**Database notes**

Syllabus

1. Introduction to dbms
   1. Purpose of database system
   2. Dbms vs file system
   3. Instance and schemas
   4. Data model
      1. Entity Relationship model
      2. Relational model
   5. DDL,DML
   6. Database Administrator and Database User
   7. Entity sets ,Relationship set, Attributes, types of attributes, domain, Mapping Cardinalities, concept of database keys
2. Relational model
   1. Structure of relational model
   2. Database schema
   3. Query language
   4. Relational algebra
      1. Fundamental operations
3. Sql
   1. Introduction to sql queries
   2. Creating, inserting, updating, deleting tables
   3. Integrity constraints-primary key, foreign key, null constraints
   4. Arithmetic, logical ,Relational operators
   5. Aggregate functions, Mathematical functions, Date functions, string functions
   6. Sub queries, concept of join
   7. View- need, creating, updating and deleting database view
   8. Concept of index
4. Pl/sql
   1. Pl/sql block structure
   2. Variables
   3. Pl/sql control structure
   4. Cursors-types, attributes
   5. Triggers-use of database trigger
   6. Stored procedure and functions-advantages , syntax for creating
   7. Exception handling in ol/sql
5. Normalization
   1. Purpose of normalization
   2. Functional dependencies and decomposition
   3. Normalization using 1NF,2NF,3NF,BCNF
6. Query processing and Transaction processing
   1. Overview of query processing
      1. Parsing and translation
      2. Optimization
      3. Evaluation
   2. Measures of query cost
   3. Concept of transaction
   4. Transaction states
   5. Concurrent executions
   6. Serializability
   7. Recoverability
7. Concurrency control
   1. Lock based protocols
      1. Locks-shared mode and exclusive mode lock
      2. Granting of locks
   2. Two phase locking protocol
   3. Time stamp based protocol
   4. Validation based protocol
8. Recovery
   1. Failure classification
   2. Data access
   3. Log based Recovery
      1. Deferred database modification
      2. Immediate database modification
      3. checkpoints

What is Database

The database is a collection of inter-related data which is used to retrieve, insert and delete the data efficiently. It is also used to organize the data in the form of a table, schema, views, and reports, etc.

**For example:** The college Database organizes the data about the admin, staff, students and faculty etc.

Using the database, you can easily retrieve, insert, and delete the information.

Database Management System

* Database management system is a software which is used to manage the database. For example: MySQL, Oracle, etc are a very popular commercial database which is used in different applications.
* DBMS provides an interface to perform various operations like database creation, storing data in it, updating data, creating a table in the database and a lot more.
* It provides protection and security to the database. In the case of multiple users, it also maintains data consistency.

Advantages of DBMS

* **Controls database redundancy:** It can control data redundancy because it stores all the data in one single database file and that recorded data is placed in the database.
* **Data sharing:** In DBMS, the authorized users of an organization can share the data among multiple users.
* **Easily Maintenance:** It can be easily maintainable due to the centralized nature of the database system.
* **Reduce time:** It reduces development time and maintenance need.
* **Backup:** It provides backup and recovery subsystems which create automatic backup of data from hardware and software failures and restores the data if required.
* **multiple user interface:** It provides different types of user interfaces like graphical user interfaces, application program interfaces

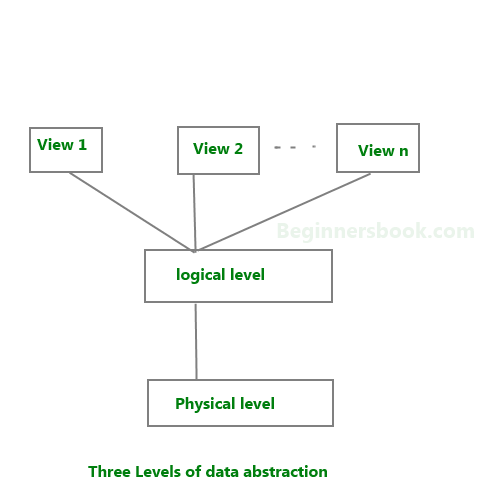
Disadvantages of DBMS

* **Cost of Hardware and Software:** It requires a high speed of data processor and large memory size to run DBMS software.
* **Size:** It occupies a large space of disks and large memory to run them efficiently.
* **Complexity:** Database system creates additional complexity and requirements.
* **Higher impact of failure:** Failure is highly impacted the database because in most of the organization, all the data stored in a single database and if the database is damaged due to electric failure or database corruption then the data may be lost forever.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DBMS vs. File System There are following differences between DBMS and File system:   |  |  | | --- | --- | | **DBMS** | **File System** | | DBMS is a collection of data. In DBMS, the user is not required to write the procedures. | File system is a collection of data. In this system, the user has to write the procedures for managing the database. | | DBMS gives an abstract view of data that hides the details. | File system provides the detail of the data representation and storage of data. | | DBMS provides a crash recovery mechanism, i.e., DBMS protects the user from the system failure. | File system doesn't have a crash mechanism, i.e., if the system crashes while entering some data, then the content of the file will lost. | | DBMS provides a good protection mechanism. | It is very difficult to protect a file under the file system. | | DBMS contains a wide variety of sophisticated techniques to store and retrieve the data. | File system can't efficiently store and retrieve the data. | | DBMS takes care of Concurrent access of data using some form of locking. | In the File system, concurrent access has many problems like redirecting the file while other deleting some information or updating some information. | |

# Data Abstraction in DBMS

Database systems are made-up of complex data structures. To ease the user interaction with database, the developers hide internal irrelevant details from users. This process of hiding irrelevant details from user is called data abstraction.



**We have three levels of abstraction**:  
**Physical level**: This is the lowest level of data abstraction. It describes how data is actually stored in database. You can get the complex data structure details at this level.

**Logical level**: This is the middle level of 3-level data abstraction architecture. It describes what data is stored in database.

**View level**: Highest level of data abstraction. This level describes the user interaction with database system.

**Example**: Let’s say we are storing customer information in a customer table. At **physical level** these records can be described as blocks of storage (bytes, gigabytes, terabytes etc.) in memory. These details are often hidden from the programmers.

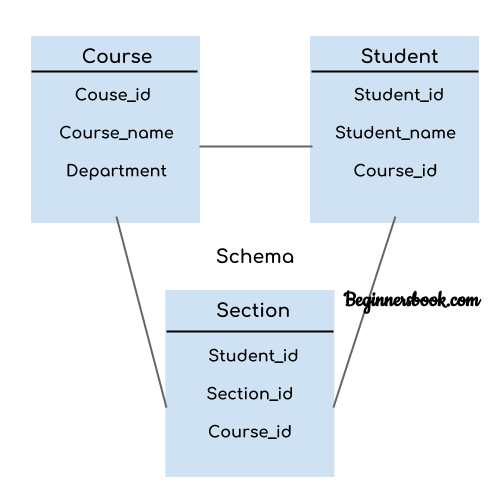
At the **logical level** these records can be described as fields and attributes along with their data types, their relationship among each other can be logically implemented. The programmers generally work at this level because they are aware of such things about database systems.

At **view level**, user just interact with system with the help of GUI and enter the details at the screen, they are not aware of how the data is stored and what data is stored; such details are hidden from them.

# Instance and schema in DBMS

## DBMS Schema

**Definition of schema**: Design of a database is called the schema. Schema is of three types: Physical schema, logical schema and view schema.

For example: In the following diagram, we have a schema that shows the relationship between three tables: Course, Student and Section. The diagram only shows the design of the database, it doesn’t show the data present in those tables. Schema is only a structural view(design) of a database as shown in the diagram below.  


* 1. The design of a database at physical level is called **physical schema**, how the data stored in blocks of storage is described at this level.
  2. Design of database at logical level is called **logical schema**, programmers and database administrators work at this level, at this level data can be described as certain types of data records gets stored in data structures, however the internal details such as implementation of data structure is hidden at this level (available at physical level).
  3. Design of database at view level is called **view schema**. This generally describes end user interaction with database systems.

.

## DBMS Instance

**Definition of instance**: The data stored in database at a particular moment of time is called instance of database. Database schema defines the variable declarations in tables that belong to a particular database; the value of these variables at a moment of time is called the instance of that database.

For example, lets say we have a single table student in the database, today the table has 100 records, so today the instance of the database has 100 records. Lets say we are going to add another 100 records in this table by tomorrow so the instance of database tomorrow will have 200 records in table. In short, at a particular moment the data stored in database is called the instance, that changes over time when we add or delete data from the database.

# DBMS Database Models

A Database model defines the logical design and structure of a database and defines how data will be stored, accessed and updated in a database management system. While the **Relational Model** is the most widely used database model, there are other models too:

* Hierarchical Model
* Network Model
* Entity-relationship Model
* Relational Model

## Entity-relationship Model

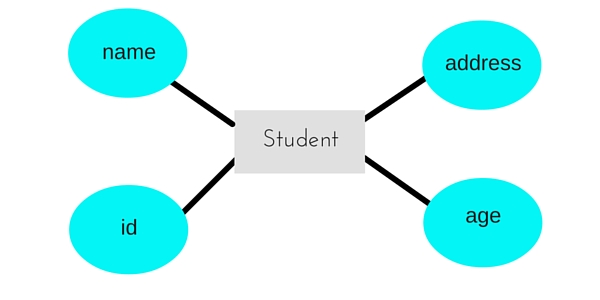
In this database model, relationships are created by dividing object of interest into entity and its characteristics into attributes.

Different entities are related using relationships.

E-R Models are defined to represent the relationships into pictorial form to make it easier for different stakeholders to understand.

This model is good to design a database, which can then be turned into tables in relational model(explained below).

Let's take an example, If we have to design a School Database, then **Student** will be an **entity** with **attributes** name, age, address etc. As **Address** is generally complex, it can be another **entity** with **attributes** street name, pincode, city etc, and there will be a relationship between them.



## Relational Model

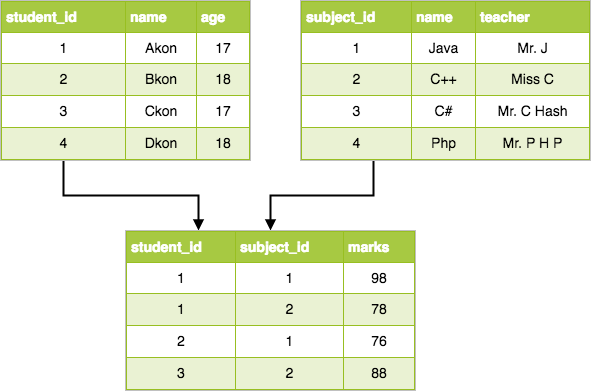
In this model, data is organised in two-dimensional **tables** and the relationship is maintained by storing a common field.

This model was introduced by E.F Codd in 1970, and since then it has been the most widely used database model, infact, we can say the only database model used around the world.

The basic structure of data in the relational model is tables. All the information related to a particular type is stored in rows of that table.

Hence, tables are also known as **relations** in relational model.

In the coming tutorials we will learn how to design tables, normalize them to reduce data redundancy and how to use Structured Query language to access data from tables.



# Basic Concepts of ER Model in DBMS

As we described in the tutorial Database models, Entity-relationship model is a model used for design and representation of relationships between data.

The main data objects are termed as Entities, with their details defined as attributes, some of these attributes are important and are used to identity the entity, and different entities are related using relationships.

In short, to understand about the ER Model, we must understand about:

* Entity and Entity Set
* What are Attributes? And Types of Attributes.
* Keys
* Relationships

Let's take an example to explain everything. For a **School Management Software**, we will have to store **Student** information, **Teacher** information, **Classes**, **Subjects** taught in each class etc.

## ER Model: Entity and Entity Set

Considering the above example, **Student** is an entity, **Teacher** is an entity, similarly, **Class**, **Subject**etc are also entities.

An Entity is generally a real-world object which has characteristics and holds relationships in a DBMS.

If a Student is an Entity, then the complete dataset of all the students will be the **Entity Set**

## ER Model: Attributes

If a Student is an Entity, then student's **roll no.**, student's **name**, student's **age**, student's **gender** etc will be its attributes.

An attribute can be of many types, here are different types of attributes defined in ER database model:

1. **Simple attribute:** The attributes with values that are atomic and cannot be broken down further are simple attributes. For example, student's **age**.
2. **Composite attribute:** A composite attribute is made up of more than one simple attribute. For example, student's **address** will contain, **house no.**, **street name**, **pincode** etc.
3. **Derived attribute:** These are the attributes which are not present in the whole database management system, but are derived using other attributes. For example, *average age of students in a class*.
4. **Single-valued attribute:** As the name suggests, they have a single value.
5. **Multi-valued attribute:** And, they can have multiple values.

## ER Model: Keys

If the attribute **roll no.** can uniquely identify a student entity, amongst all the students, then the attribute **roll no.** will be said to be a key.

Following are the types of Keys:

1. Super Key
2. Candidate Key
3. Primary Key

## ER Model: Relationships

When an Entity is related to another Entity, they are said to have a relationship. For example, A **Class**Entity is related to **Student** entity, becasue students study in classes, hence this is a relationship.

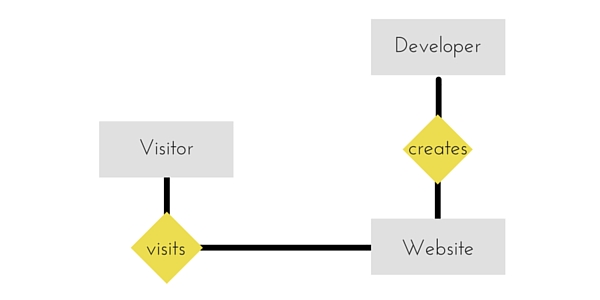
Depending upon the number of entities involved, a **degree** is assigned to relationships.

For example, if 2 entities are involved, it is said to be **Binary relationship**, if 3 entities are involved, it is said to be **Ternary** relationship, and so on.

# Working with ER Diagrams

ER Diagram is a visual representation of data that describes how data is related to each other. In ER Model, we disintegrate data into entities, attributes and setup relationships between entities, all this can be represented visually using the ER diagram.

For example, in the below diagram, anyone can see and understand what the diagram wants to convey: *Developer develops a website, whereas a Visitor visits a website*.



## Components of ER Diagram

Entitiy, Attributes, Relationships etc form the components of ER Diagram and there are defined symbols and shapes to represent each one of them.

Let's see how we can represent these in our ER Diagram.

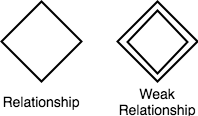
#### Entity

Simple rectangular box represents an Entity.

Entity in ER diagram

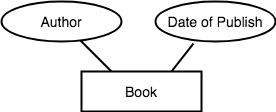
#### Relationships between Entities - Weak and Strong

Rhombus is used to setup relationships between two or more entities.



#### Attributes for any Entity

Ellipse is used to represent attributes of any entity. It is connected to the entity.



#### Weak Entity

A weak Entity is represented using double rectangular boxes. It is generally connected to another entity.

Weak Entity in ER diagram

#### Key Attribute for any Entity

To represent a Key attribute, the attribute name inside the Ellipse is underlined.

Key Attribute in ER diagram

#### Derived Attribute for any Entity

Derived attributes are those which are derived based on other attributes, for example, age can be derived from date of birth.

To represent a derived attribute, another dotted ellipse is created inside the main ellipse.



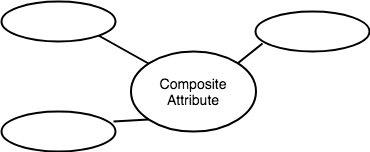
#### Multivalued Attribute for any Entity

Double Ellipse, one inside another, represents the attribute which can have multiple values.



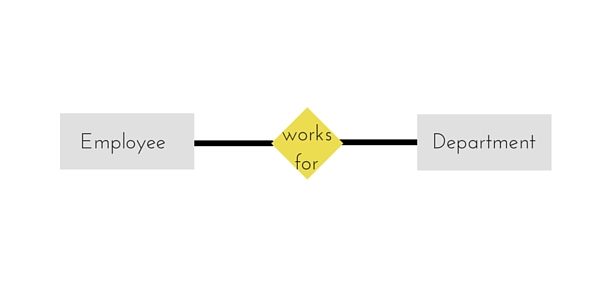
#### Composite Attribute for any Entity

A composite attribute is the attribute, which also has attributes.



### ER Diagram: Entity

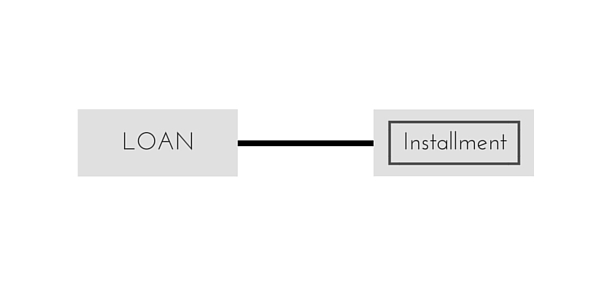
An **Entity** can be any object, place, person or class. In ER Diagram, an **entity** is represented using rectangles. Consider an example of an Organisation- Employee, Manager, Department, Product and many more can be taken as entities in an Organisation.



The yellow rhombus in between represents a relationship.

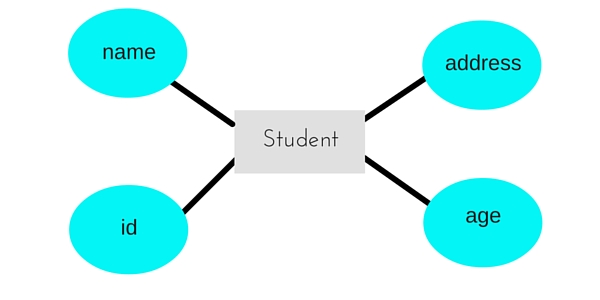
### ER Diagram: Weak Entity

Weak entity is an entity that depends on another entity. Weak entity doesn't have any key attribute of its own. Double rectangle is used to represent a weak entity.



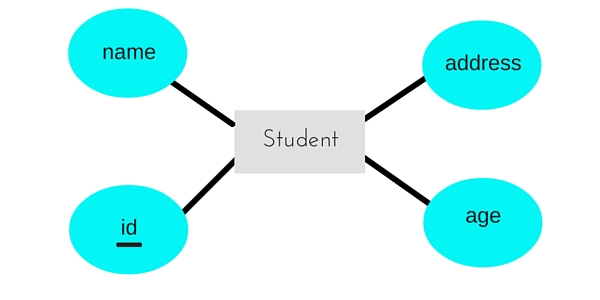
### ER Diagram: Attribute

An **Attribute** describes a property or characterstic of an entity. For example, **Name**, **Age**, **Address** etc can be attributes of a **Student**. An attribute is represented using eclipse.



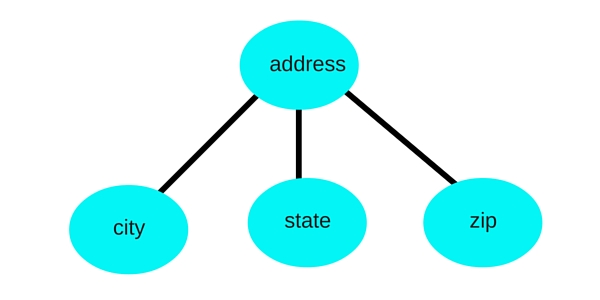
### ER Diagram: Key Attribute

Key attribute represents the main characterstic of an Entity. It is used to represent a Primary key. Ellipse with the text underlined, represents Key Attribute.



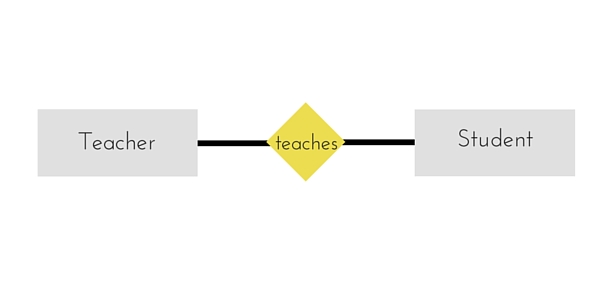
### ER Diagram: Composite Attribute

An attribute can also have their own attributes. These attributes are known as **Composite** attributes.



### ER Diagram: Relationship

A Relationship describes relation between **entities**. Relationship is represented using diamonds or rhombus.



There are three types of relationship that exist between Entities.

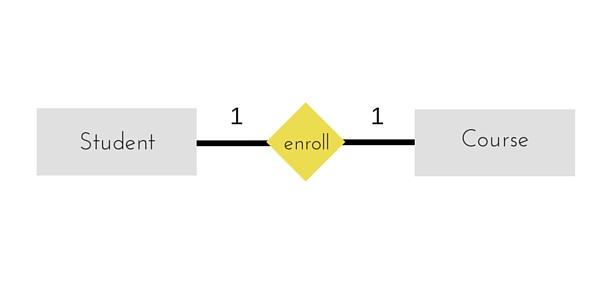
1. Binary Relationship
2. Recursive Relationship
3. Ternary Relationship

### ER Diagram: Binary Relationship

Binary Relationship means relation between two Entities. This is further divided into three types.

