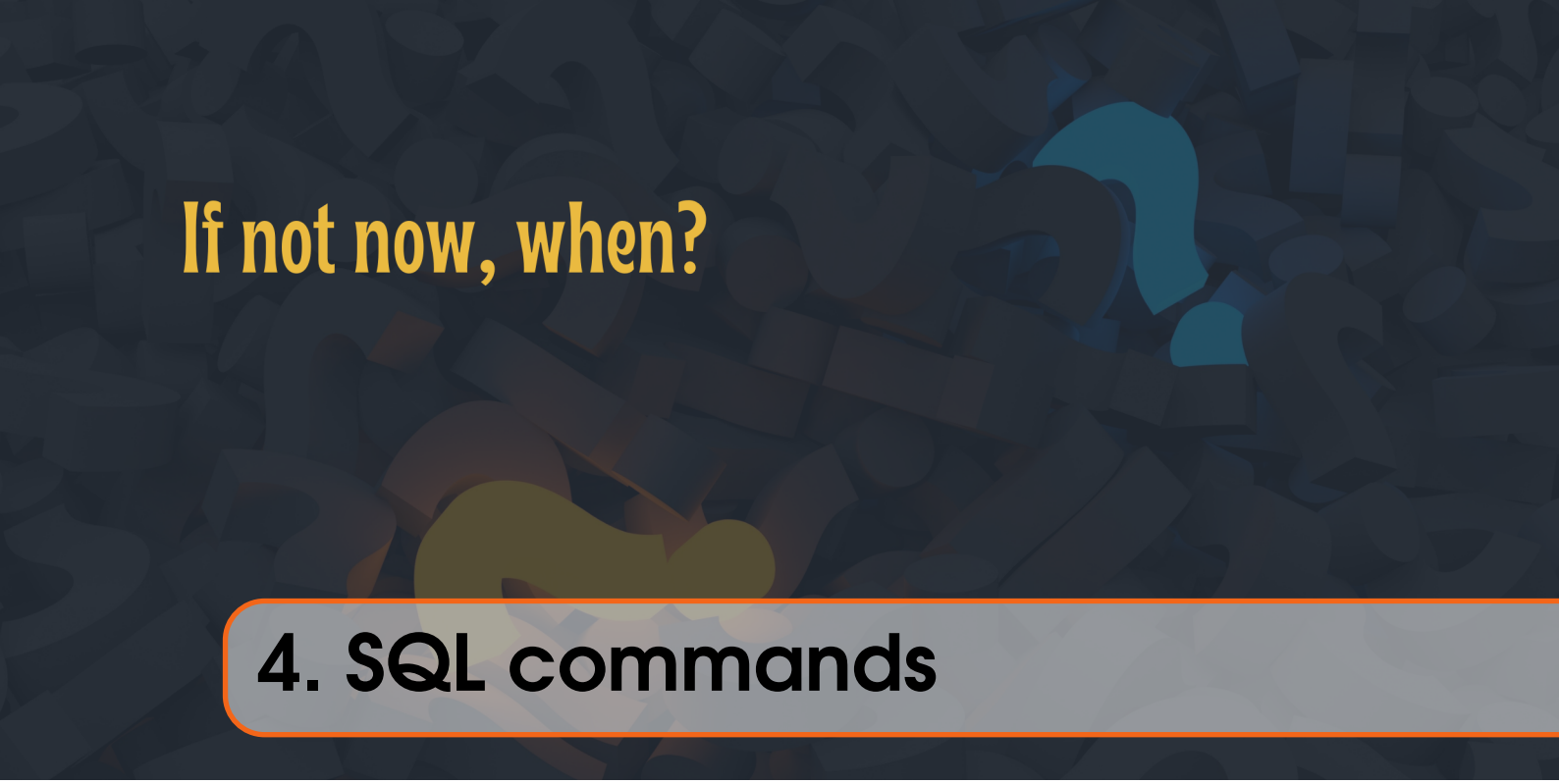
```
select * from productOrder;
ID        PNO        ODATE
203        103        06-OCT-23
204        102        06-OCT-23
```

3.2 Composite key

- A composite key (or compound key) consists of two or more columns in a table, used together to uniquely identify each row.
- A composite key relies on multiple columns to ensure uniqueness.
- It can be defined only on unique key, primary key & foreign key.
- Composite key is possible only at table level.
- Eg:

Query:

```
create table product(
    no number,
    name varchar2(30),
    constraint pk primary key (no,name)
);
```



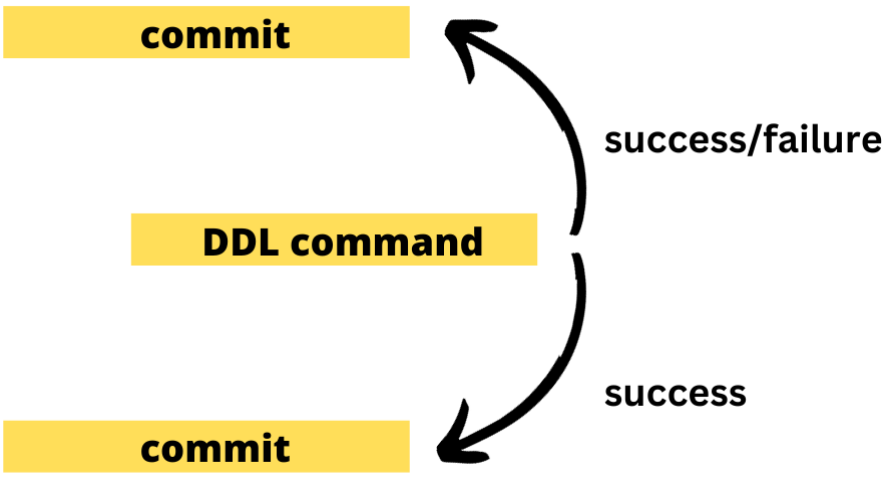
4. SQL commands

SQL commands are used for managing and querying data. SQL commands in Oracle are:

- **DDL** - Data Definition Language
- **DML** - Data Manipulation Language
- **TCL** - Transaction Control Language
- **DCL** - Data Control Language

4.1 Data Definition Language (DDL)

- DDL command deal with the structure of objects.
- DDL interacts with database directly, hence there is implicit commit before & after statements.



- Due to this, you cannot undo (rollback) the changes done by DDL statement.
- As DDL commands interact with database directly, they are faster (performance-wise).
- DDL commands:
  - create
  - alter
  - truncate
  - rename
  - drop
  - flashback
  - purge
  - comment
- flashback and purge where introduced in Oracle 10g
- Let's see each of these in detail.

4.1.1 CREATE

CREATE query is used to create:

- **TABLE**: Table is made up of rows and columns.

Let's see each of these in detail.

CREATE TABLE: Used to create a table

- Create table schema:

Syntax:

```
CREATE TABLE your_table_name (
    column1_name datatype1,
    column2_name datatype2,
    column3_name datatype3,
    – Add more columns as needed
);
```

Eg:

Query:

```
CREATE TABLE employees (
    employee_id NUMBER(5) PRIMARY KEY,
    first_name VARCHAR2(50),
    last_name VARCHAR2(50),
    hire_date DATE,
    salary NUMBER(10, 2)
);
```

- Create table using existing tables (with data)

Syntax:

```
CREATE TABLE your_table_name as select column_name from table_name;
```

Egs:

Query:

```
create table sample1 as select * from dual;
create table sample2 as select dummy from dual;
create table sample1 as select * from dual where 1 = 1;
create table sample1 as select * from dual where dummy = 'x';
```

- Create table using existing tables (only structure)

Query:

```
create table sample1 as select * from dual where 1 = 2;
```

4.1.2 TRUNCATE

- TRUNCATE deletes all rows from table.
- TRUNCATE does not generate any undo logs, so you cannot roll back the operation.
- Truncate does not reclaim space immediately; it releases space to the tablespace, making it available for future inserts.
- Use DROP STORAGE option to immediately release the space.

Syntax:

```
truncate table table_name [drop storage];
```

Query:

```
truncate table dummy;
truncate table dummy drop storage;
```

4.1.3 RENAME

Use RENAME command to rename a table or a column.

- Renaming a table:

Syntax:

```
RENAME old_table_name TO new_table_name;
```

Eg:

Query:

```
RENAME employees TO staff;
```

- Renaming a column:

Syntax:

```
ALTER TABLE table_name RENAME COLUMN
old_column_name TO new_column_name;
```

Eg:

Query:

```
alter table employee rename column name TO first_name;
```

- Renaming a constraint:

Syntax:

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

Eg:

Query:

```
CREATE TABLE products(
    no number,
    name varchar2(20),
    CONSTRAINT p PRIMARY KEY(no)
);
ALTER TABLE products RENAME CONSTRAINT p TO
pk;
```

4.1.4 ALTER

The ALTER command modify existing tables, indexes, constraints, etc.

Syntax:

```
ALTER TABLE table_name ADD/MODIFY/DROP/RENAME/SET
UNUSED/ENABLE/DISABLE;
```

- **Rename column**

  Syntax:

  ```
  ALTER TABLE table_name RENAME column
  old_column TO new_column;
  ```

  Eg:

  Query:

  ```
  CREATE TABLE product(no NUMBER);
  ALTER TABLE product RENAME column no TO id;
  ```

- **Add single column**

Syntax:

ALTER TABLE table_name **ADD** column_name datatype;
or
ALTER TABLE table_name **ADD** (column_name datatype);

Eg:

Query:

```
CREATE TABLE product(no NUMBER);
ALTER TABLE product ADD name VARCHAR2(30);
or
ALTER TABLE product ADD (name VARCHAR2(30));
```

Query:

```
DESC product;
Column          Null?          Type
NO              -              NUMBER
NAME            -              VARCHAR2(30)
```

- **Add multiple column**

Syntax:

ALTER TABLE table_name **ADD**(column1_name datatype,column2_name datatype);

Eg:

Query:

```
CREATE TABLE product(no NUMBER);
ALTER TABLE product ADD(name VARCHAR2(20),
grade CHAR(3));
```

Query:

```
DESC product;
Column          Null?          Type
NO          -          NUMBER
NAME          -          VARCHAR2(20)
GRADE          -          CHAR(3)
```

- **Modify single column size**

Syntax:

```
ALTER TABLE table_name MODIFY column_name
DATATYPE(SIZE);
or
ALTER TABLE table_name MODIFY(column_name
DATATYPE(SIZE));
```

Eg:

Query:

```
CREATE TABLE product(name VARCHAR2(10));
ALTER TABLE product MODIFY(name VARCHAR2(25));
```

Query:

```
DESC product;
Column          Null?          Type
NAME          -          VARCHAR2(25)
```

- **Modify multiple column size**

Syntax:

```
ALTER TABLE table_name MODIFY(column1_name
DATATYPE(SIZE), column2_name DATATYPE(SIZE));
```

Eg:

Query:

```
CREATE TABLE product(no NUMBER(2), name VARCHAR2(10));
ALTER TABLE product MODIFY(no NUMBER(3), name VARCHAR2(25));
```

Query:

```
DESC product;
Column        Null?        Type
NO            -            NUMBER(3,0)
NAME          -            VARCHAR2(25)
```

- **Drop single column**

Syntax:

```
ALTER TABLE table_name DROP COLUMN column_name;
or
ALTER TABLE table_name DROP(column_name);
```

Eg:

Query:

```
CREATE TABLE product(no NUMBER(2), name VARCHAR2(10));
ALTER TABLE product DROP COLUMN name;
```

Query:

```
DESC product;
Column        Null?        Type
NO            -            NUMBER(2,0)
```

- **Drop multiple column**

Syntax:

ALTER TABLE table_name **DROP**(column1_name, column2_name);

Eg:

Query:

```
CREATE TABLE product(no NUMBER(2), name VARCHAR2(10), grade CHAR(2));
ALTER TABLE product DROP(name, grade);
```

Query:

```
DESC product;
Column        Null?        Type
NO            -            NUMBER(2,0)
```

- **Hide single column**

Syntax:

ALTER TABLE product SET UNUSED COLUMN column_name;
or
ALTER TABLE product SET UNUSED(column_name);

Eg:

Query:

```
CREATE TABLE product(no NUMBER(2), name VARCHAR2(10), grade CHAR(2));
insert into product values(11, 'cadbury','AB');
insert into product values(12, 'munch','AC');
alter table product set unused column grade;
```

Query:

```
select * from product;
NO          NAME
11          cadbury
12          munch
```

- **Hide multiple column**

Syntax:

```
ALTER TABLE product SET UNUSED(column1_name, column2_name);
```

Eg:

Query:

```
CREATE TABLE product(no NUMBER(2), name VARCHAR2(10), grade CHAR(2));
insert into product values(11, 'cadbury','AB');
insert into product values(12, 'munch','AC');
alter table product set unused(name,grade);
```

Query:

```
select * from product;
NO
11
12
```

Why should we hide the column and not drop it directly?