#### One to One Relationship

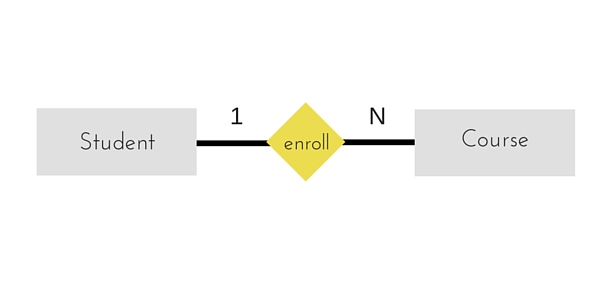
This type of relationship is rarely seen in real world.



The above example describes that one student can enroll only for one course and a course will also have only one Student. This is not what you will usually see in real-world relationships

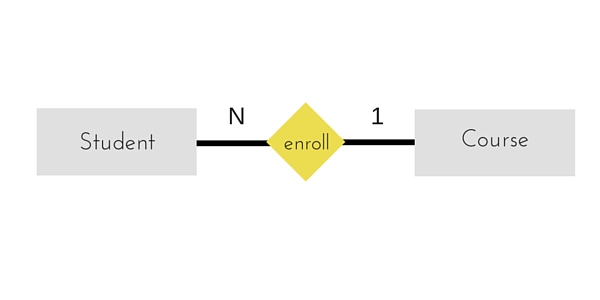
#### One to Many Relationship

The below example showcases this relationship, which means that 1 student can opt for many courses, but a course can only have 1 student. Sounds weird! This is how it is.

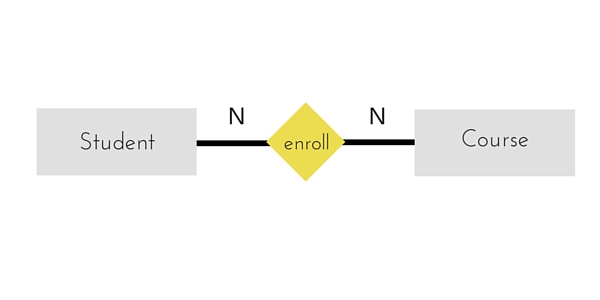


#### Many to One Relationship

It reflects business rule that many entities can be associated with just one entity. For example, Student enrolls for only one Course but a Course can have many Students.



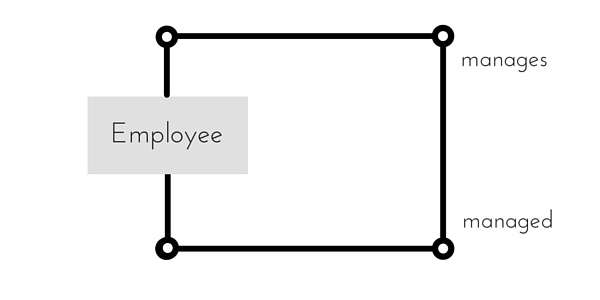
#### Many to Many Relationship



The above diagram represents that one student can enroll for more than one courses. And a course can have more than 1 student enrolled in it.

### ER Diagram: Recursive Relationship

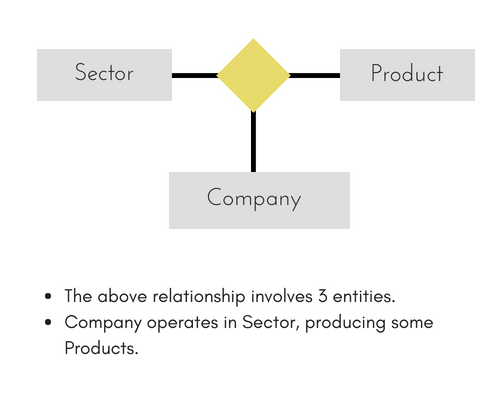
When an Entity is related with itself it is known as **Recursive** Relationship.



### ER Diagram: Ternary Relationship

Relationship of degree three is called Ternary relationship.

A Ternary relationship involves three entities. In such relationships we always consider two entites together and then look upon the third.



For example, in the diagram above, we have three related entities, **Company**, **Product** and **Sector**. To understand the relationship better or to define rules around the model, we should relate two entities and then derive the third one.

A **Company** produces many **Products**/ each product is produced by exactly one company.

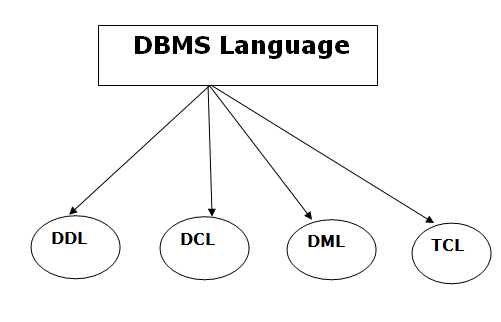
A **Company** operates in only one **Sector** / each sector has many companies operating in it.

Considering the above two rules or relationships, we see that although the complete relationship involves three entities, but we are looking at two entities at a time.

# Database Language

* A DBMS has appropriate languages and interfaces to express database queries and updates.
* Database languages can be used to read, store and update the data in the database.

## Types of Database Language



### 1. Data Definition Language

* **DDL** stands for **D**ata **D**efinition **L**anguage. It is used to define database structure or pattern.
* It is used to create schema, tables, indexes, constraints, etc. in the database.
* Using the DDL statements, you can create the skeleton of the database.
* Data definition language is used to store the information of metadata like the number of tables and schemas, their names, indexes, columns in each table, constraints, etc.

Here are some tasks that come under DDL:

* **Create:** It is used to create objects in the database.
* **Alter:** It is used to alter the structure of the database.
* **Drop:** It is used to delete objects from the database.
* **Truncate:** It is used to remove all records from a table.
* **Rename:** It is used to rename an object.

These commands are used to update the database schema that's why they come under Data definition language.

### 2. Data Manipulation Language

**DML** stands for **D**ata **M**anipulation **L**anguage. It is used for accessing and manipulating data in a database. It handles user requests.

Here are some tasks that come under DML:

* **Select:** It is used to retrieve data from a database.
* **Insert:** It is used to insert data into a table.
* **Update:** It is used to update existing data within a table.
* **Delete:** It is used to delete all records from a table.

### 3. Data Control Language

* **DCL** stands for **D**ata **C**ontrol **L**anguage. It is used to retrieve the stored or saved data.
* The DCL execution is transactional. It also has rollback parameters.

(But in Oracle database, the execution of data control language does not have the feature of rolling back.)

Here are some tasks that come under DCL:

* **Grant:** It is used to give user access privileges to a database.
* **Revoke:** It is used to take back permissions from the user.

There are the following operations which have the authorization of Revoke:

CONNECT, INSERT, USAGE, EXECUTE, DELETE, UPDATE and SELECT.

### 4. Transaction Control Languages

TCL is used to run the changes made by the DML statement. TCL can be grouped into a logical transaction.

Here are some tasks that come under TCL:

* **Commit:** It is used to save the transaction on the database.
* **Rollback:** It is used to restore the database to original since the last Commit.

1. **Database Users and User Interfaces**  
   There are four different types of database-system users, differentiated by the way  
   they expect to interact with the system. Different types of user interfaces have  
   been designed for the different types of users.

• **Naive users** are unsophisticated users who interact with the system by invoking  
one of the application programs that have been written previously.  
For example, a clerk in the university who needs to add a new instructor to  
department A invokes a program called new hire. This program asks the clerk  
for the name of the new instructor, her new ID, the name of the department  
(that is, A), and the salary.

    The typical user interface for naive users is a forms interface, where the  
user can fill in appropriate fields of the form. Naive users may also simply  
read reports generated from the database.

   As another example, consider a student, who during class registration  
period, wishes to register for a class by using a Web interface. Such a user  
connects to a Web application program that runs at a Web server. The application  
first verifies the identity of the user, and allows her to access a form  
where she enters the desired information. The form information is sent back  
to the Web application at the server, which then determines if there is room  
in the class (by retrieving information from the database) and if so adds the  
student information to the class roster in the database.

• **Application programmers** are computer professionals who write application  
programs. Application programmers can choose from many tools to develop  
user interfaces. **Rapid application development (RAD)** tools are tools that enable  
an application programmer to construct forms and reports with minimal  
programming effort.

• **Sophisticated users**interact with the system without writing programs. Instead,  
they form their requests either using a database query language or by  
using tools such as data analysis software. Analysts who submit queries to  
explore data in the database fall in this category.

• **Specialized users** are sophisticated users who write specialized database  
applications that do not fit into the traditional data-processing framework.  
Among these applications are computer-aided design systems, knowledgebase  
and expert systems, systems that store data with complex data types (for  
example, graphics data and audio data), and environment-modeling systems.  
  
**2.  Database Administrator**

One of the main reasons for using DBMSs is to have central control of both the data  
and the programs that access those data. A person who has such central control  
over the system is called a **database administrator (DBA**). The functions of a DBA  
include:

**• Schema definition.** The DBA creates the original database schema by executing  
a set of data definition statements in the DDL.

**• Storage structure and access-method definition.**

**• Schema and physical-organization modification.** The DBA carries out changes  
to the schema and physical organization to reflect the changing needs of the  
organization, or to alter the physical organization to improve performance.  
  
**• Granting of authorization for data access.** By granting different types of  
authorization, the database administrator can regulate which parts of the  
database various users can access. The authorization information is kept in a  
special system structure that the database system consults whenever someone  
attempts to access the data in the system.

**• Routine maintenance.** Examples of the database administrator’s routine  
maintenance activities are:  
 Periodically backing up the database, either onto tapes or onto remote  
servers, to prevent loss of data in case of disasters such as flooding.

Ensuring that enough free disk space is available for normal operations,  
and upgrading disk space as required.

Monitoring jobs running on the database and ensuring that performance  
is not degraded by very expensive tasks submitted by some users.

**Domain:**

A domain is defined as the set of all unique values permitted for an attribute. For example, a domain of date is the set of all possible valid dates, a domain of integer is all possible whole numbers, a domain of day-of-week is Monday, Tuesday ... Sunday.

This in effect is defining rules for a particular attribute. If it is determined that an attribute is a date then it should be implemented in the database to prevent invalid dates being entered.

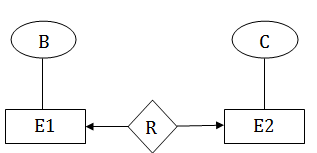
**A classic example of this is where the data from a legacy system is loaded into a newly designed database.** The new system is well designed. Columns that hold dates are defined as such whereas, in the old system, they were held as character strings.

# Mapping Cardinality

* A mapping constraint is a data constraint that expresses the number of entities to which another entity can be related via a relationship set.
* It is most useful in describing the relationship sets that involve more than two entity sets.
* For binary relationship set R on an entity set A and B, there are four possible mapping cardinalities. These are as follows:
  1. One to one (1:1)
  2. One to many (1:M)
  3. Many to one (M:1)
  4. Many to many (M:M)

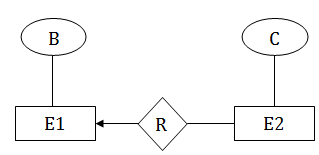
## One-to-one

In one-to-one mapping, an entity in E1 is associated with at most one entity in E2, and an entity in E2 is associated with at most one entity in E1.



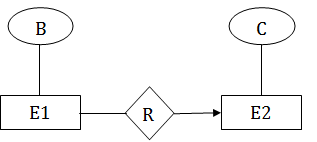
## One-to-many

In one-to-many mapping, an entity in E1 is associated with any number of entities in E2, and an entity in E2 is associated with at most one entity in E1.



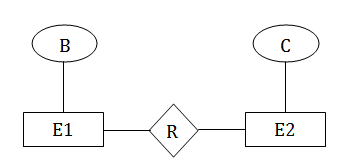
## Many-to-one

In one-to-many mapping, an entity in E1 is associated with at most one entity in E2, and an entity in E2 is associated with any number of entities in E1.



## Many-to-many

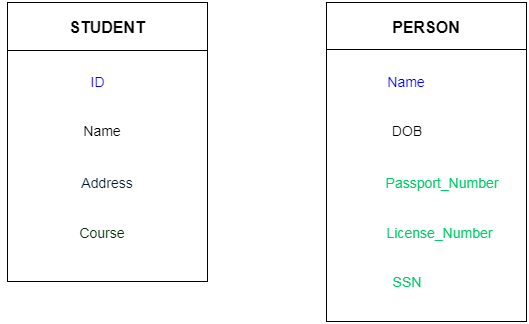
In many-to-many mapping, an entity in E1 is associated with any number of entities in E2, and an entity in E2 is associated with any number of entities in E1.



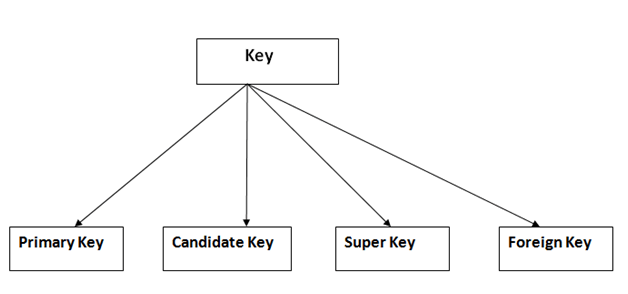
# Keys

* Keys play an important role in the relational database.
* It is used to uniquely identify any record or row of data from the table. It is also used to establish and identify relationships between tables.

**For example:** In Student table, ID is used as a key because it is unique for each student. In PERSON table, passport\_number, license\_number, SSN are keys since they are unique for each person.

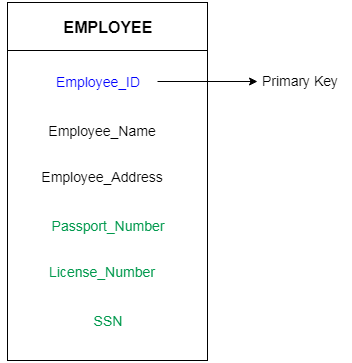


## Types of key:



### 1. Primary key

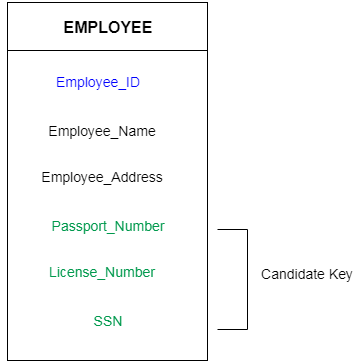
* It is the first key which is used to identify one and only one instance of an entity uniquely. An entity can contain multiple keys as we saw in PERSON table. The key which is most suitable from those lists become a primary key.
* In the EMPLOYEE table, ID can be primary key since it is unique for each employee. In the EMPLOYEE table, we can even select License\_Number and Passport\_Number as primary key since they are also unique.
* For each entity, selection of the primary key is based on requirement and developers.



### 2. Candidate key

* A candidate key is an attribute or set of an attribute which can uniquely identify a tuple.
* The remaining attributes except for primary key are considered as a candidate key. The candidate keys are as strong as the primary key.

**For example:** In the EMPLOYEE table, id is best suited for the primary key. Rest of the attributes like SSN, Passport\_Number, and License\_Number, etc. are considered as a candidate key.



### 3. Super Key

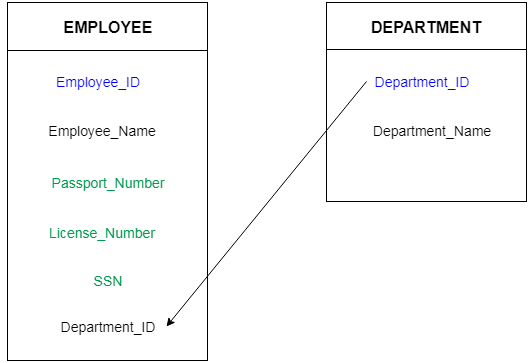
Super key is a set of an attribute which can uniquely identify a tuple. Super key is a superset of a candidate key.

**For example:** In the above EMPLOYEE table, for(EMPLOEE\_ID, EMPLOYEE\_NAME) the name of two employees can be the same, but their EMPLYEE\_ID can't be the same. Hence, this combination can also be a key.

The super key would be EMPLOYEE-ID, (EMPLOYEE\_ID, EMPLOYEE-NAME), etc.

### 4. Foreign key

* Foreign keys are the column of the table which is used to point to the primary key of another table.
* In a company, every employee works in a specific department, and employee and department are two different entities. So we can't store the information of the department in the employee table. That's why we link these two tables through the primary key of one table.
* We add the primary key of the DEPARTMENT table, Department\_Id as a new attribute in the EMPLOYEE table.
* Now in the EMPLOYEE table, Department\_Id is the foreign key, and both the tables are related.



**3 chapter**

# SQL Syntax

SQL follows some unique set of rules and guidelines called syntax. Here, we are providing all the basic SQL syntax.

* **SQL** is not case sensitive. Generally SQL keywords are written in uppercase.
* SQL statements are dependent on text lines. We can place a single SQL statement on one or multiple text lines.
* You can perform most of the action in a database with SQL statements.
* SQL depends on relational algebra and tuple relational calculus.

## SQL statement

SQL statements are started with any of the SQL commands/keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP etc. and the statement ends with a semicolon (;).

**SELECT** "column\_name" **FROM** "table\_name";

## SQL Commands

These are the some important SQL command:

**SELECT**: it extracts data from a database.

**UPDATE**: it updates data in database.

**DELETE**: it deletes data from database.

**CREATE TABLE**: it creates a new table.

**ALTER TABLE**: it is used to modify the table.

**DROP TABLE**: it deletes a table.

**INSERT INTO**: it inserts new data into a database.

# SQL Data Types

Data types are used to represent the nature of the data that can be stored in the database table. For example, in a particular column of a table, if we want to store a string type of data then we will have to declare a string data type of this column.

Data types mainly classified into three categories for every database.

* String Data types
* Numeric Data types
* Date and time Data types

### Oracle Data Types

**Oracle String data types**

|  |  |
| --- | --- |
| **CHAR(size)** | It is used to store character data within the predefined length. It can be stored up to 2000 bytes. |
| **NCHAR(size)** | It is used to store national character data within the predefined length. It can be stored up to 2000 bytes. |
| **VARCHAR2(size)** | It is used to store variable string data within the predefined length. It can be stored up to 4000 byte. |
| **VARCHAR(SIZE)** | It is the same as VARCHAR2(size). You can also use VARCHAR(size), but it is suggested to use VARCHAR2(size) |
| **NVARCHAR2(size)** | It is used to store Unicode string data within the predefined length. We have to must specify the size of NVARCHAR2 data type. It can be stored up to 4000 bytes. |

**Oracle Numeric Data Types**

|  |  |
| --- | --- |
| **NUMBER(p, s)** | It contains precision p and scale s. The precision p can range from 1 to 38, and the scale s can range from -84 to 127. |
| **FLOAT(p)** | It is a subtype of the NUMBER data type. The precision p can range from 1 to 126. |
| **BINARY\_FLOAT** | It is used for binary precision( 32-bit). It requires 5 bytes, including length byte. |
| **BINARY\_DOUBLE** | It is used for double binary precision (64-bit). It requires 9 bytes, including length byte. |

**Oracle Date and Time Data Types**

|  |  |
| --- | --- |
| **DATE** | It is used to store a valid date-time format with a fixed length. Its range varies from January 1, 4712 BC to December 31, 9999 AD. |
| **TIMESTAMP** | It is used to store the valid date in YYYY-MM-DD with time hh:mm:ss format. |

**Oracle Large Object Data Types (LOB Types)**

|  |  |
| --- | --- |
| **BLOB** | It is used to specify unstructured binary data. Its range goes up to 232-1 bytes or 4 GB. |
| **BFILE** | It is used to store binary data in an external file. Its range goes up to 232-1 bytes or 4 GB. |
| **CLOB** | It is used for single-byte character data. Its range goes up to 232-1 bytes or 4 GB. |
| **NCLOB** | It is used to specify single byte or fixed length multibyte national character set (NCHAR) data. Its range is up to 232-1 bytes or 4 GB. |
| **RAW(size)** | It is used to specify variable length raw binary data. Its range is up to 2000 bytes per row. Its maximum size must be specified. |
| **LONG RAW** | It is used to specify variable length raw binary data. Its range up to 231-1 bytes or 2 GB, per row. |

# SQL Operators

SQL statements generally contain some reserved words or characters that are used to perform operations such as comparison and arithmetical operations etc. These reserved words or characters are known as operators.

Generally there are three types of operators in SQL:

1. SQL Arithmetic Operators
2. SQL Relational Operators
3. SQL Logical Operators

## SQL Arithmetic Operators:

Let's assume two variables "a" and "b". Here "a" is valued 50 and "b" valued 100.

**Example:**

|  |  |  |
| --- | --- | --- |
| **Operators** | **Descriptions** | **Examples** |
| + | It is used to add containing values of both operands | a+b will give 150 |
| - | It subtracts right hand operand from left hand operand | a-b will give -50 |
| \* | It multiply both operand's values | a\*b will give 5000 |
| / | It divides left hand operand by right hand operand | b/a will give 2 |
| % | It divides left hand operand by right hand operand and returns reminder | b%a will give 0 |

## SQL Relational Operators:

Let's take two variables "a" and "b" that are valued 50 and 100.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Examine both operands value that are equal or not,if yes condition become true. | (a=b) is not true |
| != | This is used to check the value of both operands equal or not,if not condition become true. | (a!=b) is true |
| < > | Examines the operand's value equal or not, if values are not equal condition is true | (a<>b) is true |
| > | Examine the left operand value is greater than right Operand, if yes condition becomes true | (a>b) is not true |
| < | Examines the left operand value is less than right Operand, if yes condition becomes true | (a<="" td=""> |
| >= | Examines that the value of left operand is greater than or equal to the value of right operand or not,if yes condition become true | (a>=b) is not true |
| <= | Examines that the value of left operand is less than or equal to the value of right operand or not, if yes condition becomes true | (a<=b) is true |
| !< | Examines that the left operand value is not less than the right operand value | (a!<="" td=""> |
| !> | Examines that the value of left operand is not greater than the value of right operand | (a!>b) is true |

## SQL Logical Operators:

This is the list of logical operators used in SQL.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| ALL | this is used to compare a value to all values in another value set. |
| AND | this operator allows the existence of multiple conditions in an SQL statement. |
| ANY | this operator is used to compare the value in list according to the condition. |
| BETWEEN | this operator is used to search for values, that are within a set of values |
| IN | this operator is used to compare a value to that specified list value |
| NOT | the NOT operator reverse the meaning of any logical operator |
| OR | this operator is used to combine multiple conditions in SQL statements |
| EXISTS | the EXISTS operator is used to search for the presence of a row in a specified table |
| LIKE | this operator is used to compare a value to similar values using wildcard operator |

# SQL CREATE TABLE

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

If you want to create a table, you should name the table and define its column and each column's data type.

Let's see the simple syntax to create the table.

**create** **table** "tablename"  ("column1" "data type",  "column2" "data type", "column3" "data type",  ...  "columnN" "data type");

Example:

**CREATE** **TABLE** Employee  ( EmployeeID number(10), FirstName varchar2(255), LastName varchar2(255),  Email varchar2(255),  AddressLine varchar2(255),City varchar2(255) );

# Oracle ALTER TABLE Statement

In Oracle, ALTER TABLE statement specifies how to add, modify, drop or delete columns in a table. It is also used to rename a table.

## How to add column in a table

**Syntax:**

**ALTER** **TABLE** table\_name  **ADD** column\_name **column**-definition;

**Example:**

**ALTER** **TABLE** customers  **ADD** customer\_age varchar2(50);

How to modify column of a table

**Syntax:**

**ALTER** **TABLE** table\_name **MODIFY** column\_name column\_type;

**Example:**

**ALTER** **TABLE** customers  **MODIFY** customer\_name varchar2(100) not null;

How to drop column of a table

**Syntax:**

**ALTER** **TABLE** table\_name **DROP** **COLUMN** column\_name;

**Example:**

**ALTER** **TABLE** customers **DROP** **COLUMN** customer\_name;

How to rename column of a table

**Syntax:**

**ALTER** **TABLE** table\_name RENAME **COLUMN** old\_name **to** new\_name;

**Example:**

**ALTER** **TABLE** customers RENAME **COLUMN** customer\_name **to** cname;

How to rename table

**Syntax:**

**ALTER** **TABLE** table\_name RENAME **TO** new\_table\_name;

**Example:**

**ALTER** **TABLE** customers RENAME **TO** retailers;

# Oracle Queries

You can execute many queries in oracle database such as insert, update, delete, alter table, drop, create and select.

## 1) Oracle Select Query

Oracle select query is used to fetch records from database. For example:

1. **SELECT** \* **from** customers;

[More Details...](https://www.javatpoint.com/oracle-select)

## 2) Oracle Insert Query

Oracle insert query is used to insert records into table. For example:

1. **insert** **into** customers **values**(101,'rahul','delhi');

## 3) Oracle Update Query

Oracle update query is used to update records of a table. For example:

1. **update** customers **set** **name**='bob', city='london' **where** id=101;

## 4) Oracle Delete Query

Oracle update query is used to delete records of a table from database. For example:

1. **delete** **from** customers **where** id=101;

## 5) Oracle Truncate Query

Oracle update query is used to truncate or remove records of a table. It doesn't remove structure. For example:

1. **truncate** **table** customers;

## 6) Oracle Drop Query

Oracle drop query is used to drop a table or view. It doesn't have structure and data. For example:

1. **drop** **table** customers;

# Oracle SELECT Statement

The Oracle SELECT statement is used to retrieve data from one or more than one tables, object tables, views, object views etc.

**Syntax**

**SELECT** expressions **FROM** tables **WHERE** conditions;

## Parameters

**1) expressions:** It specifies the columns or calculations that you want to retrieve.

**2) tables:**This parameter specifies the tables that you want to retrieve records from. There must be at least one table within the FROM clause.

**3) conditions:** It specifies the conditions that must be followed for selection.

## Select Example: select all fields

Let's take an example to select all fields from an already created table named customers

**SELECT** \* **FROM** customers;

## Select Example: select specific fields

**Example**

**SELECT** age, address, salary **FROM** customers **WHERE**  age < 25  AND salary > '20000' **ORDER** **BY** age **ASC**, salary **DESC**;

# Oracle Insert Statement

In Oracle, INSERT statement is used to add a single record or multiple records into the table.

**Syntax: (Inserting a single record using the Values keyword):**

**INSERT** **INTO** **table** (column1, column2, ... column\_n ) **VALUES** (expression1, expression2, ... expression\_n );

Parameters:

**1) table:** The table to insert the records into.

**2) column1, column2, ... column\_n:**

The columns in the table to insert values.

**3) expression1, expression2, ... expression\_n:**

The values to assign to the columns in the table. So column1 would be assigned the value of expression1, column2 would be assigned the value of expression2, and so on.

**See this example:**

**INSERT** **INTO** suppliers (supplier\_id, supplier\_name) **VALUES** (50, 'Flipkart');

# Oracle INSERT ALL statement

The Oracle INSERT ALL statement is used to insert multiple rows with a single INSERT statement. You can insert the rows into one table or multiple tables by using only one SQL command.

**Syntax**

**INSERT** ALL

**INTO** table\_name (column1, column2, column\_n) **VALUES** (expr1, expr2, expr\_n)

**INTO** table\_name(column1, column2, column\_n) **VALUES** (expr1, expr2, expr\_n)

**INTO** table\_name (column1, column2, column\_n) **VALUES** (expr1, expr2, expr\_n)

Oracle INSERT ALL Example

This example specifies how to insert multiple records in one table. Here we insert three rows into the "suppliers" table.

**INSERT** ALL

**INTO** suppliers (supplier\_id, supplier\_name) **VALUES** (20, 'Google')

**INTO** suppliers (supplier\_id, supplier\_name) **VALUES** (21, 'Microsoft')

**INTO** suppliers (supplier\_id, supplier\_name) **VALUES** (22, 'Apple')

# Oracle UPDATE Statement

In Oracle, UPDATE statement is used to update the existing records in a table. You can update a table in 2 ways.

## Traditional Update table method

**Syntax:**

**UPDATE** **table** **SET** column1 = expression1, column2 = expression2,  ...   column\_n = expression\_n  **WHERE** conditions;

Oracle Update Example: (Update single column)

**UPDATE** suppliers **SET** supplier\_name = 'Kingfisher' **WHERE** supplier\_id = 2;

# Oracle DELETE Statement

In Oracle, DELETE statement is used to remove or delete a single record or multiple records from a table.

**Syntax**

**DELETE** **FROM** table\_name  **WHERE** conditions;

## Parameters

**1) table\_name:** It specifies the table which you want to delete.

**2) conditions:** It specifies the conditions that must met for the records to be deleted.

## Oracle Delete Example: On one condition

**DELETE** **FROM** customers **WHERE** **name** = 'Sohan';

# Oracle ORDER BY Clause

In Oracle, ORDER BY Clause is used to sort or re-arrange the records in the result set. The ORDER BY clause is only used with SELECT statement.

**Syntax:**

**SELECT** expressions **FROM** tables **WHERE** conditions **ORDER** **BY** expression [ **ASC** | **DESC** ];

## Parameters:

**expressions:** It specifies columns that you want to retrieve.

**tables:** It specifies the table name from where you want to retrieve records.

**conditions:** It specifies the conditions that must be fulfilled for the records to be selected.

**ASC:** It is an optional parameter that is used to sort records in ascending order.

**DESC:** It is also an optional parameter that is used to sort records in descending order.

## Oracle ORDER BY Example: (without ASC/DESC attribute)

## **Supplier table:**

**CREATE** **TABLE** "SUPPLIER"  (    "SUPPLIER\_ID" NUMBER,   "FIRST\_NAME" VARCHAR2(4000),"LAST\_NAME" VARCHAR2(4000))

Oracle ORDER BY Example: (sorting in descending order)

If you want to sort your result in descending order, you should use the DESC attribute in your ORDER BY clause:

**Execute this Query:**

**SELECT** \* **FROM** supplier **ORDER** **BY** last\_name **desc**;

# Oracle GROUP BY Clause

In Oracle GROUP BY clause is used with SELECT statement to collect data from multiple records and group the results by one or more columns.

**Syntax:**

**SELECT** expression1, expression2, ... expression\_n, aggregate\_function (aggregate\_expression) **FROM** tables  **WHERE** conditions

**GROUP** **BY** expression1, expression2, ... expression\_n;

## Parameters:

**expression1, expression2, ... expression\_n:** It specifies the expressions that are not encapsulated within aggregate function. These expressions must be included in GROUP BY clause.

**aggregate\_function:** It specifies the aggregate functions i.e. SUM, COUNT, MIN, MAX or AVG functions.

**aggregate\_expression:** It specifies the column or expression on that the aggregate function is based on.

**tables:** It specifies the table from where you want to retrieve records.

**conditions:** It specifies the conditions that must be fulfilled for the record to be selected.

Example:

**SELECT** item, SUM(sale) **AS** "Total sales"  **FROM** salesdepartment **GROUP** **BY** item;

**SELECT** state, COUNT(\*) **AS** "Number of customers" **FROM** customers **WHERE** salary > 10000 **GROUP** **BY** state;

**SELECT** department,**MIN**(salary) **AS** "Lowest salary" **FROM** employees **GROUP** **BY** department;

**SELECT** department,**MAX**(salary) **AS** "Highest salary" **FROM** employees **GROUP** **BY** department having dept\_no=110;

# Oracle HAVING Clause

In Oracle, HAVING Clause is used with GROUP BY Clause to restrict the groups of returned rows where condition is TRUE.

**Syntax:**

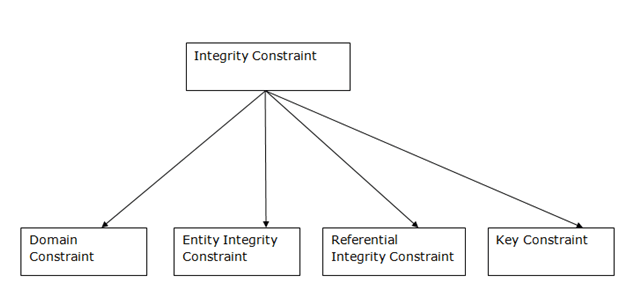
**SELECT** expression1, expression2, ... expression\_n,   aggregate\_function (aggregate\_expression)  **FROM** tables  **WHERE** conditions

**GROUP** **BY** expression1, expression2, ... expression\_n  **HAVING** having\_condition;

# Integrity Constraints

* Integrity constraints are a set of rules. It is used to maintain the quality of information.
* Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
* Thus, integrity constraint is used to guard against accidental damage to the database.

## Types of Integrity Constraint

### 1. Domain constraints

* Domain constraints can be defined as the definition of a valid set of values for an attribute.

**SQL Not Null Constraint :**

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

**Syntax to define a Not Null constraint**

[CONSTRAINT constraint name] NOT NULL

**For Example:** To create a employee table with Null value, the query would be like

CREATE TABLE employee( id number(5) not null,name char(20) CONSTRAINT nm\_nn NOT NULL,dept char(10), age number(2),salary number(10),location char(10) );

**Check Constraint :**

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

**Syntax to define a Check constraint:**

[CONSTRAINT constraint\_name] CHECK (condition)

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

**Check Constraint at column level:**

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

### 2. Entity integrity constraints

* The entity integrity constraint states that primary key value can't be null.
* This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.
* A table can contain a null value other than the primary key field.

**Primary key:**

This constraint defines a column or combination of columns which uniquely identifies each row in the table.

**Syntax to define a Primary key at column level:**

column name datatype [CONSTRAINT constraint\_name] PRIMARY KEY

**Syntax to define a Primary key at table level:**

[CONSTRAINT constraint\_name] PRIMARY KEY (column\_name1,column\_name2,..)

* **column\_name1, column\_name2** are the names of the columns which define the primary Key.
* The syntax within the bracket i.e. [CONSTRAINT constraint\_name] is optional.

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

**Primary Key at column level:**

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

**Unique Key:**

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

**Syntax to define a Unique key at column level:**

[CONSTRAINT constraint\_name] UNIQUE

**Syntax to define a Unique key at table level:**

[CONSTRAINT constraint\_name] UNIQUE(column\_name)

**For Example:**To create an employee table with Unique key, the query would be like,

**Unique Key at column level:**

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

### 3. Referential Integrity Constraints

* A referential integrity constraint is specified between two tables.
* In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

**Syntax to define a Foreign key at column level:**

[CONSTRAINT constraint\_name] REFERENCES Referenced\_Table\_name(column\_name)

**Syntax to define a Foreign key at table level**

[CONSTRAINT constraint\_name] FOREIGN KEY(column\_name) REFERENCES referenced\_table\_name(column\_name);

**For Example:**

1) Lets use the "product" table and "order\_items".   
  
**Foreign Key at column level:**

CREATE TABLE product ( product\_id number(5) CONSTRAINT pd\_id\_pk PRIMARY KEY,  product\_name char(20),supplier\_name char(20),unit\_price number(10));

CREATE TABLE order\_items( order\_id number(5) CONSTRAINT od\_id\_pk PRIMARY KEY,  
product\_id number(5) CONSTRAINT pd\_id\_fk REFERENCES product(product\_id), product\_name char(20),supplier\_name char(20),unit\_price number(10));

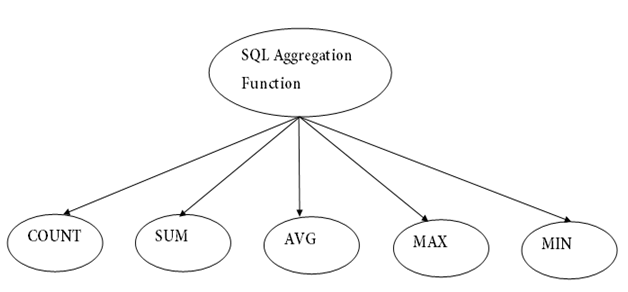
**Foreign Key at table level:**

CREATE TABLE order\_items( order\_id number(5) ,product\_id number(5),product\_name char(20),supplier\_name char(20),unit\_price number(10)CONSTRAINT od\_id\_pk PRIMARY KEY(order\_id),CONSTRAINT pd\_id\_fk FOREIGN KEY(product\_id) REFERENCES product(product\_id));

# SQL Aggregate Functions

* SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
* It is also used to summarize the data.

## Types of SQL Aggregation Function

### 1. COUNT FUNCTION

* COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
* COUNT function uses the COUNT(\*) that returns the count of all the rows in a specified table. COUNT(\*) considers duplicate and Null.

**Syntax**

COUNT(\*)

or

COUNT( [ALL|DISTINCT] expression )

**Sample table:**

**PRODUCT\_MAST**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PRODUCT** | **COMPANY** | **QTY** | **RATE** | **COST** |
| Item1 | Com1 | 2 | 10 | 20 |
| Item2 | Com2 | 3 | 25 | 75 |
| Item3 | Com1 | 2 | 30 | 60 |
| Item4 | Com3 | 5 | 10 | 50 |
| Item5 | Com2 | 2 | 20 | 40 |
| Item6 | Cpm1 | 3 | 25 | 75 |
| Item7 | Com1 | 5 | 30 | 150 |
| Item8 | Com1 | 3 | 10 | 30 |
| Item9 | Com2 | 2 | 25 | 50 |
| Item10 | Com3 | 4 | 30 | 120 |

**Example: COUNT()**

SELECT COUNT(\*) FROM PRODUCT\_MAST;

**Output:**

10

**Example: COUNT with WHERE**

SELECT COUNT(\*) FROM PRODUCT\_MAST WHERE RATE>=20;

**Output:**

7

**Example: COUNT() with DISTINCT**

SELECT COUNT(DISTINCT COMPANY) FROM PRODUCT\_MAST;

**Output:**

3

**Example: COUNT() with GROUP BY**

SELECT COMPANY, COUNT(\*) FROM PRODUCT\_MAST GROUP BY COMPANY;

**Output:**

Com1 5

Com2 3

Com3 2

**Example: COUNT() with HAVING**

SELECT COMPANY, COUNT(\*) FROM PRODUCT\_MAST GROUP BY COMPANY HAVING COUNT(\*)>2;

**Output:**

Com1 5

Com2 3

### 2. SUM Function

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

**Syntax**

SUM()

or

SUM( [ALL|DISTINCT] expression )

**Example: SUM()**

SELECT SUM(COST) FROM PRODUCT\_MAST;

**Output:**

670

**Example: SUM() with WHERE**

SELECT SUM(COST)FROM PRODUCT\_MAST WHERE QTY>3;

**Output:**

320

**Example: SUM() with GROUP BY**

SELECT SUM(COST) FROM PRODUCT\_MAST WHERE QTY>3 GROUP BY COMPANY;

**Output:**

Com1 150

Com2 170

**Example: SUM() with HAVING**

SELECT COMPANY, SUM(COST) FROM PRODUCT\_MAST GROUP BY COMPANY HAVING SUM(COST)>=170;

**Output:**

Com1 335

Com3 170

### 3. AVG function

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

**Syntax**

AVG()

or

AVG( [ALL|DISTINCT] expression )

**Example:**

SELECT AVG(COST) FROM PRODUCT\_MAST;

**Output:**

67.00

### 4. MAX Function

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

**Syntax**

MAX()

or

MAX( [ALL|DISTINCT] expression )

**Example:**

SELECT MAX(RATE) FROM PRODUCT\_MAST;

30

### 5. MIN Function

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

**Syntax**

MIN()

or

MIN( [ALL|DISTINCT] expression )

**Example:**

SELECT MIN(RATE) FROM PRODUCT\_MAST;

**Output:**

10

**SQL | Numeric (mathematical)Functions**

**Numeric Functions** are used to perform operations on numbers and return numbers.  
Following are the numeric functions defined in SQL:

**ABS():** It returns the absolute value of a number.

**Syntax:** SELECT ABS(-243.5) from dual;

**Output:**243.5

SQL> SELECT ABS(-10);

| ABS(10)

| 10

**ACOS():** It returns the cosine of a number.

**Syntax:**  SELECT ACOS(0.25);

**Output:**1.318116071652818

**ASIN():** It returns the arc sine of a number.

**Syntax:** SELECT ASIN(0.25);

**Output:**0.25268025514207865

**ATAN():** It returns the arc tangent of a number.

**Syntax:** SELECT ATAN(2.5);

**Output:**1.1902899496825317

**CEIL():** It returns the smallest integer value that is greater than or equal to a number.

**Syntax:** SELECT CEIL(25.75);

**Output:**26

**CEILING():** It returns the smallest integer value that is greater than or equal to a number.

**Syntax:** SELECT CEILING(25.75);

**Output:**26

**COS():** It returns the cosine of a number.

**Syntax:** SELECT COS(30);

**Output:**0.15425144988758405

**COT():** It returns the cotangent of a number.

**Syntax:** SELECT COT(6);

**Output:**-3.436353004180128

**DEGREES():** It converts a radian value into degrees.

**Syntax:** SELECT DEGREES(1.5);

**Output:**85.94366926962348

SQL>SELECT DEGREES(PI());

| DEGREES(PI())

| 180.000000

**DIV():** It is used for integer division.

**Syntax:** SELECT 10 DIV 5;

**Output:**2

**EXP():** It returns e raised to the power of number.

**Syntax:** SELECT EXP(1);

**Output:**2.718281828459045

**FLOOR():** It returns the largest integer value that is less than or equal to a number.

**Syntax:** SELECT FLOOR(25.75);

**Output:**25

**GREATEST():** It returns the greatest value in a list of expressions.

**Syntax:** SELECT GREATEST(30, 2, 36, 81, 125);

**Output:**125

**LEAST():** It returns the smallest value in a list of expressions.

**Syntax:** SELECT LEAST(30, 2, 36, 81, 125);

**Output:**2

**LN():** It returns the natural logarithm of a number.

**Syntax:** SELECT LN(2);

**Output:**0.6931471805599453

**LOG10():** It returns the base-10 logarithm of a number.

**Syntax:** SELECT LOG(2);

**Output:**0.6931471805599453

**LOG2():** It returns the base-2 logarithm of a number.

**Syntax:** SELECT LOG2(6);

**Output:**2.584962500721156

**MOD():** It returns the remainder of n divided by m.

**Syntax:** SELECT MOD(18, 4);

**Output:**2

**PI():** It returns the value of PI displayed with 6 decimal places.

**Syntax:** SELECT PI();

**Output:**3.141593

**POW():** It returns m raised to the nth power.

**Syntax:** SELECT POW(4, 2);

**Output:**16

**RADIANS():** It converts a value in degrees to radians.

**Syntax:** SELECT RADIANS(180);

**Output:**3.141592653589793

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

**Syntax:** SELECT RAND();

**Output:**0.33623238684258644

**ROUND():** It returns a number rounded to a certain number of decimal places.

**Syntax:** SELECT ROUND(5.553);

**Output:**6

**SIGN():** It returns a value indicating the sign of a number.

**Syntax:** SELECT SIGN(255.5);

**Output:**1

**SIN():** It returns the sine of a number.

**Syntax:** SELECT SIN(2);

**Output:**0.9092974268256817

**SQRT():** It returns the square root of a number.

**Syntax:** SELECT SQRT(25);

**Output:**5

**TAN():** It returns the tangent of a number.

**Syntax:** SELECT TAN(1.75);

**Output:**-5.52037992250933

**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

**TRUNCATE():** It returns 7.53635 truncated to 2 places right of the decimal point.

**Syntax:** SELECT TRUNCATE(7.53635, 2);

**Date Functions**

Dates are stored in the database as a number that contains both the calendar data information and the time information.  We already discussed date math, where the unit of measure is one day.  Date functions allow you to modify and compare date data types.  Dates can be tricky.  If you use SYSDATE to insert date columns in tables, you will not only get the date component but also the time component.