Ans:

- Dropping column with high volume of data during working hours can consume a lot of time.
- Hence, during working hours, we can hide the column instead of dropping it
- In non-working hours, we can remove the column.

After hiding a column, can we add a new column with same name?

Ans:

- Yes, as once the column is hidden, we cannot unhide it again. We can only drop the column after hiding it.
- Hence we can add a colum with same name.

4.1.5 DROP

- You can drop (delete) database objects using the DROP statement.
- Deleted objects are moved to Recycle Bin.

Note:

- Until Oracle 9i, dropping an object deletes it permanently.
- In Oracle 10g, recycle bin concept was introduced.

Drop table

Syntax:

```
DROP TALBE table_name
```

Eg:

Query:

```
DROP TABLE product;
```

4.1.6 FLASHBACK

- A deleted table is present in recycle bin.
- Eg: Checking table in recycle bin

Query:

```
SHOW RECYCLEBIN;
no rows selected
```

Query:

```
CREATE TABLE emp(no NUMBER);
INSERT INTO emp VALUES(101);
DROP TABLE emp;
```

Query:

```
SHOW RECYCLEBIN;
ORIGINAL NAME        RECYCLEBIN NAME        OB-
JECT TYPE        DROP TIME
_____
EMP §BIN9820203/==2930        TABLE        2023-02-
08:13:11:11
```

- A Flashback Query allows to retrieve deleted table.

Syntax:

```
FLASHBACK TABLE table_name TO BEFORE DROP;
```

- Eg:

Query:

```
FLASHBACK TABLE emp TO BEFORE DROP;
```

Query:

```
SELECT * FROM emp;
NO
____
101
```

4.1.7 PURGE

- Introduced in Oracle 10g, the PURGE statement permanently removes object from the **Recycle Bin**.

Syntax:

```
PURGE TABLE table_name
```

- Eg:

Query:

```
PURGE TABLE product;
```

Confirm table is deleted from Recycle Bin.

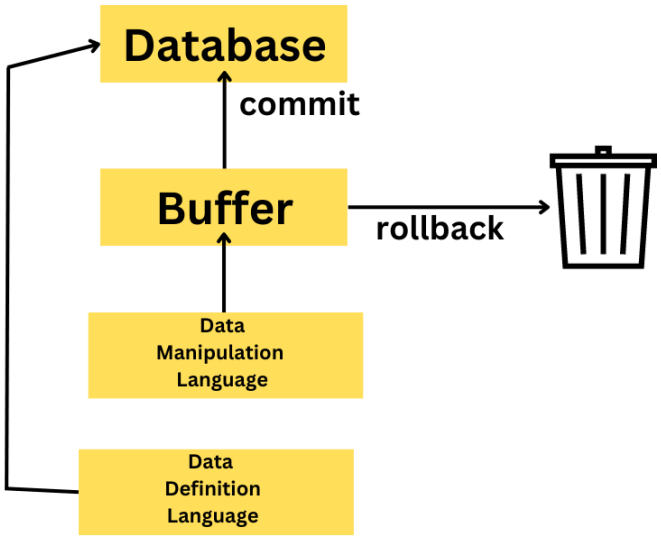
Query:

```
SHOW RECYCLEBIN;
```

4.2 Data Manipulation Language (DML)

- DML commands deal with data only.
- DML commands interact with **buffer first**.
- Data will be stored in buffer initially.
- On commit, the buffer content will be transferred to database.
- You can undo (**rollback**) the changes before commit.
- Due to buffer, the performance is relatively slow.
- Row-level locks occur implicitly on the modified rows:

  - Write operations in DML places **implicit row-level lock on rows** being modified for that particular user.
  - If 100 rows are deleted/modified, those row will be locked for that user.
  - Once record is locked, other users cannot modify the row unless lock owner releases the lock.
  - However locked row is readable by other users.



- DML commands:
  - insert
  - update
  - delete
  - merge

4.2.1 INSERT

- INSERT statement adds one or more rows of data to an already created table.

Syntax:

INSERT INTO table_name[(col1, col2, col3...)] VALUES(value1, value2, value3,...);

Eg:

Query:

```
CREATE TABLE product(id NUMBER, name VARCHAR2(20));
INSERT INTO PRODUCT(id,name) VALUES(101,'cadbury');
INSERT INTO PRODUCT VALUES(102,'munch');
```

- **'&'** is default substitution variable wherein it will ask user to enter the value.

Syntax:

INSERT INTO table_name VALUES(&col1, &col2, &col3,...);

Eg:

Query:

```
INSERT INTO PRODUCT VALUES(&no, '&name');
Enter value for no: 101
Enter value for name: cadbury
old 1: insert into product values(&no, '&name')
new 1: insert into product values(101, 'cadbury')
1 row created.
```

Query:

```
INSERT INTO PRODUCT VALUES(&a, '&b');
Enter value for no: 102
Enter value for name: munch
old 1: insert into product values(&a, '&b')
new 1: insert into product values(102, 'munch')
1 row created.
```

- Insert record using records of another table:

Syntax:

```
INSERT INTO table_name SELECT col1,col2,.. FROM
another_table_name;
```

Eg:

Query:

```
INSERT INTO product SELECT * FROM items;
INSERT INTO product(id, name) SELECT id,name FROM
items;
```

4.2.2 UPDATE

UPDATE statement modifies existing records in a table.

Syntax:

```
Updating all records:
UPDATE table_name SET col1 = value1, col2 = value2,...;
Updating particular record:
UPDATE table_name SET col = value WHERE CONDITION;
```

Eg: Update all records

Query:

```
UPDATE emp SET ename = 'test';
```

Eg: Update particular records

Query:

```
UPDATE emp SET ename = 'test' WHERE eno = 101;
```

4.2.3 DELETE

- DELETE query remove specific rows from a table:

  Syntax:

  ```
  DELETE FROM table_name WHERE conditions;
  ```

- DELETE query remove all rows from a table:

  Syntax:

  ```
  DELETE FROM table_name;
  ```

  Eg:

  Query:

  ```
  DELETE FROM product;
  DELETE FROM product WHERE id=101;
  ```

4.2.4 MERGE

- Support insert and update statement in single statement.
- Introduced in Oracle 9i.
- To insert & update records:

Syntax:

```
MERGE INTO target_table alias1
    USING source_table alias2
    ON (condition)
    WHEN MATCHED THEN
    UPDATE SET column1 = value1, column2 = value2,
...
    WHEN NOT MATCHED THEN
    INSERT (column1, column2, ...)
    VALUES (value1, value2, ...);
```

Note:

Columns names present in condition statement must not be present in update clause.