If you want to see all the records from today, a query based on today’s date will not match any rows in the database.  Basically, the times components of the two dates will not match.  This section will explain how to work with date data types and the next section will explain converting characters to dates and back again.  When working with calendars, you also have the problem that all months do not have the same number of days in them.  If you have a date and want the same date in three months, it becomes problematic.

**months\_between (l,e)**

This function returns the months between two dates.  If I wanted to know how many months an employee has worked for the company, I can use this function.  There is an emp\_hire\_date in the emp table.

SELECT  
  MONTHS\_BETWEEN(SYSDATE,EMP\_DATE\_OF\_HIRE)  
FROM  
  emp;  
  
MONTHS\_BETWEEN(SYSDATE,EMP\_DATE\_OF\_HIRE)  
----------------------------------------                              58.7710805  
                              70.7710805  
                              34.7710805  
                              46.7710805  
                              34.7710805  
                              82.7710805  
                               106.77108  
                               154.77108  
                               178.77108  
                               166.77108

10 rows selected.

Notice that it returns the fraction of a month.  You could use truncor round to make the results more readable.

**add\_months (d,n)**

The add\_months function gives you the same day, n number of months away.  The n can be positive or negative.

SELECT  
  SYSDATE,  
  ADD\_MONTHS(SYSDATE,1),  
  ADD\_MONTHS(SYSDATE,2),  
  ADD\_MONTHS(SYSDATE,3),  
  ADD\_MONTHS(SYSDATE,4),  
  ADD\_MONTHS(SYSDATE,5),  
  ADD\_MONTHS(SYSDATE,6)  
FROM  
  dual;  
  
SYSDATE   ADD\_MONTH ADD\_MONTH ADD\_MONTH ADD\_MONTH ADD\_MONTH ADD\_MONTH  
--------- --------- --------- --------- --------- --------- ---------  
24-JAN-05 24-FEB-05 24-MAR-05 24-APR-05 24-MAY-05 24-JUN-05 24-JUL-05

SELECT  
  SYSDATE,  
  ADD\_MONTHS(SYSDATE,-1),  
  ADD\_MONTHS(SYSDATE,-2),  
  ADD\_MONTHS(SYSDATE,-3),  
  ADD\_MONTHS(SYSDATE,-4),  
  ADD\_MONTHS(SYSDATE,-5),  
  ADD\_MONTHS(SYSDATE,-6)  
FROM  
  dual;

SYSDATE   ADD\_MONTH ADD\_MONTH ADD\_MONTH ADD\_MONTH ADD\_MONTH ADD\_MONTH  
--------- --------- --------- --------- --------- --------- ---------  
24-JAN-05 24-DEC-04 24-NOV-04 24-OCT-04 24-SEP-04 24-AUG-04 24-JUL-04

**last\_day (d)**

The last\_day function returns the last day of the month of the date d.  If you want to find the first day of the next month, simply add one to the last\_day results.

SELECT  
  SYSDATE,  
  LAST\_DAY(SYSDATE) EOM,  
  LAST\_DAY(SYSDATE)+1 FOM  
FROM dual;  
  
SYSDATE   EOM       FOM  
--------- --------- ---------  
24-JAN-05 31-JAN-05 01-FEB-05

**next\_day (d, day\_of\_week)**

The next\_day function returns the date of the day\_of\_week  after date d.  day\_of\_week can be the full name or abbreviation.  Below, we get the date for next Monday, next Friday, and the first Tuesday of next month.

SELECT  
  SYSDATE,  
  NEXT\_DAY(SYSDATE,'MONDAY') "Next Mon",  
  NEXT\_DAY(SYSDATE,'FRIDAY') "Next Fri",  
  NEXT\_DAY(LAST\_DAY(SYSDATE)+1,'TUESDAY') "First Tue"  
FROM dual;

SYSDATE   Next Mon  Next Fri  First Tue  
--------- --------- --------- ---------  
24-JAN-05 31-JAN-05 28-JAN-05 08-FEB-05

**round (d, format)**

We talked about the round function as a numeric function but it is also a date function.  The round function returns the date rounded to the format.

SELECT  
  SYSDATE,  
  ROUND(SYSDATE,'MONTH') Month,  
  ROUND(SYSDATE,'YEAR')  Year  
FROM  
  dual;

SYSDATE   MONTH     YEAR  
--------- --------- ---------  
24-JAN-05 01-FEB-05 01-JAN-05

Notice that SYSDATE is past midmonth so the month was rounded to the next month.  We are not past midyear, however, so the year was rounded to the beginning of the current year.

**trunc (d, format)**

As with the numeric trunc, the date version simply truncates the date to the level specified in the format.

SELECT  
  SYSDATE,  
  TRUNC(SYSDATE,'MONTH') Month,  
  TRUNC(SYSDATE,'YEAR')  Year  
FROM  
  dual;

SYSDATE   MONTH     YEAR  
--------- --------- ---------  
24-JAN-05 01-JAN-05 01-JAN-05

## Character Functions

Character functions operate on values of dataype  CHAR or VARCHAR.

#### LOWER

Returns a given string in lower case.

select LOWER(‘SAMI’) from dual;  
  
LOWER  
-------------  
sami

#### UPPER

Returns a given string in UPPER case.

select UPPER(‘Sami’) from dual;  
  
UPPER  
------------------  
SAMI

#### INITCAP

Returns a given string with Initial letter in capital.

select INITCAP(‘mohammed sami’) from dual;  
  
INITCAP  
------------------  
Mohammed Sami

#### LENGTH

Returns the length of a given string.

select length(‘mohammed sami’) from dual;  
  
LENGTH  
------------  
        13

#### SUBSTR

Returns a substring from a given string. Starting from position p to n characters.

For example the following query returns “sam” from the string “mohammed sami”.

select substr('mohammed sami',10,3) from dual;  
  
Substr  
--------  
sam

#### REPLACE

Replaces a given set of characters in a string with another set of characters.

Example

The following query replaces “mohd” with “mohammed” .

select replace('ali mohd khan','mohd','mohammed') from dual;  
  
REPLACE  
---------  
ali mohammed khan

#### REPLACE

Replaces a given set of characters in a string with another set of characters.

Example

The following query replaces “mohd” with “mohammed” .

select replace('ali mohd khan','mohd','mohammed') from dual;  
  
REPLACE  
---------  
ali mohammed khan

#### 

#### TRANSLATE

This function is used to encrypt characters. For example you can use this function to replace characters in a given string with your coded characters.

Example

The following query replaces characters A with B, B with C, C with D, D with E,...Z with A, and a with b,b with c,c with d, d with e ....z with a.

select translate('interface','ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz',  
      'BCDEFGHIJKLMNOPQRSTUVWXYZAbcdefghijklmnopqrstuvwxyza') “Encrypt” from dual;  
  
Encrypt  
-----------  
joufsgbdf

#### SOUNDEX

This function is used to check pronounciation rather than exact characters. For example many people write names as “smith” or “smyth” or “smythe” but they are pronounced as smith only.

Example

The following example compare those names which are spelled differently but are pronouced as “smith”.

Select ename from emp where soundex(ename)=soundex('smith');  
  
ENAME  
---------  
Smith  
Smyth  
Smythe

#### RPAD

Right pads a given string with a given character to n number of characters.

Example

The following query rights pad ename with '\*'  until it becomes 10 characters.

select rpad(ename,'\*',10) from emp;  
  
Ename  
----------  
Smith\*\*\*\*\*  
John\*\*\*\*\*\*  
Mohammed\*\*  
Sami\*\*\*\*\*\*

#### LPAD

Left pads a given string with a given character upto n number of characters.

Example

The following query left pads ename with '\*'  until it becomes 10 characters.

select lpad(ename,'\*',10) from emp;  
  
Ename  
----------  
\*\*\*\*\*Smith  
\*\*\*\*\*\*John  
\*\*Mohammed  
\*\*\*\*\*\*Sami

#### LTRIM

Trims blank spaces from a given string from left.

Example

The following query returns string “       Interface        “ left trimmed.

select ltrim('       Interface       ') from dual;  
  
Ltrim  
--------------  
Interface

#### RTRIM

Trims blank spaces from a given string from Right.

Example

The following query returns string “       Interface        “ right trimmed.

select rtrim('       Interface       ') from dual;  
  
Rtrim  
------------  
   Interface

#### TRIM

Trims a given character from left or right or both from a given string.

Example

The following query removes zero from left and right of a given string.

Select trim(0 from '00003443500') from dual;  
  
Trim  
----------  
34435

#### CONCAT

Combines a given string with another string.

Example

The following Query combines ename with literal string “ is a “ and jobid.

Select concat(concat(ename,' is a '),job) from emp;  
  
Concat  
----------------  
Smith is a clerk  
John is a Manager  
Sami is a G.Manager

# Oracle Subqueries

In general, a subquery is a query within another query,  the subquery is used to return data that will be used in the main query. Subqueries can be used in various places within a query (such as: SELECT, FROM, WHERE), this tutorial explains how to use subqueries in the Oracle WHERE clause.

## Using Oracle Subqueries

Subqueries are widely used to answer a question within another question. For example, which products cost more than product no. 54 ?  
To retrieve this information, you need to answer two question, each in a separate query  :

* What is product no. 54’s price?

|  |  |
| --- | --- |
|  | SELECT price  FROM products  WHERE product\_id = 54    -- Let's assume that this is the result :    price  ------  62 |

* Which products cost more than product no. 54?

|  |  |
| --- | --- |
|  | SELECT product\_id, product\_name, price  FROM products  WHERE price > 62 |

Instead of executing each query separately, you can combine the two queries, placing one query inside the other.

## Basic Oracle Subquery Syntax

|  |  |
| --- | --- |
|  | SELECT …  FROM table  WHERE condition (SELECT … FROM table) |

## Guidelines

|  |  |
| --- | --- |
|  | SELECT product\_id , product\_name , price  FROM products  WHERE price > ( SELECT price                   FROM products                   WHERE product\_id = 54) |

* The subquery is executed once before the main query, then the result returned by the subquery is submitted to the main
* The subquery must be enclosed by round brackets.
* Place subqueries on the right side of the comparison condition.
* A subquery cannot be placed in the Oracle GROUP BY Clause.
* Sub-queries can be divided into two main categories :
  + Single Row Subqueries – subqueries that return zero or one row to the outer SQL statement.
  + Multiple Row Subqueries – subqueries that return more than one row to the outer SQL statement.

## Oracle Single Row Subquery

You may use [comparison operators](http://ramkedem.com/en/oracle-comparison-operators/) (also referred as single-row operators) in the outer query to handle a subquery that returns a single value. This Oracle example would retrieve the products whose category number is the same as that of product 64.

|  |  |
| --- | --- |
|  | SELECT product\_name , product\_id , price , category\_id  FROM products  WHERE category\_id = (   SELECT category\_id                          FROM products                          WHERE product\_id = 64) |

* The subquery must return a single row; a subquery written without a Oracle WHERE Clause (hence usually returns more than one row) will generate an error.
* The subquery must return a single column; specifying more than one column in the subquery’s SELECT clause will result in an error.
* If you need to display all products whose category number is the same as that of product 54, excluding product 54, simply add this condition
* AND product\_id <> 54

|  |  |
| --- | --- |
|  | SELECT product\_name , product\_id , price , category\_id  FROM products  WHERE category\_id =  (SELECT category\_id                          FROM products                          WHERE product\_id = 54)  AND product\_id <> 54 |

* You can use group functions in a subquery to return a single row. For example: retrieve all products that cost more than the average price in category no. 60:

|  |  |
| --- | --- |
|  | SELECT product\_id , product\_name , price  FROM products  WHERE price > ( SELECT AVG(price)                  FROM products                  WHERE category\_id = 60) |

* A subquery also can be used in the Oracle HAVING Clause. This Oracle example retrieves a summary of the average price for each category, for all categories whose average price is greater than the average price of category no. 90:

|  |  |
| --- | --- |
|  | SELECT category\_id , AVG(price)  FROM products  GROUP BY category\_id  HAVING AVG(price)  > (SELECT AVG(price)                        FROM products                        WHERE category\_id = 90) |

* The next Oracle example would retrieve all products that cost more than product no. 54 and their category number equals the category number of product no. 42, not including product no. 42. This Oracle example consists of three queries, main query and two subqueries. The subqueries are executed first, generating the query results. Then the main query is processed and uses the values returned by these subqueries.

|  |  |
| --- | --- |
|  | SELECT product\_id , product\_name , price  FROM products  WHERE price > (SELECT price                  FROM products                  WHERE product\_id = 54 )  AND category\_id = (SELECT category\_id                       FROM products                        WHERE product\_id = 42)  AND product\_id <> 42 |

## Oracle Multiple Row Subquery

You may use the IN, ANY, or ALL operators (multiple row operators) in the outer query to handle a subquery that returns multiple rows, the multiple row operators expect one or more values.

The column below represents the prices of different products in category number 80, the following examples will use these values as the multiple row subquery result.

|  |  |  |
| --- | --- | --- |
|  | SELECT price  FROM products  WHERE category\_id = 80 | |
| price |
| 4300 |
| 5200 |
| 6700 |
| 8200 |
| 12500 |

## Oracle IN Operator

The Oracle IN operator allows comparing a column with a list of values returned from the subquery. This Oracle example would retrieve all products whose price is equal to one of the prices of products in category 80:

|  |
| --- |
| SELECT product\_id , product\_name , price  FROM products  WHERE price IN (SELECT price                   FROM products                   WHERE category\_id = 80) |

* In fact, the main query would look like the following to the Database Server :

|  |
| --- |
| SELECT product\_id , product\_name , price  FROM products  WHERE price IN (4300,5200,6700,8200,12500) |

* To display all products whose price is equal to one of the prices of products in category 80, excluding the products in category 80, simply use this condition : AND category\_id <> 80.

|  |
| --- |
| SELECT product\_id , product\_name , price  FROM products  WHERE price IN (SELECT price                   FROM products                   WHERE category\_id = 80)  AND category\_id <> 80 |

## Oracle ANY Operator

The Oracle ANY operator allows comparing a column with at least one of the values returned from the subequry. When using this operator, it is possible to work with the following comparison methods : >ANY, <ANY, =ANY

### ****> ANY****

The following Oracle example would retrieve all products whose price is greater than at least one of the prices of the products in category 80.

When seeking to know which value is greater than at least one of the values in a specific list, you actually seek to find the value that is greater than the minimum (because the requested value must be greater than at least one of the values in the list, no matter which value).

|  |
| --- |
| SELECT product\_id , product\_name , price  FROM products  WHERE price > ANY (SELECT price                      FROM products                      WHERE category\_id = 80) |

### ****< ANY****

The following Oracle example would retrieve all products whose price is lower than at least one of the prices of the products in category 80.

When seeking to know which value is less than at least one of the values in a specific list, you actually seek to find the value that is less than the maximum (because the requested value must be less than at least one of the values in the list, no matter which value).

|  |
| --- |
| SELECT product\_id , product\_name , price  FROM products  WHERE price < ANY (SELECT price                      FROM products                      WHERE category\_id = 80) |

### ****= ANY****

The following Oracle example would retrieve all products whose price is equal to at least one of the prices of products in category 80. =ANY is equivalent to IN

|  |
| --- |
| SELECT product\_id , product\_name , price  FROM products  WHERE price = ANY (SELECT price  FROM products  WHERE category\_id = 80) |

## ALL Operator

The Oracle ALL operator allows comparing a column with all of the values returned from the subquery. When using this operator, it is possible to work with the following comparison methods: < ALL, >ALL.

### ****> ALL****

The followng Oracle example would retrieve all products whose price is greater than the prices of all products in category 80.

When seeking to find the value that is greater than all values in a certain list, you actually look for a value that is greater than the maximum (for a value to be greater than all of the values, it must necessarily be greater than the maximum value in the list of values).

|  |  |
| --- | --- |
|  | SELECT product\_id , product\_name , price  FROM products  WHERE price > ALL (SELECT price                      FROM products                       WHERE category\_id = 80) |

# SQL | Join (Inner, Left, Right and Full Joins)

A SQL Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are:

* INNER JOIN
* LEFT JOIN
* RIGHT JOIN
* FULL JOIN

Consider the two tables below:

**Student**

[](https://www.geeksforgeeks.org/wp-content/uploads/table1-3.png)

**StudentCourse**

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/table5.png)

The simplest Join is INNER JOIN.

1. **INNER JOIN:** The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be same.  
   **Syntax**:

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

INNER JOIN table2

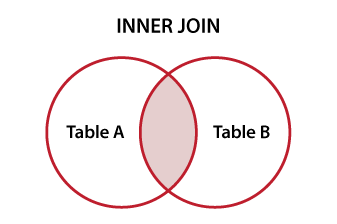
ON table1.matching\_column = table2.matching\_column;

**table1**: First table.

**table2**: Second table

**matching\_column**: Column common to both the tables.

**Note**: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.



**Example Queries(INNER JOIN)**

This query will show the names and age of students enrolled in different courses.

SELECT StudentCourse.COURSE\_ID, Student.NAME, Student.AGE FROM Student

INNER JOIN StudentCourse

ON Student.ROLL\_NO = StudentCourse.ROLL\_NO;

**Output**:  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/table22.png)

1. **LEFT JOIN**: This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain null. LEFT JOIN is also known as LEFT OUTER JOIN.**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

LEFT JOIN table2

ON table1.matching\_column = table2.matching\_column;

table1: First table.

table2: Second table

matching\_column: Column common to both the tables.

**Note**: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are same.  
[](https://i.stack.imgur.com/VkAT5.png)

**Example Queries(LEFT JOIN)**:

SELECT Student.NAME,StudentCourse.COURSE\_ID

FROM Student

LEFT JOIN StudentCourse

ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output**:  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/table31.png)

1. **RIGHT JOIN**: RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. The rows for which there is no matching row on left side, the result-set will contain null. RIGHT JOIN is also known as RIGHT OUTER JOIN.**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

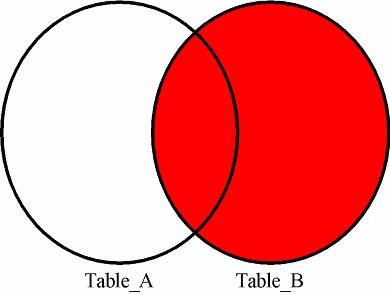
RIGHT JOIN table2

ON table1.matching\_column = table2.matching\_column;

table1: First table.

table2: Second table

matching\_column: Column common to both the tables.

**Note**: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are same.  


**Example Queries(RIGHT JOIN)**:

SELECT Student.NAME,StudentCourse.COURSE\_ID

FROM Student

RIGHT JOIN StudentCourse

ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output:**  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/table6.png)

1. **FULL JOIN:** FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain NULL values.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

FULL JOIN table2

ON table1.matching\_column = table2.matching\_column;

table1: First table.

table2: Second table

matching\_column: Column common to both the tables.



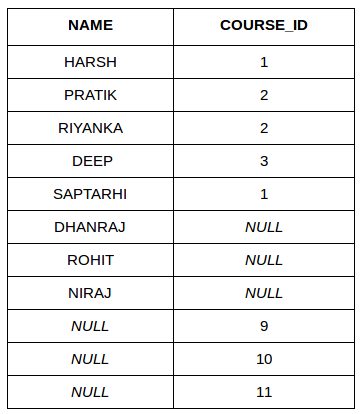
**Example Queries(FULL JOIN)**:

SELECT Student.NAME,StudentCourse.COURSE\_ID

FROM Student

FULL JOIN StudentCourse

ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output:**  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/table7.png)

# Views in SQL

* Views in SQL are considered as a virtual table. A view also contains rows and columns.
* To create the view, we can select the fields from one or more tables present in the database.
* A view can either have specific rows based on certain condition or all the rows of a table.

### Sample table:

**Student\_Detail**

|  |  |  |
| --- | --- | --- |
| **STU\_ID** | **NAME** | **ADDRESS** |
| 1 | Stephan | Delhi |
| 2 | Kathrin | Noida |
| 3 | David | Ghaziabad |
| 4 | Alina | Gurugram |

**Student\_Marks**

|  |  |  |  |
| --- | --- | --- | --- |
| **STU\_ID** | **NAME** | **MARKS** | **AGE** |
| 1 | Stephan | 97 | 19 |
| 2 | Kathrin | 86 | 21 |
| 3 | David | 74 | 18 |
| 4 | Alina | 90 | 20 |
| 5 | John | 96 | 18 |

## 1. Creating view

A view can be created using the **CREATE VIEW** statement. We can create a view from a single table or multiple tables.

**Syntax:**

CREATE VIEW view\_name AS SELECT column1, column2.....  FROM table\_name  WHERE condition;

## 2. Creating View from a single table

In this example, we create a View named DetailsView from the table Student\_Detail.

**Query:**

CREATE VIEW DetailsView AS SELECT NAME, ADDRESS FROM Student\_Details WHERE STU\_ID < 4;

Just like table query, we can query the view to view the data.

SELECT \* FROM DetailsView;

**Output:**

|  |  |
| --- | --- |
| **NAME** | **ADDRESS** |
| Stephan | Delhi |
| Kathrin | Noida |
| David | Ghaziabad |

## 3. Creating View from multiple tables

View from multiple tables can be created by simply include multiple tables in the SELECT statement.

In the given example, a view is created named MarksView from two tables Student\_Detail and Student\_Marks.

**Query:**

CREATE VIEW MarksView AS SELECT Student\_Detail.NAME, Student\_Detail.ADDRESS, Student\_Marks.MARKS

FROM Student\_Detail, Student\_Mark WHERE Student\_Detail.NAME = Student\_Marks.NAME;

To display data of View MarksView:

1. SELECT \* FROM MarksView;

|  |  |  |
| --- | --- | --- |
| **NAME** | **ADDRESS** | **MARKS** |
| Stephan | Delhi | 97 |
| Kathrin | Noida | 86 |
| David | Ghaziabad | 74 |
| Alina | Gurugram | 90 |

## 4. Deleting View

A view can be deleted using the Drop View statement.

**Syntax**

DROP VIEW view\_name;

**Example:**

If we want to delete the View **MarksView**, we can do this as:

### DROP VIEW MarksView;

## SQL Updating a View

A view can be updated with the CREATE OR REPLACE VIEW command.

### SQL CREATE OR REPLACE VIEW Syntax

CREATE OR REPLACE VIEW view\_name AS SELECT column1, column2, ...FROM table\_name  
WHERE condition;

The following SQL adds the "City" column to the "Brazil Customers" view:

### Example

CREATE OR REPLACE VIEW [Brazil Customers] AS SELECT CustomerName, ContactName, City  
FROM Customers WHERE Country = "Brazil";

# SQL Index

* Indexes are special lookup tables. It is used to retrieve data from the database very fast.
* An Index is used to speed up select queries and where clauses. But it shows down the data input with insert and update statements. Indexes can be created or dropped without affecting the data.
* An index in a database is just like an index in the back of a book.
* **For example:** When you reference all pages in a book that discusses a certain topic, you first have to refer to the index, which alphabetically lists all the topics and then referred to one or more specific page numbers.

## 1. Create Index statement

It is used to create an index on a table. It allows duplicate value.

**Syntax**

CREATE INDEX index\_name ON table\_name (column1, column2, ...);

**Example**

CREATE INDEX idx\_name ON Persons (LastName, FirstName);

## 2. Unique Index statement

It is used to create a unique index on a table. It does not allow duplicate value.

**Syntax**

CREATE UNIQUE INDEX index\_name ON table\_name (column1, column2, ...);

**Example**

CREATE UNIQUE INDEX websites\_idx ON websites (site\_name);

## 3. Drop Index Statement

It is used to delete an index in a table.

**Syntax**

DROP INDEX index\_name;

**Example**

DROP INDEX websites\_idx;

Chapter 2

## What is Relational Model

The relational model represents the database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship.

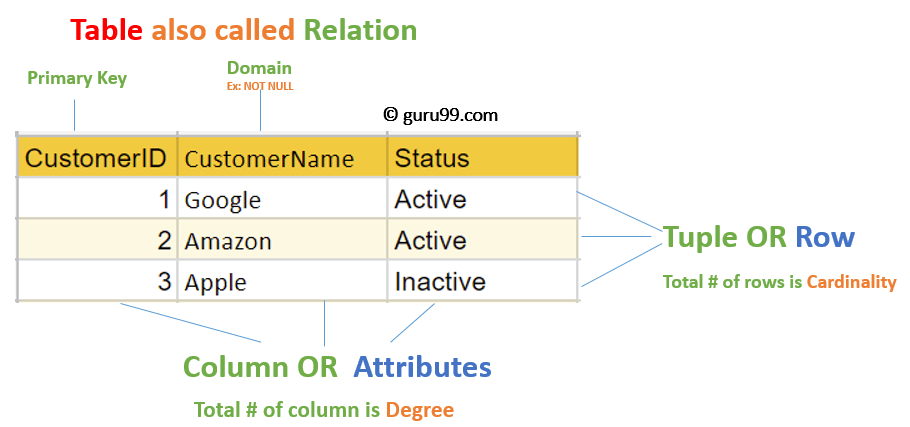
The table name and column names are helpful to interpret the meaning of values in each row. The data are represented as a set of relations. In the relational model, data are stored as tables. However, the physical storage of the data is independent of the way the data are logically organized.

Some popular Relational Database management systems are:

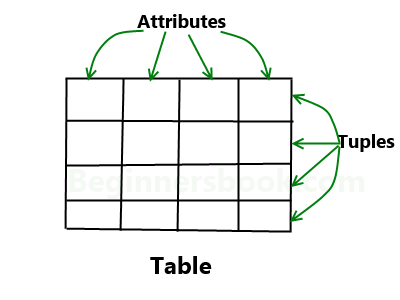
* DB2 and Informix Dynamic Server - IBM
* Oracle and RDB – Oracle
* SQL Server and Access - Microsoft

## Relational Model Concepts

1. **Attribute:** Each column in a Table. Attributes are the properties which define a relation. e.g., Student\_Rollno, NAME,etc.
2. **Tables** – In the Relational model the, relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.
3. **Tuple** – It is nothing but a single row of a table, which contains a single record.
4. **Relation Schema:** A relation schema represents the name of the attributes.
5. **Degree:** The total number of attributes which in the relation is called the degree of the relation.
6. **Cardinality:**Total number of rows present in the Table.
7. **Column:** The column represents the set of values for a specific attribute.
8. **Relation instance** – Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.
9. **Relation key** - Every row has one, two or multiple attributes, which is called relation key.
10. **Attribute domain** – Every attribute has some pre-defined value and scope which is known as attribute domain

[](https://www.guru99.com/images/1/091318_0803_RelationalD1.png)

# What is an attribute in DBMS? – Definition and explanation

In RDBMS, a table organizes data in rows and columns. The columns are known as attributes whereas the rows are known as records.  
  
**Example**: A school maintains the data of students in a table named “student”. Suppose the data they store in table is student id, student name & student age. To do this they have had three columns in the table: **student\_id**, **student\_age**, **student\_name**. The table looks like this:

|  |  |  |
| --- | --- | --- |
| student\_id | student\_age | student\_name |
| 101 | 12 | Jon |
| 102 | 13 | Arya |
| 103 | 12 | Sansa |

Here student\_id, student\_age and student\_name are the **attributes**.

**Domain Constraints**

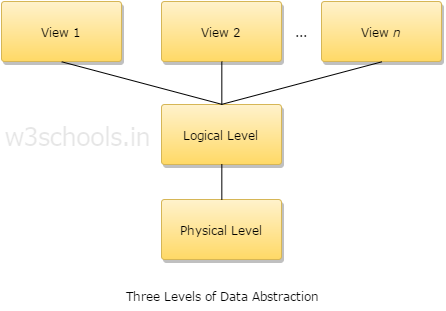
A domain is defined as the set of all unique values permitted for an attribute. For example, a domain of date is the set of all possible valid dates, a domain of integer is all possible whole numbers, a domain of day-of-week is Monday, Tuesday ... Sunday.

This in effect is defining rules for a particular attribute. If it is determined that an attribute is a date then it should be implemented in the database to prevent invalid dates being entered.

A classic example of this is where the data from a legacy system is loaded into a newly designed database. The new system is well designed. Columns that hold dates are defined as such whereas, in the old system, they were held as character strings. Much data is rejected because of invalid dates, eg 30 February 2000.

A **schema** can be defined as the design of a database. The overall description of the database is called the database schema. It can be categorized into three parts. These are:

* Physical Schema
* Logical Schema
* View Schema



A physical schema can be defined as the design of a database at its physical level. In this level, it is expressed how data is stored in blocks of storage.

A logical schema can be defined as the design of the database at its logical level. In this level, the programmers as well as the database administrator (DBA) work. At this level, data can be described as certain types of data records which can be stored in the form of data structures. However, the internal details (such as an implementation of data structure) will be remaining hidden at this level.

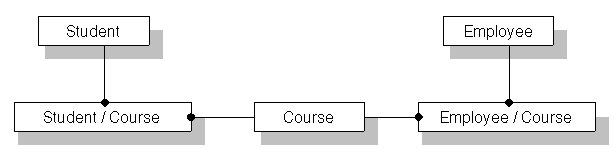
View schema can be defined as the design of the database at view level which generally describes end-user interaction with database systems.

For example: Let suppose, you are storing students' information on a student's table. At the physical level, these records are described as chunks of storage (in bytes, gigabytes, terabytes or higher) in memory, and these elements often remain hidden from the programmers. Then comes the logical level; here in logical level these records can be illustrated as fields and attributes along with their data type(s), their relationship with each other can be logically implemented. Programmers generally work at this level because they are aware of such things about database systems. At view level, a user can able to interact with the system, with the help of GUI and enter the details on the screen. The users are not aware of the fact how the data is stored and what data is stored; such details are hidden from them.

# Relational Schema for Example

The following figure shows the relational schema for this example. Employee in this example includes teachers and support personnel. Teachers include instructors, professors, and graduate students in a workstudy program. In this example, one set of type codes is used for all these types of employees.

There are two intersection entities in this schema: Student/Course and Employee/Course. These handle the two many-to-many relationships: 1) between Student and Course, and 2) between Employee and Course. In the first case, a Student may take many Courses and a Course may be taken by many Students. Similarly, in the second case, an Employee (one of the types of Teachers) may teach many Courses and a Course may be taught by many Teachers. Also see [more on many-to-many relationships.](https://www.service-architecture.com/articles/object-oriented-databases/more_on_many-to-many_relationships.html)



## Overview of database authorization

A privilege is a type of permission for an authorization name, or a permission to perform an action or a task. The privilege allows a user to create or access database resources. Privileges are stored in the database catalogs. Authorized users can pass on privileges on their own objects to other users by using the GRANT statement. Privileges can be granted to individual users, to groups, or to PUBLIC. PUBLIC is a special group that consists of all users, including future users. Users that are members of a group will indirectly take advantage of the privileges granted to the group, where groups are supported.

A role is a database object that groups one or more privileges. Roles can be assigned to users or groups or other roles by using the **GRANT** statement. Users that are members of roles have the privileges that are defined for the role with which to access data.

The forms of authorization, such as administrative authority, privileges, and Row and column access (RCAC) access, are discussed in [Authorization of Big SQL objects](https://www.ibm.com/support/knowledgecenter/SSPT3X_4.2.0/com.ibm.swg.im.infosphere.biginsights.analyze.doc/doc/bi_admin_biga_enable_authorization.html?view=kc#reference_b1x_pxx_b4). In addition, ownership of objects brings with it a degree of authorization on the objects created.

**Revoking privileges**

The REVOKE statement is used to revoke previously granted privileges. The revoking of a privilege from an authorization name revokes the privilege granted by all authorization names.

**Authorization matrix**

* Authorization rules are the controls incorporated in the data management system that restrict access to dada and actions that people can take on data.
* Authorization matrix mainly includes subject,object,action,and constraints
* Each row of the table indicates that a perticular subject is authorized to take a certain action on an object in the databse.
* Example

|  |  |  |  |
| --- | --- | --- | --- |
| **Subject** | **Object** | **Action** | **Constaint** |
| Sales department | Customer record | Insert | Credit limit $5000 |
| Order transaction | Customer record | Read | None |
| Terminal 14 | Customer record | Modify | Balance due only |
| Accounts department | Order record | Delete | None |
| John smith | Order record | Insert | Order amt $2000 |
| Program B22 | Order record | Modify | None |

# Query Language

Query language (QL) refers to any computer programming language that requests and retrieves data from database and information systems by sending queries. It works on user entered structured and formal programming command based queries to find and extract data from host databases.

Query language may also be termed database query language.

Query language is primarily created for creating, accessing and modifying data in and out from a database management system (DBMS). Typically, QL requires users to input a structured command that is similar and close to the English language querying construct.

For example, the SQL query: SELECT \* FROM customer;

The customer will retrieve all data from the customer records/table.

Relational Algebra

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either **unary** or **binary**. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

The fundamental operations of relational algebra are as follows −

* Select
* Project
* Union
* Set different
* Cartesian product
* Rename

We will discuss all these operations in the following sections.

Select Operation (σ)

It selects tuples that satisfy the given predicate from a relation.

**Notation** − σ*p*(r)

Where **σ** stands for selection predicate and **r** stands for relation. *p* is prepositional logic formula which may use connectors like **and, or,** and **not**. These terms may use relational operators like − =, ≠, ≥, < ,  >,  ≤.

**For example** −

**σ*subject = "database"*(Books)**

**Output** − Selects tuples from books where subject is 'database'.

**σsubject = "database" and price = "450"(Books)**

**Output** − Selects tuples from books where subject is 'database' and 'price' is 450.

**σsubject = "database" and price = "450" or year > "2010"(Books)**

**Output** − Selects tuples from books where subject is 'database' and 'price' is 450 or those books published after 2010.

**Project Operation (∏)**

It projects column(s) that satisfy a given predicate.

Notation − ∏A1, A2, An (r)

Where A1, A

2 , An are attribute names of relation **r**.

Duplicate rows are automatically eliminated, as relation is a set.

For example −

**∏subject, author (Books)**

Selects and projects columns named as subject and author from the relation Books.

Union Operation (∪)

It performs binary union between two given relations and is defined as −

r ∪ s = { t | t ∈ r or t ∈ s}

**Notation** − r U s

Where **r** and **s** are either database relations or relation result set (temporary relation).

For a union operation to be valid, the following conditions must hold −

* **r**, and **s** must have the same number of attributes.
* Attribute domains must be compatible.
* Duplicate tuples are automatically eliminated.

**∏ author (Books) ∪ ∏ author (Articles)**

**Output** − Projects the names of the authors who have either written a book or an article or both.

Set Difference (−)

The result of set difference operation is tuples, which are present in one relation but are not in the second relation.

**Notation** − **r** – **s**

Finds all the tuples that are present in **r** but not in **s**.

∏ author (Books) − ∏ author (Articles)

**Output** − Provides the name of authors who have written books but not articles.

**Cartesian Product (Χ)**

Combines information of two different relations into one.

**Notation** − r Χ s

Where **r** and **s** are relations and their output will be defined as −

r Χ s = { q t | q ∈ r and t ∈ s}

**σauthor = 'Balagurusamy'(Books Χ Articles)**

**Output** −Yields a relation, which shows all the books and articles written by Balagurusamy.

**Rename Operation (ρ)**

The results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. 'rename' operation is denoted with small Greek letter **rho** *ρ*.

**Notation** − *ρ* x (E)

Where the result of expression **E** is saved with name of **x**.

**Chapter 4**

**PL/SQL**

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL. PL/SQL is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java. This tutorial will give you great understanding on PL/SQL to proceed with Oracle database and other advanced RDBMS concepts.

PL/SQL has the following features −

* PL/SQL is tightly integrated with SQL.
* It offers extensive error checking.
* It offers numerous data types.
* It offers a variety of programming structures.
* It supports structured programming through functions and procedures.
* It supports object-oriented programming.
* It supports the development of web applications and server pages.

Advantages of PL/SQL

PL/SQL has the following advantages −

* SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
* PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
* PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
* PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
* Applications written in PL/SQL are fully portable.
* PL/SQL provides high security level.
* PL/SQL provides access to predefined SQL packages.
* PL/SQL provides support for Object-Oriented Programming.
* PL/SQL provides support for developing Web Applications and Server Pages.

Basic Syntax of PL/SQL which is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts −

|  |  |
| --- | --- |
| **S.No** | **Sections & Description** |
| 1 | **Declarations**  This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program. |
| 2 | **Executable Commands**  This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed. |
| 3 | **Exception Handling**  This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program. |

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Following is the basic structure of a PL/SQL block −

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling>

END;

The 'Hello World' Example

DECLARE

message varchar2(20):= 'Hello, World!';

BEGIN

dbms\_output.put\_line(message);

END;

/

The **end;** line signals the end of the PL/SQL block. To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code. When the above code is executed at the SQL prompt, it produces the following result −

Hello World

PL/SQL procedure successfully completed.

The PL/SQL Identifiers

PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.

The PL/SQL Delimiters

A delimiter is a symbol with a special meaning. Following is the list of delimiters in PL/SQL −

|  |  |
| --- | --- |
| **Delimiter** | **Description** |
| **+, -, \*, /** | Addition, subtraction/negation, multiplication, division |
| **%** | Attribute indicator |
| **'** | Character string delimiter |
| **.** | Component selector |
| **(,)** | Expression or list delimiter |
| **:** | Host variable indicator |
| **,** | Item separator |
| **"** | Quoted identifier delimiter |
| **=** | Relational operator |
| **@** | Remote access indicator |
| **;** | Statement terminator |
| **:=** | Assignment operator |
| **=>** | Association operator |
| **||** | Concatenation operator |
| **\*\*** | Exponentiation operator |
| **<<, >>** | Label delimiter (begin and end) |
| **/\*, \*/** | Multi-line comment delimiter (begin and end) |
| **--** | Single-line comment indicator |
| **..** | Range operator |
| **<, >, <=, >=** | Relational operators |
| **<>, '=, ~=, ^=** | Different versions of NOT EQUAL |

The PL/SQL Comments

Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow some form of comments.

The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by the PL/SQL compiler. The PL/SQL single-line comments start with the delimiter -- (double hyphen) and multi-line comments are enclosed by /\* and \*/.

DECLARE

-- variable declaration

message varchar2(20):= 'Hello, World!';

BEGIN

/\*

\* PL/SQL executable statement(s)

\*/

dbms\_output.put\_line(message);

END;

/

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

Hello World

PL/SQL procedure successfully completed.

PL/SQL Program Units

A PL/SQL unit is any one of the following −

* PL/SQL block
* Function
* Package
* Package body
* Procedure
* Trigger
* Type
* Type body

PL/SQL Scalar Data Types and Subtypes

PL/SQL Scalar Data Types and Subtypes come under the following categories −

|  |  |
| --- | --- |
| **S.No** | **Date Type & Description** |
| 1 | **Numeric**  Numeric values on which arithmetic operations are performed. |
| 2 | **Character**  Alphanumeric values that represent single characters or strings of characters. |
| 3 | **Boolean**  Logical values on which logical operations are performed. |
| 4 | **Datetime**  Dates and times. |

PL/SQL provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. You can use the subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding the PL/SQL code in another program, such as a Java program.