Eg: Consider below table:

Query:

```
create table product1(no number, name varchar2(20));
insert into product1 values(101,'Cadbury');
insert into product1 values(102,'Munch');
insert into product1 values(103,'Perk');
```

Create another empty table:

Query:

```
create table product2(no number, name varchar2(20));
```

Using merge, you can insert/update records from product1 table to product2:

Query:

```
MERGE INTO product2 p2
    USING product1 p1
    ON (p1.no = p2.no)
    WHEN MATCHED THEN
    UPDATE SET p2.name = p1.name
    WHEN NOT MATCHED THEN
    INSERT (no, name)
    VALUES (p1.no, p1.name);
```

4.3 Data Query Language (DQL)

Data Query Language (DQL) is used to fetch the data from database. It uses only one command i.e SELECT.

4.3.1 SELECT

Used to select the attribute based on the condition described by WHERE clause.

Syntax:

```
SELECT column1, column2, ...
FROM table_name;
or
SELECT column1, column2, ...
FROM table_name
WHERE [condition];
```

The [condition] can be any SQL expression, specified using comparison or logical operators like **>, <, =, <>, >=, <=, LIKE, NOT, IN, BETWEEN**

etc.

Eg:

Query:

```
CREATE TABLE emp(id NUMBER, name VARCHAR2(20));
INSERT INTO emp VALUES(101, 'Ram');
INSERT INTO emp VALUES(102, 'Ravi');
INSERT INTO emp VALUES(103, 'Sham');
```

Query:

```
select id,name
2 from emp
3 where id>102;

ID          NAME
_______-        _________-
103          Sham
```

Variations with select query:

- **Wildcard character**: * is the wildcard character to select all columns in a table.
  Eg:

  Query:

  ```
  SELECT * FROM employees;
  ```

- **Column Aliases**:
  - Make column names more readable by setting alias for column name.
  - You can use single quotes (’), double quotes (") and square brackets ([]) to create an alias.

- Eg:

Query:

```
SELECT FName AS "First Name",
MName AS 'Middle Name',
LName AS [Last Name]
FROM employees;
```

- If the alias has a single word you can write it without quotes or brackets.
- Eg:

Query:

```
SELECT
FName AS FirstName,
LName AS LastName
FROM employees;
```

- **COUNT returns total number of records**:

Syntax:

```
SELECT Count(*) column_name FROM table_name;
```

Eg:

Code:

```
CREATE TABLE corder (
order_id NUMBER,
customer_id NUMBER,
order_amount NUMBER
);
insert into corder values(101, 201, 3400);
insert into corder values(102, 202, 400);
insert into corder values(103, 201, 100);
insert into corder values(104, 202, 2300);
```

Query:

```
SELECT COUNT(*) total FROM corder;
4
```

- **Selecting with table alias**:

Query:

```
SELECT e.Fname, e.LName
FROM Employees e
```

The Employees table is given the alias 'e' directly after the table name.

- **Selecting with more than 1 condition**
  - **AND**: Makes sure all condition are true.

Query:

```
SELECT name FROM persons WHERE gender = 'M' AND age > 20;
```

  - **OR**: Makes sure at least once condition is true.

Query:

```
SELECT name FROM persons WHERE gender = 'M' OR age < 20;
```

  - These keywords can be combined to allow for more complex criteria combinations:

Query:

```
SELECT name
FROM persons
WHERE (gender = 'M' AND age < 20)
OR (gender = 'F' AND age > 20);
```

- **Selecting with Aggregate functions**:

- **AVG()**: Return the average of values selected.

Query:

```
SELECT AVG(Salary) FROM employees;
SELECT AVG(Salary) FROM employees where DepartmentId = 1;
```

- **MIN()**: Returns the minimum of values selected.

Query:

```
SELECT MIN(Salary) FROM employees;
```

- **MAX()**: Returns the maximum of values selected.

Query:

```
SELECT MAX(Salary) FROM employees;
```

- **COUNT()**: Returns the count of values selected.

Query:

```
SELECT COUNT(*) FROM employees;
SELECT COUNT(*) FROM employees WHERE managerId IS NOT NULL
SELECT COUNT(managerId) FROM employees;
```

Count can also be combined with the distinct keyword for a DISTINCT count.

Query:

```
Select COUNT(DISTINCT DepartmentId) FROM employees;
```

- **SUM()**: Returns the sum of the values selected for all rows.

Query:

```
SELECT SUM(Salary) FROM employees;
```

- **Select with condition of multiple values from column**:

Query:

```
SELECT * FROM Cars WHERE status IN ('Waiting', 'Working')
or
SELECT * FROM Cars WHERE (status = 'Waiting' OR status = 'Working')
```

Note:

value IN (<value list>) is a shorthand for disjunction (logical OR).

- **Selection with sorted results**:

Query:

```
SELECT * FROM employees ORDER BY name;
SELECT * FROM employees ORDER BY name DESC;
SELECT * FROM employees ORDER BY name ASC;
SELECT * FROM employees ORDER BY lame ASC, name ASC
```

To save retyping the column name in the ORDER BY clause, use column's number. Column numbers start from 1.

Query:

```
SELECT Id, FName, LName, PhoneNumber FROM employees ORDER BY 3
```

Use ORDER BY with TOP to return the top x rows based on a column's value:

Query:

```
SELECT TOP 5 DisplayName, Reputation FROM Users ORDER BY Reputation desc
```

Order by Alias:

Query:

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY jd, rep
```

Sorting by multiple columns:

Query:

```
SELECT DisplayName, JoinDate, Reputation FROM Users
ORDER BY JoinDate, Reputation
```

- **Selecting with null**:

Query:

```
SELECT Name FROM Customers WHERE PhoneNumber
IS NULL
```

Note:

Selection with nulls take a different syntax. Don't use =, use IS NULL or IS NOT NULL instead.

- **Select distinct (unique values only)**

Query:

```
SELECT DISTINCT ContinentCode FROM Countries;
```

- **Select rows from multiple tables**:

Query:

```
SELECT * FROM table1, table2;
SELECT table1.column1, table1.column2, table2.column1
FROM table1, table2
```

- **GROUP BY**

- Results of a SELECT query can be grouped by one or more columns using the GROUP BY statement.
- This generates a table of partial results, instead of one result.
- GROUP BY can be used with aggregation functions using the HAVING statement to define how non-grouped columns are aggregated.
- It might be easier if you think of GROUP BY as "for each" for the sake of explanation.
- Eg: Consider below table:

EmpID	MonthlySalary
1	200
1	300
2	300

Query:

```
SELECT EmpID, SUM (MonthlySalary)
FROM Employee
GROUP BY EmpID
```

1	500
2	300

- **Filter GROUP BY results using a HAVING clause**
  - A HAVING clause is used in conjunction with the GROUP BY clause to filter the results of a query based on the result of an aggregate function.
  - Eg: Consider below table

Query:

```
CREATE TABLE orders (
    order_id NUMBER,
    customer_id NUMBER,
    order_amount NUMBER
);
INSERT INTO orders VALUES (1, 101, 500);
INSERT INTO orders VALUES (2, 101, 700);
INSERT INTO orders VALUES (3, 102, 300);
INSERT INTO orders VALUES (4, 103, 900);
INSERT INTO orders VALUES (5, 103, 600);
```

Retrieve only those customers who have a total order value greater than 1000:

Query:

```
SELECT customer_id, SUM(order_amount) AS total_order_value
FROM orders
GROUP BY customer_id
HAVING SUM(order_amount) > 1000;
```

4.3.2 DESCRIBE

DESCRIBE or **desc**

- In Oracle Database, the DESCRIBE statement is used to retrieve metadata information about a database object, such as a table, view, or synonym.
- It provides details about the structure of the object, including column names, data types, constraints, and other properties.

Syntax:

```
describe  table_name;
or
Desc  table_name;
or
desc  schema_name.table_name;
```

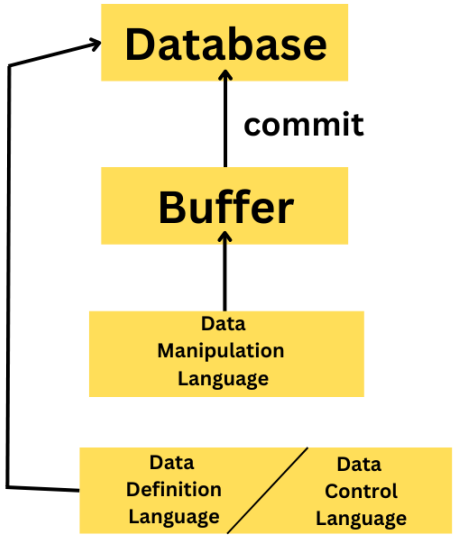
Mostly, the schema_name is the user name. Eg:

Query:

```
desc  jack.emp1;
```

4.4 Data Control Language (DCL)

- Data Control Language(DCL) command deal with **privileges** only & deal with database directly.
- DCL command enforces an implicit commit before and after statement because of direct commit, hence rollback is not possible.
- Performance is faster due to direct interaction with database.



What is Privileges?

- Privilege is right to execute a particular type of SQL statement
- It allows/prevents the right to:
  - Connect to the database
  - Create a table
  - Select rows from someone else's tables
  - Execute stored procedures

4.4.1 GRANT

- GRANT statement give specific privileges or permissions to users or roles, allowing them to perform operations on database objects such as tables, views, procedures, and more.

Syntax:

```
GRANT privilege_list ON object_name TO user;
```

where,

- **privilege_list**: Include **SELECT, INSERT, UPDATE, DELETE, EXECUTE, and ALL(which grants all available privileges)**
- **object_name**: Name of the database object (e.g, table, view, procedure) on which you are granting privileges.
- **user**: User to which you are granting the privileges.

Eg:

Query:

```
GRANT SELECT ON employees TO user1;
GRANT ALL ON orders TO user2;
GRANT ALL ON orders TO user2, user1;
```

- GRANT statement with the **"WITH GRANT OPTION"** clause means to grant a privilege to a user along with the permission to further grant that same privilege to others.

  Eg:

Query:

```
GRANT SELECT ON employees TO user1 WITH GRANT OPTION;
```

Note:

- You cannot execute grant on multiple objects in single grant statement.
- User can grant privileges only those privileges he/she has to other users.
- But not those privilege that he/she has not to other users.
- Person who have granted access can only revoke the permission.

4.4.2 REVOKE

- Revokes granted permissions from a user & role.

Syntax:

```
REVOKE privilege_list ON object_name FROM user
[CASCADE CONSTRAINTS];
```

where,

- **privilege_list**: Include SELECT, INSERT, UPDATE, DELETE, EXECUTE, and ALL (which revokes all available privileges).
- **object_name**: Name of object (e.g., table, view, procedure).
- **user**: User or other entity.
- **CASCADE CONSTRAINTS**: Revokes privilege & any dependent privileges.

Eg:

Query:

```
REVOKE SELECT ON employees FROM user1;
REVOKE ALL ON employees FROM user2;
REVOKE CREATE SESSION FROM user3;
REVOKE SELECT, INSERT ON employees FROM user4
CASCADE CONSTRAINTS;
```

4.5 Transaction Control Language (TCL)

What is a transaction?

- A set of DML operations with commit or rollback is a transaction.
- DML read is not starting point of transaction.
- Transaction starts with write operation & commit or rollback is ending point of transaction.
- This commit or rollback can be explicit or implicit by database server.

Transaction Control Language (TCL)

- TCL manages database transactions like:
  - Committing changes
  - Rolling back changes
  - Managing the isolation level of transactions

4.5.1 COMMIT

When you execute COMMIT, the following actions occur:

- Changes of all transaction are saved permanently to database.
- Locks acquired during the transaction are released, allowing other users to access the affected data.
- The changes become visible to other transactions and users.

All DDL commands causes implicit commit, while you would need to perform implicit command for all DML commands.

Syntax:

```
COMMIT;
```

Eg:

Query:

```
// Implicit commit
CREATE TABLE product(no NUMBER, name VARCHAR2(20));

// Explicit commit required
INSERT INTO demo VALUES(101,'chair');
INSERT INTO demo VALUES(102,'mirror');
COMMIT;

// Implicit commit
ALTER TABLE product RENAME COLUMN no TO id;

// Explicit commit required
UPDATE product SET id=103 WHERE id=102;
COMMIT;
```

4.5.2 ROLLBACK

- A rollback reverts any changes made within the current transaction to the last committed state.

Syntax:

```
rollback;
or
roll;
```

Eg:

Query:

```
INSERT INTO product VALUES(102,'MUNCH');
COMMIT;
INSERT INTO product VALUES(101,'Cadbury');
ROLLBACK;
```

Query:

```
SELECT * FROM product;
NO  NAME
--- ----------
102 Munch
```

Whatever you are, be a good one!

- Abraham Lincoln

5. Joins

5.1 Set Operators

Consider below 2 tables:

Code:

```
create table employee(id number, first_name varchar2(20), second_name varchar2(20));
insert into employee values(101, 'Ravi', 'Sharma');
insert into employee values(102, 'Kavi', 'Verma');
insert into employee values(103, 'Jimmy', 'Kumar');
insert into employee values(104, 'Josh', 'Kumar');

create table contractor(id number, first_name varchar2(20), second_name varchar2(20));
insert into contractor values(101, 'Jimmy', 'Kumar');
insert into contractor values(102, 'Jack', 'Snow');
insert into contractor values(103, 'Penny', 'Copper');
insert into contractor values(104, 'Kavi', 'Verma');
```

- **UNION**:
  - Combine the result set of two or more queries without duplication.
  - Sort data in ascending order based on the first column.
  - Column names of the first query are displayed as column headings of the iutput.
  - Do not use the query with NULL columns as the first, if so, use aliases for the NULL columns.
  - Number of columns should match in all the queries.
  - Use NULL columns, where sufficient number of columns are not present.
  - Datatypes of the corresponding columns should match.

Query:

```
SELECT id, first_name, second_name
FROM employee
UNION
SELECT id, first_name, second_name
FROM contractor;
ID          FIRST_NAME          SECOND_NAME
101          Jimmy          Kumar
101          Ravi          Sharma
102          Jack          Snow
102          Kavi          Verma
103          Jimmy          Kumar
103          Penny          Copper
104          Josh          Kumar
104          Kavi          Verma

SELECT first_name, second_name
FROM employee
UNION
SELECT first_name, second_name
FROM contractor;
FIRST_NAME          SECOND_NAME
Jack          Snow
Jimmy          Kumar
Josh          Kumar
Kavi          Verma
Penny          Copper
Ravi          Sharma
```

- **UNION ALL**:
  - If you want to include duplicate rows in the result set, use **UNION ALL** operator instead of UNION.

Query:

```
SELECT  first_name,  second_name
FROM  employee
UNION ALL
SELECT  first_name,  second_name
FROM  contractor;

FIRST_NAME  SECOND_NAME
Ravi          Sharma
Kavi          Verma
Jimmy          Kumar
Josh          Kumar
Jimmy          Kumar
Jack          Snow
Penny          Copper
Kavi          Verma
```

- **INTERSECT**:
  - Common records from two or more queries without duplication.
  - Sorts data in ascending order based on the first column.
  - Column names of the first query are displayed as Column headings of the output.
  - Do not use the query with NULL columns as the first, if so, use aliases for the NULL columns.
  - Number of columns should match in all the queries.
  - Must use NULL columns, where sufficient number of columns are not present. item Datatypes of the corresponding columns should match.

Query:

```
SELECT first_name, second_name
FROM employee
INTERSECT
SELECT first_name, second_name
FROM contractor;

FIRST_NAME          SECOND_NAME
Jimmy               Kumar
Kavi                Verma
```

- **MINUS**:
  - Resultant rows in the first query after eliminating the common rows of the second query.
  - Sorts data in ascending order based on the first column.
  - Column names of the first query are displayed as column headings of the output.
  - Do not use the query with NULL columns as the first, if so, use aliases for the NULL columns.
  - Number of columns should match in all the queries.
  - Must use NULL columns, where sufficient number of columns are not present.
  - Datatypes of the corresponding columns should match.

Query:

```
SELECT  first_name,  second_name
FROM  employee
MINUS
SELECT  first_name,  second_name
FROM  contractor;
FIRST_NAME          SECOND_NAME
Josh          Kumar
Ravi          Sharma
```

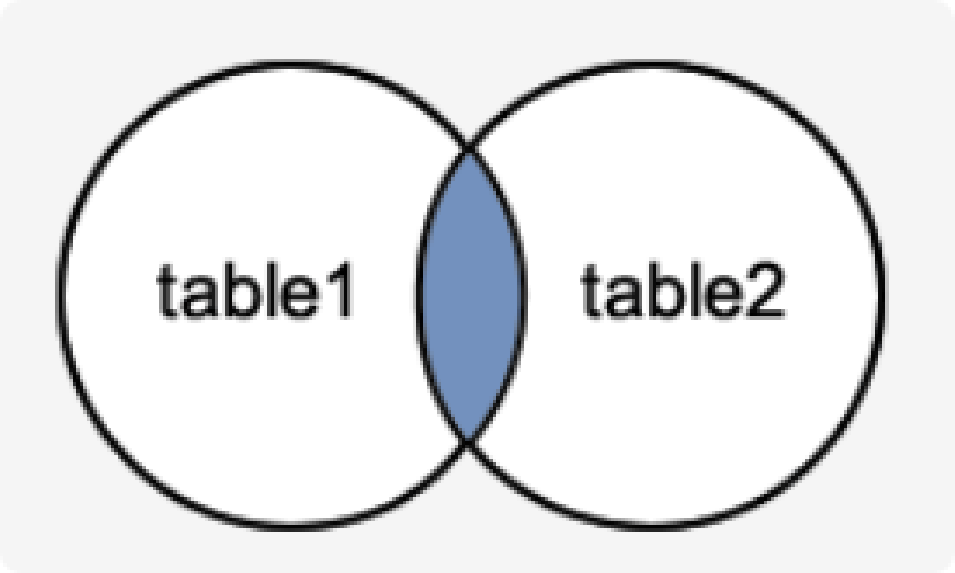
5.2 What is Join?

- JOINS are used to retrieve data from multiple tables.
- Types of Join:
  1. Inner Joins or Simple Joins
  2. Outer Joins
     - Left Outer Joins or Left Joins
     - Right Outer Joins or Right Joins
     - Full Outer Joins or Full Joins
  3. Self Joins or Inner Self Join
  4. Cross Joins or Cartesian Products

  Let's see each of these in detail

5.2.1 Inner Join

Inner join returns all rows from multiple tables where the join condition is met.



Consider below 2 tables:

Code:

```
create table employee(id number, first_name varchar2(20), second_name varchar2(20));
insert into employee values(101, 'Ravi', 'Sharma');
insert into employee values(102, 'Kavi', 'Verma');
insert into employee values(103, 'Jimmy', 'Kumar');
insert into employee values(104, 'Josh', 'Kumar');

create table contractor(id number, first_name varchar2(20), second_name varchar2(20));
insert into contractor values(101, 'Jimmy', 'Kumar');
insert into contractor values(102, 'Jack', 'Snow');
insert into contractor values(103, 'Penny', 'Copper');
insert into contractor values(104, 'Kavi', 'Verma');
```

Eg: Inner Join

Query:

```
select * from employee
inner join contractor
on employee.first_name = contractor.first_name;
```

```
ID          FIRST_NAME          SECOND_NAME          ID
    FIRST_NAME        SECOND_NAME
_____________________________________________
103         Jimmy         Kumar         101         Jimmy         Kumar
102         Kavi          Verma         104         Kavi          Verma
```

INNER JOIN could be rewritten using the older implicit syntax as follows:

Query:

```
SELECT *
FROM employee, contractor
WHERE employee.first_name = contractor.first_name;
```

5.2.2 Outer Join

An outer join returns matching rows in both tables, as well as any unmatched rows from one or both tables, depending on the type of outer join used.

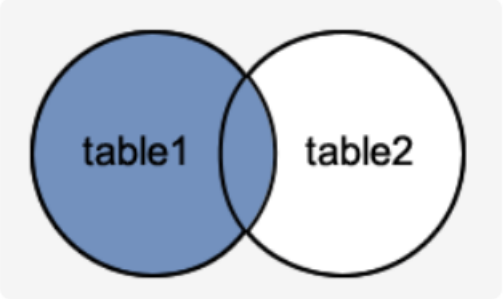
Consider below 2 tables:

Query:

```
CREATE TABLE employee(id number, first_name varchar2(20), second_name varchar2(20));
INSERT INTO employee VALUES(101, 'Ravi', 'Sharma');
INSERT INTO employee VALUES(102, 'Kavi', 'Verma');
INSERT INTO employee VALUES(103, 'Jimmy', 'Kumar');
INSERT INTO employee VALUES(104, 'Josh', 'Kumar');

CREATE TABLE contractor(id number, first_name varchar2(20), second_name varchar2(20));
INSERT INTO contractor VALUES(101, 'Jimmy', 'Kumar');
INSERT INTO contractor VALUES(102, 'Jack', 'Snow');
INSERT INTO contractor VALUES(103, 'Penny', 'Copper');
INSERT INTO contractor VALUES(104, 'Kavi', 'Verma');
```

- **LEFT OUTER JOIN (or LEFT 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).

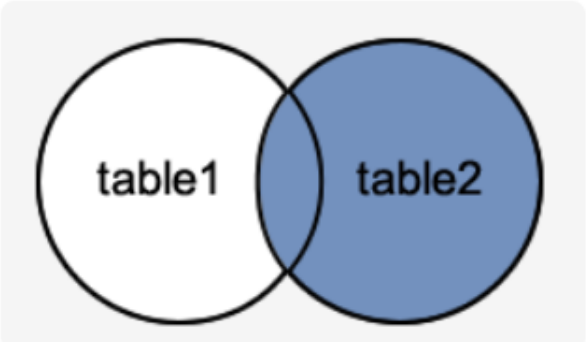


Query:

```
SELECT * FROM employee
LEFT OUTER JOIN contractor
ON employee.first_name = contractor.first_name;
or
SELECT * FROM employee
LEFT JOIN contractor
ON employee.first_name = contractor.first_name;
```

```
ID          FIRST_NAME          SECOND_NAME
 ID        FIRST_NAME        SECOND_NAME
-------------------------------------------------------
103         Jimmy         Kumar         101         Jimmy
Kumar
102          Kavi         Verma         104          Kavi
Verma
101         Ravi         Sharma         -         -         -
104         Josh         Kumar         -         -         -
```

- **RIGHT OUTER JOIN (or RIGHT 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).

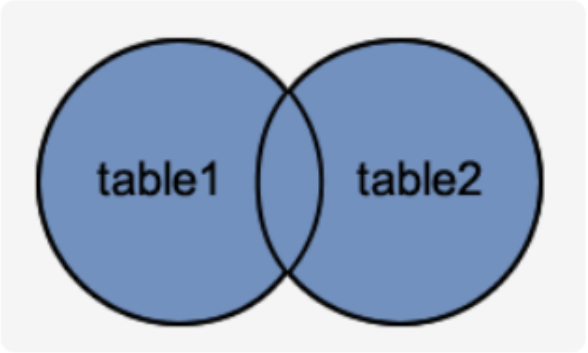


Query:

```
SELECT * FROM employee
RIGHT OUTER JOIN contractor
ON employee.first_name = contractor.first_name;
or
SELECT * FROM employee
RIGHT JOIN contractor
ON employee.first_name = contractor.first_name;
```

```
ID      FIRST_NAME      SECOND_NAME      ID
   FIRST_NAME      SECOND_NAME
102      Kavi      Verma      104      Kavi
Verma
103      Jimmy      Kumar      101      Jimmy
Kumar
-      -      -      103      Penny      Copper
-      -      -      102      Jack      Snow
```

- **Full Outer Joins or Full Joins:** Returns all rows from the LEFT-hand table and RIGHT-hand table with nulls in place where the join condition is not met.



Query:

```
SELECT * FROM employee
FULL JOIN contractor
ON employee.first_name = contractor.first_name;
or
SELECT * FROM employee
FULL OUTER JOIN contractor
ON employee.first_name = contractor.first_name;
```

```
ID          FIRST_NAME          SECOND_NAME
 ID        FIRST_NAME        SECOND_NAME
-------------------------------------------------------------
103         Jimmy         Kumar         101         Jimmy
Kumar
-           -             -             102         Jack          Snow
-           -             -             103         Penny         Copper
102         Kavi          Verma         104         Kavi
Verma
101         Ravi          Sharma        -           -             -
104         Josh          Kumar         -           -             -
```

5.2.3 Self-Join

The self ioin is joining a table by itself. In self join, you have to create the alias for the table name.

Consider below table:

Query:

```
CREATE TABLE Employee(
    EmployeeID INT, FullName VARCHAR(20),
    Gender VARCHAR(10),
    ManagerID INT
);
INSERT INTO Employee VALUES(1, 'Pranaya', 'Male', 3);
INSERT INTO Employee VALUES(2, 'Priyanka', 'Female', 1);
INSERT INTO Employee VALUES(3, 'Preety', 'Female',
NULL);
INSERT INTO Employee VALUES(4, 'Anurag', 'Male', 1);
INSERT INTO Employee VALUES(5, 'Sambit', 'Male', 1);
INSERT INTO Employee VALUES(6, 'Rajesh', 'Male', 3);
INSERT INTO Employee VALUES(7, 'Hina', 'Female', 3);
```

Inner Self Join

Query:

```
SELECT E.FullName as Employee, M.FullName as Manager
FROM Employee E
INNER JOIN Employee M
ON E.ManagerId = M.EmployeeId;
EMPLOYEE        MANAGER
Priyanka          Pranaya
Anurag          Pranaya
```

Query:

```
Sambit          Pranaya
Pranaya          Preety
Rajesh          Preety
Hina                Preety
```

5.2.4 Cartesian Join

- Returns the Cartesian product of two tables.
- It combines every row from the first table with every row from the second table, resulting in all possible combinations.
- Cross joins are also known as Cartesian joins or cross products.

Query:

```
CREATE TABLE employee(id number, first_name varchar2(20), second_name varchar2(20));
INSERT INTO employee VALUES(101, 'Ravi', 'Sharma');
INSERT INTO employee VALUES(102, 'Kavi', 'Verma');
INSERT INTO employee VALUES(103, 'Jimmy', 'Kumar');
INSERT INTO employee VALUES(104, 'Josh', 'Kumar');

CREATE TABLE department(id number, name varchar2(20));
INSERT INTO department VALUES(101, 'HR');
INSERT INTO department VALUES(102, 'IT');
```

Query:

```
SELECT employee.first_name, department.name
FROM employee
CROSS JOIN department;
```

Query:

```
_____________________________-

FIRST_NAME          NAME
Ravi            HR
Kavi            HR
Jimmy             HR
Josh            HR
Ravi            IT
Kavi            IT
Jimmy             IT
Josh            IT
```

The best view comes after the hardest climb!

6. Subquery

6.0.1 What is subquery?

- Subquery is also called as **nested query** or **inner query**.
- **WHERE or HAVING** clause of one query may contain another query called **subquery**.
- Oracle processes subqueries before the main query.
- A subquery can return single/multiple values.
- Types of subquery:
  - Single value subquery
  - Multi-value subquery
  - Multi-table subquery
  - Multi-column subquery
  - Correlated subquery

6.0.2 Single value/row subquery

- Subquery that returns a single value(1 row of 1 column only).
- Single value subquery is also called scalar subquery or single row subquery.

Query:

```
CREATE TABLE employee(
    id NUMBER,
    name VARCHAR2(20),
    salary NUMBER
);
INSERT INTO employee VALUES(101, 'Ram',34000);
INSERT INTO employee VALUES(102, 'Ravi',56000);
INSERT INTO employee VALUES(103, 'Kavi',16000);
INSERT INTO employee VALUES(104, 'Tanvi',34000);
INSERT INTO employee VALUES(105, 'Juhi',34000);
```

Query:

```
SELECT * FROM employee
WHERE salary = (
    SELECT salary
    FROM employee
    WHERE name = 'Ram'
);

ID        NAME        SALARY
-------------------------------
101       Ram         34000
104       Tanvi       34000
105       Juhi        34000
```

Query:

```
SELECT * FROM employee WHERE
salary = (
    SELECT max(salary)
    FROM employee
);
ID        NAME        SALARY
_____________________________

102        Ravi        56000

DELETE from employee WHERE salary = (
SELECT MAX(salary) FROM employee)

SELECT * FROM (
    SELECT * FROM employee
    WHERE salary > 16000)
WHERE name = 'Ram' OR name = 'Kavi';

ID        NAME        SALARY
_____________________________-

101        Ram        34000
```

6.0.3 Multi value/row subquery

- Subquery that returns any number of values (Multiple rows of single column only).
- Use operator that supports multiple values, such as the **IN, ANY, or ALL** operators.

Query:

```
create table employee(
    id number,
    name varchar2(20),
    salary number
);
insert into employee values(101, 'Ram',34000);
insert into employee values(102, 'Ravi',56000);
insert into employee values(103, 'Kavi',16000);
insert into employee values(104, 'Tanvi',34000);
insert into employee values(105, 'Juhi',34000);
```

Query:

```
select * from employee
where salary IN (
    select salary from employee
    where salary > 16000)
order by 3;
```

ID	NAME	SALARY
101	Ram	34000
104	Tanvi	34000
105	Juhi	34000
102	Ravi	56000

6.0.4 Multi-column subquery

- Subquery that returns values of multiple columns (1/More rows of multiple columns).
- Consider below table:

Query:

```
CREATE TABLE EMPLOYEE(
    EMPNO NUMBER(4),
    ENAME VARCHAR2(10),
    JOB VARCHAR2(9),
    MGR NUMBER(4),
    HDATE DATE
);
INSERT INTO EMPLOYEE VALUES (7369, 'SMITH',
'CLERK', 7902, TO_DATE('17-DEC-1980', 'DD-MON-
YYYY'));

INSERT INTO EMPLOYEE VALUES (7499, 'ALLEN',
'SALESMAN', 7698, TO_DATE('20-FEB-1981', 'DD-
MON-YYYY'));

INSERT INTO EMPLOYEE VALUES (7521, 'WARD',
'SALESMAN', 7698, TO_DATE('22-FEB-1981', 'DD-
MON-YYYY'));
```

- Consider below query:

Query:

```
SELECT * FROM EMPLOYEE
WHERE(JOB, MGR) IN(
    SELECT JOB, MGR FROM
    EMPLOYEE WHERE ENAME='WARD');
```

Query:

EMPNO	ENAME	JOB	MGR	HDATE
7521	SMITH	SALESMAN	7698	22-FEB-81
7499	ALLEN	SALESMAN	7698	20-FEB-81
7521	WARD	SALESMAN	7698	22-FEB-81

6.0.5 Correlated subquery

- A correlated subquery is a subquery which is executed repeatedly once for each row of the main query.
- A correlated subquery is a nested query which is executed once for each 'candidate row' selected by the main.

Query:

```
CREATE TABLE employee(
deptid NUMBER,
name VARCHAR2(20),
salary NUMBER
);
INSERT INTO employee VALUES(101, 'Ram',34000);
INSERT INTO employee VALUES(102, 'Ravi',56000);
INSERT INTO employee VALUES(103, 'Kavi',16000);
INSERT INTO employee VALUES(104, 'Tanvi',34000);
INSERT INTO employee VALUES(105, 'Juhi',34000);
INSERT INTO employee VALUES(106, 'Josh',156000);
INSERT INTO employee VALUES(101, 'Jim',52000);
INSERT INTO employee VALUES(104, 'Jack',89000);
INSERT INTO employee VALUES(102, 'Anny',38000);
```

Eg: Below query display all employees who earn salaries greater than the average salary within their own department:

Query:

```
SELECT  name
FROM  employee  e
WHERE  salary  >  (
SELECT  AVG(salary)  FROM  employee
WHERE  deptid  =  e.deptid
);

NAME          SALARY
_____________________

Ravi           56000
Jim           52000
Jack           89000
```