PL/SQL Numeric Data Types and Subtypes

Following table lists out the PL/SQL pre-defined numeric data types and their sub-types −

|  |  |
| --- | --- |
| **S.No** | **Data Type & Description** |
| 1 | **PLS\_INTEGER**  Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits |
| 2 | **BINARY\_INTEGER**  Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits |
| 3 | **BINARY\_FLOAT**  Single-precision IEEE 754-format floating-point number |
| 4 | **BINARY\_DOUBLE**  Double-precision IEEE 754-format floating-point number |
| 5 | **NUMBER(prec, scale)**  Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0 |
| 6 | **DEC(prec, scale)**  ANSI specific fixed-point type with maximum precision of 38 decimal digits |
| 7 | **DECIMAL(prec, scale)**  IBM specific fixed-point type with maximum precision of 38 decimal digits |
| 8 | **NUMERIC(pre, secale)**  Floating type with maximum precision of 38 decimal digits |
| 9 | **DOUBLE PRECISION**  ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits) |
| 10 | **FLOAT**  ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits) |
| 11 | **INT**  ANSI specific integer type with maximum precision of 38 decimal digits |
| 12 | **INTEGER**  ANSI and IBM specific integer type with maximum precision of 38 decimal digits |
| 13 | **SMALLINT**  ANSI and IBM specific integer type with maximum precision of 38 decimal digits |
| 14 | **REAL**  Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits) |

Following is a valid declaration −

DECLARE

num1 INTEGER;

num2 REAL;

num3 DOUBLE PRECISION;

BEGIN

null;

END;

/

When the above code is compiled and executed, it produces the following result −

PL/SQL procedure successfully completed

PL/SQL Character Data Types and Subtypes

Following is the detail of PL/SQL pre-defined character data types and their sub-types −

|  |  |
| --- | --- |
| **S.No** | **Data Type & Description** |
| 1 | **CHAR**  Fixed-length character string with maximum size of 32,767 bytes |
| 2 | **VARCHAR2**  Variable-length character string with maximum size of 32,767 bytes |
| 3 | **RAW**  Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL |
| 4 | **NCHAR**  Fixed-length national character string with maximum size of 32,767 bytes |
| 5 | **NVARCHAR2**  Variable-length national character string with maximum size of 32,767 bytes |
| 6 | **LONG**  Variable-length character string with maximum size of 32,760 bytes |
| 7 | **LONG RAW**  Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL |
| 8 | **ROWID**  Physical row identifier, the address of a row in an ordinary table |
| 9 | **UROWID**  Universal row identifier (physical, logical, or foreign row identifier) |

PL/SQL Boolean Data Types

The **BOOLEAN** data type stores logical values that are used in logical operations. The logical values are the Boolean values **TRUE** and **FALSE** and the value **NULL**.

However, SQL has no data type equivalent to BOOLEAN. Therefore, Boolean values cannot be used in −

* SQL statements
* Built-in SQL functions (such as **TO\_CHAR**)
* PL/SQL functions invoked from SQL statements

PL/SQL Datetime and Interval Types

The **DATE** datatype is used to store fixed-length datetimes, which include the time of day in seconds since midnight. Valid dates range from January 1, 4712 BC to December 31, 9999 AD.

The default date format is set by the Oracle initialization parameter NLS\_DATE\_FORMAT. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year. For example, 01-OCT-12.

Each DATE includes the century, year, month, day, hour, minute, and second. The following table shows the valid values for each field −

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Valid Datetime Values** | **Valid Interval Values** |
| YEAR | -4712 to 9999 (excluding year 0) | Any nonzero integer |
| MONTH | 01 to 12 | 0 to 11 |
| DAY | 01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale) | Any nonzero integer |
| HOUR | 00 to 23 | 0 to 23 |
| MINUTE | 00 to 59 | 0 to 59 |
| SECOND | 00 to 59.9(n), where 9(n) is the precision of time fractional seconds | 0 to 59.9(n), where 9(n) is the precision of interval fractional seconds |
| TIMEZONE\_HOUR | -12 to 14 (range accommodates daylight savings time changes) | Not applicable |
| TIMEZONE\_MINUTE | 00 to 59 | Not applicable |
| TIMEZONE\_REGION | Found in the dynamic performance view V$TIMEZONE\_NAMES | Not applicable |
| TIMEZONE\_ABBR | Found in the dynamic performance view V$TIMEZONE\_NAMES | Not applicable |

PL/SQL Large Object (LOB) Data Types

Large Object (LOB) data types refer to large data items such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Following are the predefined PL/SQL LOB data types −

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Description** | **Size** |
| BFILE | Used to store large binary objects in operating system files outside the database. | System-dependent. Cannot exceed 4 gigabytes (GB). |
| BLOB | Used to store large binary objects in the database. | 8 to 128 terabytes (TB) |
| CLOB | Used to store large blocks of character data in the database. | 8 to 128 TB |
| NCLOB | Used to store large blocks of NCHAR data in the database. | 8 to 128 TB |

PL/SQL User-Defined Subtypes

A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values.

PL/SQL predefines several subtypes in package **STANDARD**. For example, PL/SQL predefines the subtypes **CHARACTER** and **INTEGER** as follows −

SUBTYPE CHARACTER IS CHAR;

SUBTYPE INTEGER IS NUMBER(38,0);

You can define and use your own subtypes. The following program illustrates defining and using a user-defined subtype −

DECLARE

SUBTYPE name is char(20);

SUBTYPE message IS varchar2(100);

salutation name;

greetings message;

BEGIN

salutation := 'Reader ';

greetings := 'Welcome to the World of PL/SQL';

dbms\_output.put\_line('Hello ' || salutation || greetings);

END;

/

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

Hello Reader Welcome to the World of PL/SQL

PL/SQL procedure successfully completed.

NULLs in PL/SQL

PL/SQL NULL values represent **missing** or **unknown data** and they are not an integer, a character, or any other specific data type. Note that **NULL** is not the same as an empty data string or the null character value **'\0'**. A null can be assigned but it cannot be equated with anything, including itself.

## Variable Declaration in PL/SQL

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is −

variable\_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial\_value]

Where, *variable\_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type which we already have discussed in the last chapter. Some valid variable declarations along with their definition are shown below −

sales number(10, 2);

pi CONSTANT double precision := 3.1415;

name varchar2(25);

address varchar2(100);

When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations. For example −

sales number(10, 2);

name varchar2(25);

address varchar2(100);

## Initializing Variables in PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following −

* The **DEFAULT** keyword
* The **assignment** operator

For example −

counter binary\_integer := 0;

greetings varchar2(20) DEFAULT 'Have a Good Day';

You can also specify that a variable should not have a **NULL** value using the **NOT NULL** constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometimes programs would produce unexpected results. Try the following example which makes use of various types of variables −

DECLARE

a integer := 10;

b integer := 20;

c integer;

f real;

BEGIN

c := a + b;

dbms\_output.put\_line('Value of c: ' || c);

f := 70.0/3.0;

dbms\_output.put\_line('Value of f: ' || f);

END;

/

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

Value of c: 30

Value of f: 23.333333333333333333

PL/SQL procedure successfully completed.

## Variable Scope in PL/SQL

PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope −

* **Local variables** − Variables declared in an inner block and not accessible to outer blocks.
* **Global variables** − Variables declared in the outermost block or a package.

Following example shows the usage of **Local** and **Global** variables in its simple form −

DECLARE

-- Global variables

num1 number := 95;

num2 number := 85;

BEGIN

dbms\_output.put\_line('Outer Variable num1: ' || num1);

dbms\_output.put\_line('Outer Variable num2: ' || num2);

DECLARE

-- Local variables

num1 number := 195;

num2 number := 185;

BEGIN

dbms\_output.put\_line('Inner Variable num1: ' || num1);

dbms\_output.put\_line('Inner Variable num2: ' || num2);

END;

END;

/

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

Outer Variable num1: 95

Outer Variable num2: 85

Inner Variable num1: 195

Inner Variable num2: 185

PL/SQL procedure successfully completed.

## Assigning SQL Query Results to PL/SQL Variables

You can use the **SELECT INTO** statement of SQL to assign values to PL/SQL variables. For each item in the **SELECT list**, there must be a corresponding, type-compatible variable in the **INTO list**. The following example illustrates the concept. Let us create a table named CUSTOMERS −

(**For SQL statements, please refer to the**[SQL tutorial](https://www.tutorialspoint.com/sql/index.htm))

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25),

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

Table Created

Let us now insert some values in the table −

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

VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

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

VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

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

VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

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

VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

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

VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

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

VALUES (6, 'Komal', 22, 'MP', 4500.00 );

The following program assigns values from the above table to PL/SQL variables using the **SELECT INTO clause** of SQL −

DECLARE

c\_id customers.id%type := 1;

c\_name customers.name%type;

c\_addr customers.address%type;

c\_sal customers.salary%type;

BEGIN

SELECT name, address, salary INTO c\_name, c\_addr, c\_sal

FROM customers

WHERE id = c\_id;

dbms\_output.put\_line

('Customer ' ||c\_name || ' from ' || c\_addr || ' earns ' || c\_sal);

END;

/

**Decision-making structures** require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical conditional (i.e., decision making) structure found in most of the programming languages −



PL/SQL programming language provides following types of decision-making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **S.No** | **Statement & Description** |
| 1 | [IF - THEN statement](https://www.tutorialspoint.com/plsql/plsql_if_then.htm)  The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing. |
| 2 | [IF-THEN-ELSE statement](https://www.tutorialspoint.com/plsql/plsql_if_then_else.htm)  **IF statement** adds the keyword **ELSE** followed by an alternative sequence of statement. If the condition is false or NULL, then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed. |
| 3 | [IF-THEN-ELSIF statement](https://www.tutorialspoint.com/plsql/plsql_if_then_elsif.htm)  It allows you to choose between several alternatives. |
| 4 | [Case statement](https://www.tutorialspoint.com/plsql/plsql_case_statement.htm)  Like the IF statement, the **CASE statement** selects one sequence of statements to execute.  However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives. |
| 5 | [Searched CASE statement](https://www.tutorialspoint.com/plsql/plsql_searched_case.htm)  The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values. |
| 6 | [nested IF-THEN-ELSE](https://www.tutorialspoint.com/plsql/plsql_nested_if.htm)  You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s). |

The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is **TRUE**, the statements get executed, and if the condition is **FALSE** or **NULL**, then the **IF** statement does nothing.

Syntax

Syntax for **IF-THEN** statement is −

IF condition THEN

S;

END IF;

Where *condition* is a Boolean or relational condition and S is a simple or compound statement. Following is an example of the IF-THEN statement −

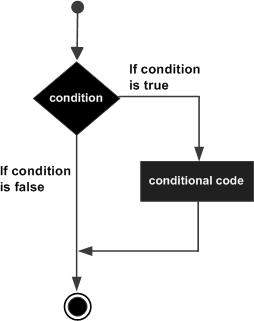
IF (a <= 20) THEN

c:= c+1;

END IF;

If the Boolean expression condition evaluates to true, then the block of code inside the **if statement** will be executed. If the Boolean expression evaluates to false, then the first set of code after the end of the **if statement** (after the closing end if) will be executed.

Flow Diagram



Example 1

Let us try an example that will help you understand the concept −

DECLARE

a number(2) := 10;

BEGIN

a:= 10;

-- check the boolean condition using if statement

IF( a < 20 ) THEN

-- if condition is true then print the following

dbms\_output.put\_line('a is less than 20 ' );

END IF;

dbms\_output.put\_line('value of a is : ' || a);

END;

/

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

a is less than 20

value of a is : 10

A sequence of **IF-THEN** statements can be followed by an optional sequence of **ELSE** statements, which execute when the condition is **FALSE**.

Syntax

Syntax for the IF-THEN-ELSE statement is −

IF condition THEN

S1;

ELSE

S2;

END IF;

Where, *S1* and *S2* are different sequence of statements. In the **IF-THEN-ELSE statements**, when the test condition is TRUE, the statement *S1* is executed and *S2*is skipped; when the test condition is FALSE, then *S1* is bypassed and statement *S2*is executed. For example −

IF color = red THEN

dbms\_output.put\_line('You have chosen a red car')

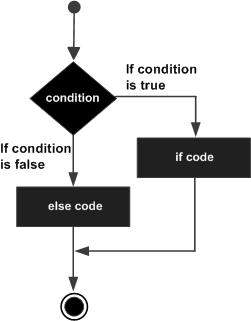
ELSE

dbms\_output.put\_line('Please choose a color for your car');

END IF;

If the Boolean expression condition evaluates to true, then the **if-then block of code** will be executed otherwise the else block of code will be executed.

Flow Diagram



Example

Let us try an example that will help you understand the concept −

DECLARE

a number(3) := 100;

BEGIN

-- check the boolean condition using if statement

IF( a < 20 ) THEN

-- if condition is true then print the following

dbms\_output.put\_line('a is less than 20 ' );

ELSE

dbms\_output.put\_line('a is not less than 20 ' );

END IF;

dbms\_output.put\_line('value of a is : ' || a);

END;

/

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

a is not less than 20

value of a is : 100

The **IF-THEN-ELSIF** statement allows you to choose between several alternatives. An **IF-THEN** statement can be followed by an optional **ELSIF...ELSE** statement. The **ELSIF** clause lets you add additional conditions.

When using **IF-THEN-ELSIF** statements there are a few points to keep in mind.

* It's ELSIF, not ELSEIF.
* An IF-THEN statement can have zero or one ELSE's and it must come after any ELSIF's.
* An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.
* Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

Syntax

The syntax of an **IF-THEN-ELSIF** Statement in PL/SQL programming language is −

IF(boolean\_expression 1)THEN

S1; -- Executes when the boolean expression 1 is true

ELSIF( boolean\_expression 2) THEN

S2; -- Executes when the boolean expression 2 is true

ELSIF( boolean\_expression 3) THEN

S3; -- Executes when the boolean expression 3 is true

ELSE

S4; -- executes when the none of the above condition is true

END IF;

Example

DECLARE

a number(3) := 100;

BEGIN

IF ( a = 10 ) THEN

dbms\_output.put\_line('Value of a is 10' );

ELSIF ( a = 20 ) THEN

dbms\_output.put\_line('Value of a is 20' );

ELSIF ( a = 30 ) THEN

dbms\_output.put\_line('Value of a is 30' );

ELSE

dbms\_output.put\_line('None of the values is matching');

END IF;

dbms\_output.put\_line('Exact value of a is: '|| a );

END;

/

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

None of the values is matching

Exact value of a is: 100

the **CASE** statement uses a selector rather than multiple Boolean expressions. A selector is an expression, the value of which is used to select one of several alternatives.

Syntax

The syntax for the case statement in PL/SQL is −

CASE selector

WHEN 'value1' THEN S1;

WHEN 'value2' THEN S2;

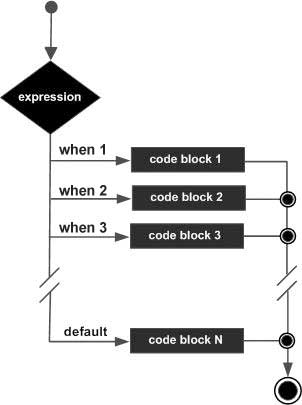
WHEN 'value3' THEN S3;

...

ELSE Sn; -- default case

END CASE;

Flow Diagram



Example

DECLARE

grade char(1) := 'A';

BEGIN

CASE grade

when 'A' then dbms\_output.put\_line('Excellent');

when 'B' then dbms\_output.put\_line('Very good');

when 'C' then dbms\_output.put\_line('Well done');

when 'D' then dbms\_output.put\_line('You passed');

when 'F' then dbms\_output.put\_line('Better try again');

else dbms\_output.put\_line('No such grade');

END CASE;

END;

/

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

Excellent

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −



PL/SQL provides the following types of loop to handle the looping requirements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **S.No** | **Loop Type & Description** |
| 1 | [PL/SQL Basic LOOP](https://www.tutorialspoint.com/plsql/plsql_basic_loop.htm)  In this loop structure, sequence of statements is enclosed between the LOOP and the END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop. |
| 2 | [PL/SQL WHILE LOOP](https://www.tutorialspoint.com/plsql/plsql_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| 3 | [PL/SQL FOR LOOP](https://www.tutorialspoint.com/plsql/plsql_for_loop.htm)  Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 4 | [Nested loops in PL/SQL](https://www.tutorialspoint.com/plsql/plsql_nested_loops.htm)  You can use one or more loop inside any another basic loop, while, or for loop. |

Basic loop structure encloses sequence of statements in between the **LOOP** and **END LOOP** statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

Syntax

The syntax of a basic loop in PL/SQL programming language is −

LOOP

Sequence of statements;

END31 LOOP;

Here, the sequence of statement(s) may be a single statement or a block of statements. An **EXIT statement** or an **EXIT WHEN statement** is required to break the loop.

Example

DECLARE

x number := 10;

BEGIN

LOOP

dbms\_output.put\_line(x);

x := x + 10;

IF x > 50 THEN

exit;

END IF;

END LOOP;

-- after exit, control resumes here

dbms\_output.put\_line('After Exit x is: ' || x);

END;

/

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

10

20

30

40

50

After Exit x is: 60

PL/SQL procedure successfully completed.

You can use the **EXIT WHEN** statement instead of the **EXIT** statement −

DECLARE

x number := 10;

BEGIN

LOOP

dbms\_output.put\_line(x);

x := x + 10;

exit WHEN x > 50;

END LOOP;

-- after exit, control resumes here

dbms\_output.put\_line('After Exit x is: ' || x);

END;

/

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

10

20

30

40

50

After Exit x is: 60

PL/SQL procedure successfully completed.

A **WHILE LOOP** statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

WHILE condition LOOP

sequence\_of\_statements

END LOOP;

Example

DECLARE

a number(2) := 10;

BEGIN

WHILE a > 1 LOOP

dbms\_output.put\_line('value of a: ' || a);

a := a - 1;

END LOOP;

END;

/

A **FOR LOOP** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

FOR counter IN initial\_value .. final\_value LOOP

sequence\_of\_statements;

END LOOP;

Following is the flow of control in a **For Loop** −

* The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
* Next, the condition, i.e., *initial\_value .. final\_value* is evaluated. If it is TRUE, the body of the loop is executed. If it is FALSE, the body of the loop does not execute and the flow of control jumps to the next statement just after the for loop.
* After the body of the for loop executes, the value of the counter variable is increased or decreased.
* The condition is now evaluated again. If it is TRUE, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes FALSE, the FOR-LOOP terminates.

Following are some special characteristics of PL/SQL for loop −

* The *initial\_value* and *final\_value* of the loop variable or counter can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE\_ERROR.
* The *initial\_value* need not be 1; however, the **loop counter increment (or decrement) must be 1**.
* PL/SQL allows the determination of the loop range dynamically at run time.

Example

DECLARE

a number(2);

BEGIN

FOR a in 10 .. 20 LOOP

dbms\_output.put\_line('value of a: ' || a);

END LOOP;

END;

/

PL/SQL allows using one loop inside another loop. Following section shows a few examples to illustrate the concept.

The syntax for a nested basic LOOP statement in PL/SQL is as follows −

LOOP

Sequence of statements1

LOOP

Sequence of statements2

END LOOP;

END LOOP;

The syntax for a nested FOR LOOP statement in PL/SQL is as follows −

FOR counter1 IN initial\_value1 .. final\_value1 LOOP

sequence\_of\_statements1

FOR counter2 IN initial\_value2 .. final\_value2 LOOP

sequence\_of\_statements2

END LOOP;

END LOOP;

The syntax for a nested WHILE LOOP statement in Pascal is as follows −

WHILE condition1 LOOP

sequence\_of\_statements1

WHILE condition2 LOOP

sequence\_of\_statements2

END LOOP;

END LOOP;

Example

The following program uses a nested basic loop to find the prime numbers from 2 to 100 −

DECLARE

i number(3);

j number(3);

BEGIN

i := 2;

LOOP

j:= 2;

LOOP

exit WHEN ((mod(i, j) = 0) or (j = i));

j := j +1;

END LOOP;

IF (j = i ) THEN

dbms\_output.put\_line(i || ' is prime');

END IF;

i := i + 1;

exit WHEN i = 50;

END LOOP;

END;

/

## Labeling a PL/SQL Loop

PL/SQL loops can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop.

The following program illustrates the concept −

DECLARE

i number(1);

j number(1);

BEGIN

<< outer\_loop >>

FOR i IN 1..3 LOOP

<< inner\_loop >>

FOR j IN 1..3 LOOP

dbms\_output.put\_line('i is: '|| i || ' and j is: ' || j);

END loop inner\_loop;

END loop outer\_loop;

END;

/

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

i is: 1 and j is: 1

i is: 1 and j is: 2

i is: 1 and j is: 3

i is: 2 and j is: 1

i is: 2 and j is: 2

i is: 2 and j is: 3

i is: 3 and j is: 1

i is: 3 and j is: 2

i is: 3 and j is: 3

PL/SQL procedure successfully completed.

## The Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

PL/SQL supports the following control statements. Labeling loops also help in taking the control outside a loop. Click the following links to check their details.

|  |  |
| --- | --- |
| **S.No** | **Control Statement & Description** |
| 1 | [EXIT statement](https://www.tutorialspoint.com/plsql/plsql_exit_statement.htm)  The Exit statement completes the loop and control passes to the statement immediately after the END LOOP. |
| 2 | [CONTINUE statement](https://www.tutorialspoint.com/plsql/plsql_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | [GOTO statement](https://www.tutorialspoint.com/plsql/plsql_goto_statement.htm)  Transfers control to the labeled statement. Though it is not advised to use the GOTO statement in your program. |

Declaring String Variables

Oracle database provides numerous string datatypes, such as CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB. The datatypes prefixed with an **'N'**are **'national character set'** datatypes, that store Unicode character data.

If you need to declare a variable-length string, you must provide the maximum length of that string. For example, the VARCHAR2 data type. The following example illustrates declaring and using some string variables −

DECLARE

name varchar2(20);

company varchar2(30);

introduction clob;

choice char(1);

BEGIN

name := 'John Smith';

company := 'Infotech';

introduction := ' Hello! I''m John Smith from Infotech.';

choice := 'y';

IF choice = 'y' THEN

dbms\_output.put\_line(name);

dbms\_output.put\_line(company);

dbms\_output.put\_line(introduction);

END IF;

END;

/

## Creating a Procedure

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

CREATE [OR REPLACE] PROCEDURE procedure\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

{IS | AS}

BEGIN

< procedure\_body >

END procedure\_name;

Where,

* *procedure-name* specifies the name of the procedure.
* [OR REPLACE] option allows the modification of an existing procedure.
* 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.
* *procedure-body* contains the executable part.
* The AS keyword is used instead of the IS keyword for creating a standalone procedure.

### Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

dbms\_output.put\_line('Hello World!');

END;

/

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

Procedure created.

## Executing a Standalone Procedure

A standalone procedure can be called in two ways −

* Using the **EXECUTE** keyword
* Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as −

EXECUTE greetings;

The above call will display −

Hello World

PL/SQL procedure successfully completed.

The procedure can also be called from another PL/SQL block −

BEGIN

greetings;

END;

/

The above call will display −

Hello World

PL/SQL procedure successfully completed.

## Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

DROP PROCEDURE procedure-name;

You can drop the greetings procedure by using the following statement −

DROP PROCEDURE greetings;

## Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms −

|  |  |
| --- | --- |
| **S.No** | **Parameter Mode & Description** |
| 1 | **IN**  An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**. |
| 2 | **OUT**  An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**. |
| 3 | **IN OUT**  An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read.  The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.** |

### IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

DECLARE

a number;

b number;

c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS

BEGIN

IF x < y THEN

z:= x;

ELSE

z:= y;

END IF;

END;

BEGIN

a:= 23;

b:= 45;

findMin(a, b, c);

dbms\_output.put\_line(' Minimum of (23, 45) : ' || c);

END;

/

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

Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.

### IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

DECLARE

a number;

PROCEDURE squareNum(x IN OUT number) IS

BEGIN

x := x \* x;

END;

BEGIN

a:= 23;

squareNum(a);

dbms\_output.put\_line(' Square of (23): ' || a);

END;

/

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

Square of (23): 529

PL/SQL procedure successfully completed.

## Methods for Passing Parameters

Actual parameters can be passed in three ways −

* Positional notation
* Named notation
* Mixed notation

### Positional Notation

In positional notation, you can call the procedure as −

findMin(a, b, c, d);

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, **a** is substituted for **x, b** is substituted for **y, c** is substituted for **z** and **d** is substituted for **m**.

### Named Notation

In named notation, the actual parameter is associated with the formal parameter using the **arrow symbol ( => )**. The procedure call will be like the following −

findMin(x => a, y => b, z => c, m => d);

### Mixed Notation

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal −

findMin(a, b, c, m => d);

However, this is not legal:

findMin(x => a, b, c, d);

## 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.

### 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.

We will use the CUSTOMERS table, which we had created in the [PL/SQL Variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) chapter −

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

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 −

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.

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.

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

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 | Ahmedabad | 2500.00 |

| 2 | Khilan | 25 | Delhi | 2000.00 |

| 3 | kaushik | 23 | Kota | 2500.00 |

| 4 | Chaitali | 25 | Mumbai | 7000.00 |

| 5 | Hardik | 27 | Bhopal | 9000.00 |

| 6 | Komal | 22 | MP | 5000.00 |

+----+----------+-----+-----------+----------+

## 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 &minua;

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 Ahmedabad

2 Khilan Delhi

3 kaushik Kota

4 Chaitali Mumbai

5 Hardik Bhopal

6 Komal MP

PL/SQL procedure successfully completed.

An exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions −

* System-defined exceptions
* User-defined exceptions

## Syntax for Exception Handling

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using ***WHEN others THEN*** −

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling goes here >

WHEN exception1 THEN

exception1-handling-statements

WHEN exception2 THEN

exception2-handling-statements

WHEN exception3 THEN

exception3-handling-statements

........

WHEN others THEN

exception3-handling-statements

END;

### Example

Let us write a code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters −

DECLARE

c\_id customers.id%type := 8;

c\_name customerS.Name%type;

c\_addr customers.address%type;

BEGIN

SELECT name, address INTO c\_name, c\_addr

FROM customers

WHERE id = c\_id;

DBMS\_OUTPUT.PUT\_LINE ('Name: '|| c\_name);

DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);

EXCEPTION

WHEN no\_data\_found THEN

dbms\_output.put\_line('No such customer!');

WHEN others THEN

dbms\_output.put\_line('Error!');

END;

/

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

No such customer!

PL/SQL procedure successfully completed.

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO\_DATA\_FOUND**, which is captured in the **EXCEPTION block**.

## Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax for raising an exception −

DECLARE

exception\_name EXCEPTION;

BEGIN

IF condition THEN

RAISE exception\_name;

END IF;

EXCEPTION

WHEN exception\_name THEN

statement;

END;

You can use the above syntax in raising the Oracle standard exception or any user-defined exception. In the next section, we will give you an example on raising a user-defined exception. You can raise the Oracle standard exceptions in a similar way.

## User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure **DBMS\_STANDARD.RAISE\_APPLICATION\_ERROR**.

The syntax for declaring an exception is −

DECLARE

my-exception EXCEPTION;

### Example

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception **invalid\_id** is raised.

DECLARE

c\_id customers.id%type := &cc\_id;

c\_name customerS.Name%type;

c\_addr customers.address%type;

-- user defined exception

ex\_invalid\_id EXCEPTION;

BEGIN

IF c\_id <= 0 THEN

RAISE ex\_invalid\_id;

ELSE

SELECT name, address INTO c\_name, c\_addr

FROM customers

WHERE id = c\_id;

DBMS\_OUTPUT.PUT\_LINE ('Name: '|| c\_name);

DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);

END IF;

EXCEPTION

WHEN ex\_invalid\_id THEN

dbms\_output.put\_line('ID must be greater than zero!');

WHEN no\_data\_found THEN

dbms\_output.put\_line('No such customer!');

WHEN others THEN

dbms\_output.put\_line('Error!');

END;

/

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

Enter value for cc\_id: -6 (let's enter a value -6)

old 2: c\_id customers.id%type := &cc\_id;

new 2: c\_id customers.id%type := -6;

ID must be greater than zero!

PL/SQL procedure successfully completed.

## Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO\_DATA\_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions −

|  |  |  |  |
| --- | --- | --- | --- |
| **Exception** | **Oracle Error** | **SQLCODE** | **Description** |
| ACCESS\_INTO\_NULL | 06530 | -6530 | It is raised when a null object is automatically assigned a value. |
| CASE\_NOT\_FOUND | 06592 | -6592 | It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause. |
| COLLECTION\_IS\_NULL | 06531 | -6531 | It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray. |
| DUP\_VAL\_ON\_INDEX | 00001 | -1 | It is raised when duplicate values are attempted to be stored in a column with unique index. |
| INVALID\_CURSOR | 01001 | -1001 | It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor. |
| INVALID\_NUMBER | 01722 | -1722 | It is raised when the conversion of a character string into a number fails because the string does not represent a valid number. |
| LOGIN\_DENIED | 01017 | -1017 | It is raised when a program attempts to log on to the database with an invalid username or password. |
| NO\_DATA\_FOUND | 01403 | +100 | It is raised when a SELECT INTO statement returns no rows. |
| NOT\_LOGGED\_ON | 01012 | -1012 | It is raised when a database call is issued without being connected to the database. |
| PROGRAM\_ERROR | 06501 | -6501 | It is raised when PL/SQL has an internal problem. |
| ROWTYPE\_MISMATCH | 06504 | -6504 | It is raised when a cursor fetches value in a variable having incompatible data type. |
| SELF\_IS\_NULL | 30625 | -30625 | It is raised when a member method is invoked, but the instance of the object type was not initialized. |
| STORAGE\_ERROR | 06500 | -6500 | It is raised when PL/SQL ran out of memory or memory was corrupted. |
| TOO\_MANY\_ROWS | 01422 | -1422 | It is raised when a SELECT INTO statement returns more than one row. |
| VALUE\_ERROR | 06502 | -6502 | It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs. |
| ZERO\_DIVIDE | 01476 | 1476 | It is raised when an attempt is made to divide a number by zero. |

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;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

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, 'Kriti', 22, 'HP', 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

**5.Normalization**

# Normalization of Database

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy(repetition) and undesirable characteristics like Insertion, Update and Deletion Anomalies. It is a multi-step process that puts data into tabular form, removing duplicated data from the relation tables.

Normalization is used for mainly two purposes,

* Eliminating redundant(useless) data.
* Ensuring data dependencies make sense i.e data is logically stored.

Problems Without Normalization

If a table is not properly normalized and have data redundancy then it will not only eat up extra memory space but will also make it difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anomalies are very frequent if database is not normalized. To understand these anomalies let us take an example of a **Student** table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **rollno** | **name** | **branch** | **hod** | **office\_tel** |
| 401 | Akon | CSE | Mr. X | 53337 |
| 402 | Bkon | CSE | Mr. X | 53337 |
| 403 | Ckon | CSE | Mr. X | 53337 |
| 404 | Dkon | CSE | Mr. X | 53337 |

* In the table above, we have data of 4 Computer Sci. students. As we can see, data for the fields branch, hod(Head of Department) and office\_tel is repeated for the students who are in the same branch in the college, this is **Data Redundancy**.

#### Insertion Anomaly

Suppose for a new admission, until and unless a student opts for a branch, data of the student cannot be inserted, or else we will have to set the branch information as **NULL**.

Also, if we have to insert data of 100 students of same branch, then the branch information will be repeated for all those 100 students.

These scenarios are nothing but **Insertion anomalies**.

#### Updation Anomaly

What if Mr. X leaves the college? or is no longer the HOD of computer science department? In that case all the student records will have to be updated, and if by mistake we miss any record, it will lead to data inconsistency. This is Updation anomaly.

#### Deletion Anomaly

In our **Student** table, two different informations are kept together, Student information and Branch information. Hence, at the end of the academic year, if student records are deleted, we will also lose the branch information. This is Deletion anomaly.

Normalization Rule

Normalization rules are divided into the following normal forms:

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form

### First Normal Form (1NF)

For a table to be in the First Normal Form, it should follow the following 4 rules:

1. It should only have single(atomic) valued attributes/columns.
2. Values stored in a column should be of the same domain
3. All the columns in a table should have unique names.
4. And the order in which data is stored, does not matter.

### Second Normal Form (2NF)

For a table to be in the Second Normal Form,

1. It should be in the First Normal form.
2. And, it should not have Partial Dependency.

### Third Normal Form (3NF)

A table is said to be in the Third Normal Form when,

1. It is in the Second Normal form.
2. And, it doesn't have Transitive Dependency.

### Boyce and Codd Normal Form (BCNF)

**Boyce and Codd Normal Form** is a higher version of the Third Normal form. This form deals with certain type of anomaly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF. For a table to be in BCNF, following conditions must be satisfied:

* Relation must be in 3rd Normal Form
* and, for each functional dependency ( X → Y ), X should be a super Key.

### Fourth Normal Form (4NF)

A table is said to be in the Fourth Normal Form when,

1. It is in the Boyce-Codd Normal Form.
2. And, it doesn't have Multi-Valued Dependency.

## Rules for First Normal Form

The first normal form expects you to follow a few simple rules while designing your database, and they are:

#### Rule 1: Single Valued Attributes

Each column of your table should be single valued which means they should not contain multiple values. We will explain this with help of an example later, let's see the other rules for now.

#### Rule 2: Attribute Domain should not change

This is more of a "Common Sense" rule. In each column the values stored must be of the same kind or type.

**For example:** If you have a column dob to save date of births of a set of people, then you cannot or you must not save 'names' of some of them in that column along with 'date of birth' of others in that column. It should hold only 'date of birth' for all the records/rows.

#### Rule 3: Unique name for Attributes/Columns

This rule expects that each column in a table should have a unique name. This is to avoid confusion at the time of retrieving data or performing any other operation on the stored data.

If one or more columns have same name, then the DBMS system will be left confused

#### Rule 4: Order doesn't matters

This rule says that the order in which you store the data in your table doesn't matter.

## Time for an Example

Although all the rules are self explanatory still let's take an example where we will create a table to store student data which will have student's roll no., their name and the name of subjects they have opted for.

Here is our table, with some sample data added to it.

|  |  |  |
| --- | --- | --- |
| **roll\_no** | **name** | **Subject** |
| 101 | Akon | OS, CN |
| 103 | Ckon | Java |
| 102 | Bkon | C, C++ |

Our table already satisfies 3 rules out of the 4 rules, as all our column names are unique, we have stored data in the order we wanted to and we have not inter-mixed different type of data in columns.

But out of the 3 different students in our table, 2 have opted for more than 1 subject. And we have stored the subject names in a single column. But as per the 1st Normal form each column must contain atomic value.

### How to solve this Problem?

It's very simple, because all we have to do is break the values into atomic values.

Here is our updated table and it now satisfies the First Normal Form.

|  |  |  |
| --- | --- | --- |
| **roll\_no** | **name** | **Subject** |
| 101 | Akon | OS |
| 101 | Akon | CN |
| 103 | Ckon | Java |
| 102 | Bkon | C |
| 102 | Bkon | C++ |

By doing so, although a few values are getting repeated but values for the subject column are now atomic for each record/row.

Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

# Second Normal Form

For a table to be in the Second Normal Form, it must satisfy two conditions:

1. The table should be in the First Normal Form.
2. There should be no Partial Dependency.

## What is Dependency?

Let's take an example of a **Student** table with columns student\_id, name, reg\_no(registration number), branch and address(student's home address).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **student\_id** | **name** | **reg\_no** | **branch** | **address** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

In this table, student\_id is the primary key and will be unique for every row, hence we can use student\_id to fetch any row of data from this table

Even for a case, where student names are same, if we know the student\_id we can easily fetch the correct record.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **student\_id** | **name** | **reg\_no** | **branch** | **address** |
| 10 | Akon | 07-WY | CSE | Kerala |
| 11 | Akon | 08-WY | IT | Gujarat |

Hence we can say a **Primary Key** for a table is the column or a group of columns(composite key) which can uniquely identify each record in the table.

I can ask from branch name of student with student\_id **10**, and I can get it. Similarly, if I ask for name of student with student\_id **10** or **11**, I will get it. So all I need is student\_id and every other column **depends** on it, or can be fetched using it.

This is **Dependency** and we also call it **Functional Dependency**.

## What is Partial Dependency?

Now that we know what dependency is, we are in a better state to understand what partial dependency is.

For a simple table like Student, a single column like student\_id can uniquely identfy all the records in a table.

But this is not true all the time. So now let's extend our example to see if more than 1 column together can act as a primary key.

Let's create another table for **Subject**, which will have subject\_id and subject\_name fields and subject\_id will be the primary key.

|  |  |
| --- | --- |
| **subject\_id** | **subject\_name** |
| 1 | Java |
| 2 | C++ |
| 3 | Php |

Now we have a **Student** table with student information and another table **Subject** for storing subject information.

Let's create another table **Score**, to store the **marks** obtained by students in the respective subjects. We will also be saving **name of the teacher** who teaches that subject along with marks.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** | **teacher** |
| 1 | 10 | 1 | 70 | Java Teacher |
| 2 | 10 | 2 | 75 | C++ Teacher |
| 3 | 11 | 1 | 80 | Java Teacher |

In the score table we are saving the **student\_id** to know which student's marks are these and **subject\_id** to know for which subject the marks are for.

Together, student\_id + subject\_id forms a **Candidate Key**(learn about [Database Keys](https://www.studytonight.com/dbms/database-key.php)) for this table, which can be the **Primary key**.

Confused, How this combination can be a primary key?

See, if I ask you to get me marks of student with student\_id 10, can you get it from this table? No, because you don't know for which subject. And if I give you subject\_id, you would not know for which student. Hence we need student\_id + subject\_id to uniquely identify any row.

### But where is Partial Dependency?

Now if you look at the **Score** table, we have a column names teacher which is only dependent on the subject, for Java it's Java Teacher and for C++ it's C++ Teacher & so on.

Now as we just discussed that the primary key for this table is a composition of two columns which is student\_id & subject\_id but the teacher's name only depends on subject, hence the subject\_id, and has nothing to do with student\_id.

This is **Partial Dependency**, where an attribute in a table depends on only a part of the primary key and not on the whole key.

## How to remove Partial Dependency?

There can be many different solutions for this, but out objective is to remove teacher's name from Score table.

The simplest solution is to remove columns teacher from Score table and add it to the Subject table. Hence, the Subject table will become:

|  |  |  |
| --- | --- | --- |
| **subject\_id** | **subject\_name** | **teacher** |
| 1 | Java | Java Teacher |
| 2 | C++ | C++ Teacher |
| 3 | Php | Php Teacher |

And our Score table is now in the second normal form, with no partial dependency.

|  |  |  |  |
| --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** |
| 1 | 10 | 1 | 70 |
| 2 | 10 | 2 | 75 |
| 3 | 11 | 1 | 80 |

# Third Normal Form (3NF)

let's use the same example, where we have 3 tables, **Student**, **Subject** and **Score**.

#### Student Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **student\_id** | **name** | **reg\_no** | **branch** | **address** |
| 10 | Akon | 07-WY | CSE | Kerala |
| 11 | Akon | 08-WY | IT | Gujarat |
| 12 | Bkon | 09-WY | IT | Rajasthan |

#### Subject Table

|  |  |  |
| --- | --- | --- |
| **subject\_id** | **subject\_name** | **teacher** |
| 1 | Java | Java Teacher |
| 2 | C++ | C++ Teacher |
| 3 | Php | Php Teacher |

#### Score Table

|  |  |  |  |
| --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** |
| 1 | 10 | 1 | 70 |
| 2 | 10 | 2 | 75 |
| 3 | 11 | 1 | 80 |

In the Score table, we need to store some more information, which is the exam name and total marks, so let's add 2 more columns to the Score table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** | **exam\_name** | **total\_marks** |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

## Requirements for Third Normal Form

For a table to be in the third normal form,

1. It should be in the Second Normal form.
2. And it should not have Transitive Dependency.

### What is Transitive Dependency?

With exam\_name and total\_marks added to our Score table, it saves more data now. Primary key for our Score table is a composite key, which means it's made up of two attributes or columns → **student\_id + subject\_id**.

Our new column exam\_name depends on both student and subject. For example, a mechanical engineering student will have Workshop exam but a computer science student won't. And for some subjects you have Practical exams and for some you don't. So we can say that exam\_name is dependent on both student\_id and subject\_id.

And what about our second new column total\_marks? Does it depend on our Score table's primary key?

Well, the column total\_marks depends on exam\_name as with exam type the total score changes. For example, practicals are of less marks while theory exams are of more marks.

But, exam\_name is just another column in the score table. It is not a primary key or even a part of the primary key, and total\_marks depends on it.

This is **Transitive Dependency**. When a non-prime attribute depends on other non-prime attributes rather than depending upon the prime attributes or primary key.

### How to remove Transitive Dependency?

Again the solution is very simple. Take out the columns exam\_name and total\_marks from Score table and put them in an **Exam** table and use the exam\_id wherever required.

#### Score Table: In 3rd Normal Form

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** | **exam\_id** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

#### The new Exam table

|  |  |  |
| --- | --- | --- |
| **exam\_id** | **exam\_name** | **total\_marks** |
| 1 | Workshop | 200 |
| 2 | Mains | 70 |
| 3 | Practicals | 30 |

### Advantage of removing Transitive Dependency

The advantage of removing transitive dependency is,

* Amount of data duplication is reduced.
* Data integrity achieved.

# Boyce-Codd Normal Form (BCNF)

Boyce-Codd Normal Form or BCNF is an extension to the [third normal form](https://www.studytonight.com/dbms/third-normal-form.php), and is also known as 3.5 Normal Form.

## Rules for BCNF

For a table to satisfy the Boyce-Codd Normal Form, it should satisfy the following two conditions:

1. It should be in the **Third Normal Form**.
2. And, for any dependency A → B, A should be a **super key**.

The second point sounds a bit tricky, right? In simple words, it means, that for a dependency A → B, A cannot be a **non-prime attribute**, if B is a **prime attribute**.

## Time for an Example

Below we have a college enrolment table with columns student\_id, subject and professor.

|  |  |  |
| --- | --- | --- |
| **student\_id** | **subject** | **professor** |
| 101 | Java | P.Java |
| 101 | C++ | P.Cpp |
| 102 | Java | P.Java2 |
| 103 | C# | P.Chash |
| 104 | Java | P.Java |

As you can see, we have also added some sample data to the table.

In the table above:

* One student can enrol for multiple subjects. For example, student with **student\_id** 101, has opted for subjects - Java & C++
* For each subject, a professor is assigned to the student.
* And, there can be multiple professors teaching one subject like we have for Java.

What do you think should be the **Primary Key**?

Well, in the table above student\_id, subject together form the primary key, because using student\_id and subject, we can find all the columns of the table.

One more important point to note here is, one professor teaches only one subject, but one subject may have two different professors.

Hence, there is a dependency between subject and professor here, where subject depends on the professor name.

This table satisfies the **1st Normal form** because all the values are atomic, column names are unique and all the values stored in a particular column are of same domain.

This table also satisfies the **2nd Normal Form** as their is no **Partial Dependency**.

And, there is no **Transitive Dependency**, hence the table also satisfies the **3rd Normal Form**.

But this table is not in **Boyce-Codd Normal Form**.

### Why this table is not in BCNF?

In the table above, student\_id, subject form primary key, which means subject column is a **prime attribute**.

But, there is one more dependency, professor → subject.

And while subject is a prime attribute, professor is a **non-prime attribute**, which is not allowed by BCNF.

### How to satisfy BCNF?

To make this relation(table) satisfy BCNF, we will decompose this table into two tables, **student** table and **professor** table.

Below we have the structure for both the tables.

**Student Table**

|  |  |
| --- | --- |
| **student\_id** | **p\_id** |
| 101 | 1 |
| 101 | 2 |
| and so on... | |

And, **Professor Table**

|  |  |  |
| --- | --- | --- |
| **p\_id** | **professor** | **subject** |
| 1 | P.Java | Java |
| 2 | P.Cpp | C++ |
| and so on... | | |

And now, this relation satisfy Boyce-Codd Normal Form. In the next tutorial we will learn about the **Fourth Normal Form**.

# Functional Dependency

## Definition - *Functional Dependency*

Functional dependency is a relationship that exists when one attribute uniquely determines another attribute.

If R is a relation with attributes X and Y, a functional dependency between the attributes is represented as X->Y, which specifies Y is functionally dependent on X. Here X is a determinant set and Y is a dependent attribute. Each value of X is associated with precisely one Y value.

Functional dependency in a database serves as a constraint between two sets of attributes. Defining functional dependency is an important part of relational database design and contributes to aspect normalization.

## *Functional Dependency*

A functional dependency is trivial if Y is a subset of X. In a table with the attributes of employee name and Social Security number (SSN), employee name is functionally dependent on SSN because the SSN is unique for individual names. An SSN identifies the employee specifically, but an employee name cannot distinguish the SSN because more than one employee could have the same name.

Functional dependency defines Boyce-Codd normal form and third normal form. This preserves dependency between attributes, eliminating the repetition of information. Functional dependency is related to a candidate key, which uniquely identifies a tuple and determines the value of all other attributes in the relation. In some cases, functionally dependent sets are irreducible if:

* The right-hand set of functional dependency holds only one attribute
* Th e left-hand set of functional dependency cannot be reduced, since this may change the entire content of the set
* Reducing any of the existing functional dependency might change the content of the set

An important property of a functional dependency is Armstrong’s axiom, which is used in database normalization. In a relation, R, with three attributes (X, Y, Z) Armstrong’s axiom holds true if the following conditions are satisfied:

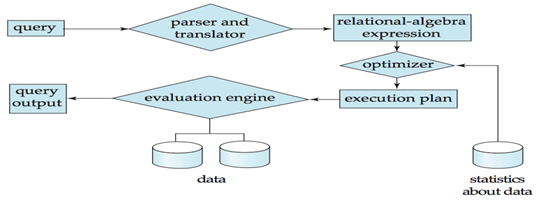
* Axiom of Transivity: If X->Y and Y->Z, then X->Z
* Axiom of Reflexivity (Subset Property): If Y is a subset of X, then X->Y
* Axiom of Augmentation: If X->Y, then XZ->YZ

**CHAPTER 6**

**Query processing and Transaction processing**

# Introduction(Query processing)

The main goal of creating a database is to store the related data at one place, access and manipulate them as and when it is required by the user. Accessing and manipulating the data should be done efficiently i.e.; it should be accessed easily and quickly.  
  
But a database is a system and the users are either another system or application or a person. The user can request the data in a language that he understands. But DBMS has its own language (SQL) which it understands. Hence the users are asked to query the database in its language – SQL. This SQL is a high level language created to build a bridge between user and DBMS for their communication. But the underlying systems in the DBMS will not understand SQL. There has to be some low level language which these systems can understand. Usually any query written in SQL is converted into low level language using relational algebra which system can understand. But it will be difficult for any user to directly write relational algebra kind of queries. It requires thorough knowledge of it.  
  
Hence what DBMS does is it asks its users to write query in SQL. It verifies the code written by the user and then converts them into low level languages. It then selects the best execution path and executes the query and gets the data from internal memory. All these processes are together known as query processing.



Above diagram depicts how a query is processed in the database to show the result. When a query is submitted to the database, it is received by the query compiler.

It then scans the query and **divides it into individual tokens**. Once the tokens are generated, they are **verified** for their correctness by the **parser.**

Then the tokenized queries are transformed into different possible **relational expressions**, relational trees and relational graphs (Query Plans).

Query optimizer then picks them to identify the **best query plan** to process. It checks in the system catalog for the constraints and indexes and decides the best query plan.

It generates different execution plans for the **query plan**. The query execution plan then decides the best and **optimized execution** plan for execution.

The command processor then uses this execution plan to **retrieve** the data from the **database and returns the result.**

There are four phases in a typical query processing.

* Parsing and Translation
* Query Optimization
* Evaluation or query code generation

**1. Parsing and translation**

* Translate the query into its internal form. This is then translated into relational algebra.
* Parser checks syntax, verifies relation.

**2. Optimization**

* SQL is a very high level language:
  + The users specify what to search for- not how the search is actually done
  + The algorithms are chosen automatically by the DBMS.
* For a given SQL query there may be many possible execution plans.
* Amongst all equivalent plans choose the one with lowest cost.
* Cost is estimated using statistical information from the database catalog.

**3. Evaluation**

* The query evaluation engine takes a query evaluation plan, executes that plan and returns the answer to that query.

**Measures of Query Cost Measures of Query Cost !**

# Measures of Query Cost

There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their (estimated) cost, and choose the best plan. To do so, we must estimate the cost of individual operations, and combine them to get the cost of a query evaluation plan. Thus, as we study evaluation algorithms for each operation later in this chapter, we also outline how to estimate the cost of the operation.

The cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in a distributed or parallel database system, the cost of communication.

In large database systems, the cost to access data from disk is usually the most important cost, since disk accesses are slow compared to in-memory operations. Moreover, CPU speeds have been improving much faster than have disk speeds. Thus, it is likely that the time spent in disk activity will continue to dominate the total time to execute a query. The CPU time taken for a task is harder to estimate since it depends on low-level details of the execution code. Although real-life query optimizers do take CPU costs into account, for simplicity in this book we ignore CPU costs and use only disk-access costs to measure the cost of a query-evaluation plan.

We use the number of block transfers from disk and the number of disk seeks to estimate the cost of a query-evaluation plan. If the disk subsystem takes an average of tT seconds to transfer a block of data, and has an average block-access time (disk seek time plus rotational latency) of tS seconds, then an operation that transfers b blocks and performs S seeks would take b ∗ tT + S ∗ tS seconds. The values of tT and tS must be calibrated for the disk system used, but typical values for high-end disks today would be tS = 4 milliseconds and tT = 0.1 milliseconds, assuming a 4-kilobyte block size and a transfer rate of 40 megabytes per second.

tT – time to transfer one block

**tS**– time for one seek

Cost for b block transfers plus S seeks

b \* tT + S \* tS

We ignore CPU costs for simplicity

Real systems do take CPU cost into account

We do not include cost to writing output to disk in our cost formulae

**What is a Database Transaction?**

A transaction is a logical unit of processing in a DBMS which entails one or more database access operation. In a nutshell, database transactions represent real-world events of any enterprise.

All types of database access operation which are held between the beginning and end transaction statements are considered as a single logical transaction. During the transaction the database is inconsistent. Only once the database is committed the state is changed from one consistent state to another.

**Facts about Database Transactions**

* A transaction is a program unit whose execution may or may not change the contents of a database.
* The transaction is executed as a single unit
* If the database operations do not update the database but only retrieve data, this type of transaction is called a read-only transaction.
* A successful transaction can change the database from one CONSISTENT STATE to another
* DBMS transactions must be atomic, consistent, isolated and durable
* If the database were in an inconsistent state before a transaction, it would remain in the inconsistent state after the transaction.

**Why do you need concurrency in Transactions?**

A database is a shared resource accessed. It is used by many users and processes concurrently. For example, the banking system, railway, and air reservations systems, stock market monitoring, supermarket inventory, and checkouts, etc.

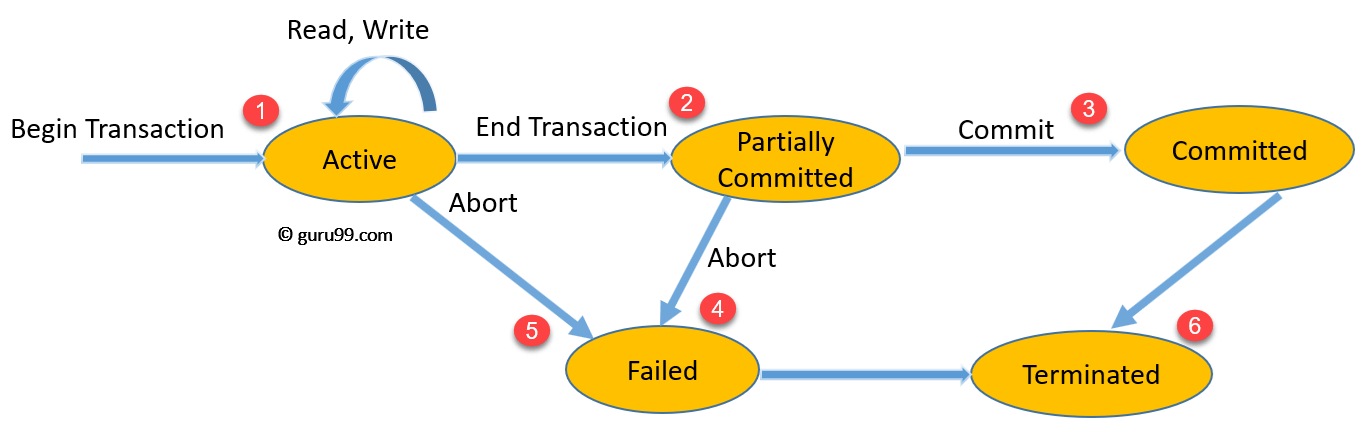
Not managing concurrent access may create issues like:

* Hardware failure and system crashes
* Concurrent execution of the same transaction, deadlock, or slow performance

**States of Transactions**

The various states of a Database Transaction are listed below

|  |  |
| --- | --- |
| **State** | **Transaction types** |
| Active State | A transaction enters into an active state when the execution process begins. During this state read or write operations can be performed. |
| Partially Committed | A transaction goes into the partially committed state after the end of a transaction. |
| Committed State | When the transaction is committed to state, it has already completed its execution successfully. Moreover, all of its changes are recorded to the database permanently. |
| Failed State | A transaction considers failed when any one of the checks fails or if the transaction is aborted while it is in the active state. |
| Terminated State | State of transaction reaches terminated state when certain transactions which are leaving the system can't be restarted. |



State Transition Diagram for a Database Transaction

**What are ACID Properties?**

For maintaining the integrity of data, the DBMS system you have to ensure ACID properties. ACID stands for **A**tomicity, **C**onsistency, **I**solation, and **D**urability.

* **Atomicity:** A transaction is a single unit of operation. You either execute it entirely or do not execute it at all. There cannot be partial execution.
* **Consistency:** Once the transaction is executed, it should move from one consistent state to another.
* **Isolation:**Transaction should be executed in isolation from other transactions (no Locks). During concurrent transaction execution, intermediate transaction results from simultaneously executed transactions should not be made available to each other. (Level 0,1,2,3)
* **Durability:** **·**After successful completion of a transaction, the changes in the database should persist. Even in the case of system failures.

# Concurrent Execution in Transaction | DBMS |

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data.

Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run serially—that is, one at a time, each starting only after the previous one has completed.

However, there are two good reasons for allowing concurrency:

### Improved throughput and resource utilization:

* A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU.
* The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel.
* While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction.
* All of this increases the throughput of the system—that is, the number of transactions executed in a given amount of time.
* Correspondingly, the processor and disk utilization also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

### Reduced waiting time:

* There may be a mix of transactions running on a system, some short and some long.
* If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction.
* If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them.
* Concurrent execution reduces the unpredictable delays in running transactions.
* Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.

# DBMS Serializability

BY CHAITANYA SINGH | FILED UNDER: [DBMS](https://beginnersbook.com/category/dbms/)

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. Serializability is a concept that helps us to check which [schedules](https://beginnersbook.com/2018/12/dbms-schedules/) are serializable. A serializable schedule is the one that always leaves the database in consistent state.

## What is a serializable schedule?

A serializable schedule always leaves the database in consistent state. A [serial schedule](https://beginnersbook.com/2018/12/dbms-schedules/) is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution. However a non-serial schedule needs to be checked for Serializability.

A non-serial schedule of n number of transactions is said to be serializable schedule, if it is equivalent to the serial schedule of those n transactions. A serial schedule doesn’t allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

## Types of Serializability

There are two types of Serializability.

1. [Conflict Serializability](https://beginnersbook.com/2018/12/dbms-conflict-serializability/)  
2. [View Serializability](https://beginnersbook.com/2018/12/dbms-view-serializability/)

# Conflict Serializability

A schedule is called conflict serializable if we can convert it into a serial schedule after swapping its non-conflicting operations.

### Conflicting operations

Two operations are said to be in conflict, if they satisfy all the following three conditions:

1. Both the operations should belong to different transactions.  
2. Both the operations are working on same data item.  
3. At least one of the operation is a write operation.

### Example of Conflict Serializability

Lets consider this schedule:

T1 T2

----- ------

R(A)

R(B)

R(A)

R(B)

W(B)

W(A)

To convert this schedule into a serial schedule we must have to swap the R(A) operation of transaction T2 with the W(A) operation of transaction T1. However we cannot swap these two operations because they are conflicting operations, thus we can say that this given schedule is **not Conflict Serializable**.

Lets take another example:

T1 T2

----- ------

R(A)

R(A)

R(B)

W(B)

R(B)

W(A)

Lets **swap non-conflicting operations**:

After swapping R(A) of T1 and R(A) of T2 we get:

T1 T2

----- ------

R(A)

R(A)

R(B)

W(B)

R(B)

W(A)

After swapping R(A) of T1 and R(B) of T2 we get:

T1 T2

----- ------

R(A)

R(B)

R(A)

W(B)

R(B)

W(A)

After swapping R(A) of T1 and W(B) of T2 we get:

T1 T2

----- ------

R(A)

R(B)

W(B)

R(A)

R(B)

W(A)

We finally got a serial schedule after swapping all the non-conflicting operations so we can say that the given schedule is **Conflict Serializable**.

# View Serializability

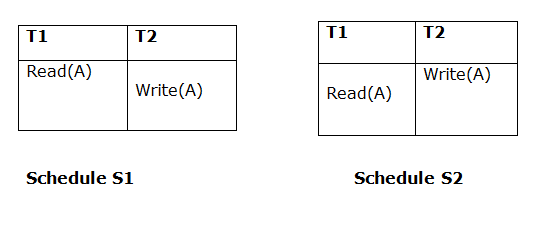
* A schedule will view serializable if it is view equivalent to a serial schedule.
* If a schedule is conflict serializable, then it will be view serializable.
* The view serializable which does not conflict serializable contains blind writes.

## View Equivalent

Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

### 1. Initial Read

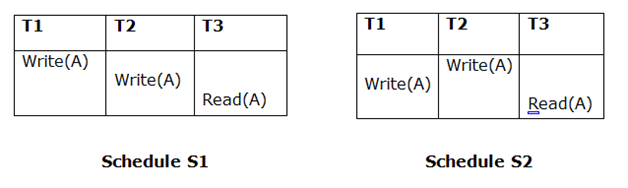
An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.



Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

### 2. Updated Read

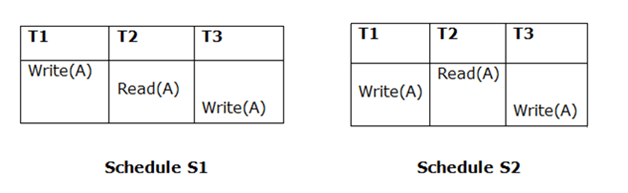
In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.



Above two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

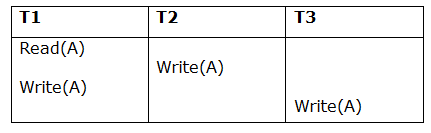
### 3. Final Write

A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.



Above two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

**Example:**

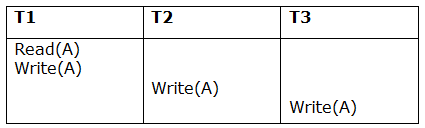


**Schedule S**

With 3 transactions, the total number of possible schedule

1. = 3! = 6
2. S1 = <T1 T2 T3>
3. S2 = <T1 T3 T2>
4. S3 = <T2 T3 T1>
5. S4 = <T2 T1 T3>
6. S5 = <T3 T1 T2>
7. S6 = <T3 T2 T1>

**Taking first schedule S1:**



**Schedule S1**

**Step 1:** final updation on data items

In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.

**Step 2:** Initial Read

The initial read operation in S is done by T1 and in S1, it is also done by T1.

**Step 3:** Final Write

The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.

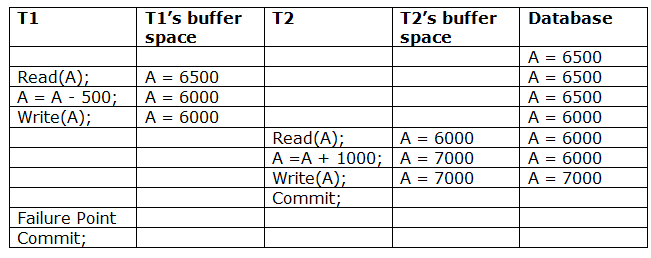
The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.

**Hence, view equivalent serial schedule is:**

1. T1    →      T2    →    T3

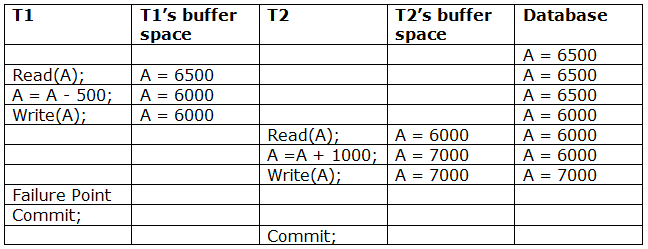
# Recoverability of Schedule

Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.



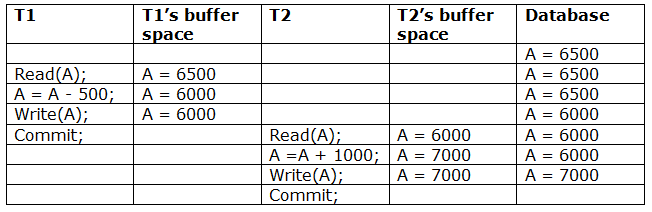
The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

**Irrecoverable schedule:** The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.



The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

**Recoverable with cascading rollback:** The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti. Commit of Tj is delayed till commit of Ti.



The above Table 3 shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

# 

# Chapter 7

# Concurrency Control

* In the concurrency control, the multiple transactions can be executed simultaneously.
* It may affect the transaction result. It is highly important to maintain the order of execution of those transactions.

## Problems of concurrency control

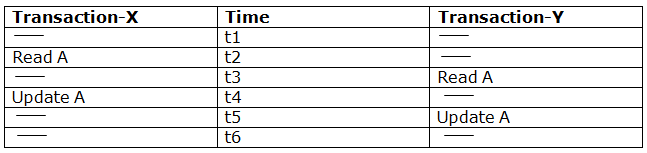
Several problems can occur when concurrent transactions are executed in an uncontrolled manner. Following are the three problems in concurrency control.

1. Lost updates
2. Dirty read
3. Unrepeatable read

### 1. Lost update problem

* When two transactions that access the same database items contain their operations in a way that makes the value of some database item incorrect, then the lost update problem occurs.
* If two transactions T1 and T2 read a record and then update it, then the effect of updating of the first record will be overwritten by the second update.

**Example:**



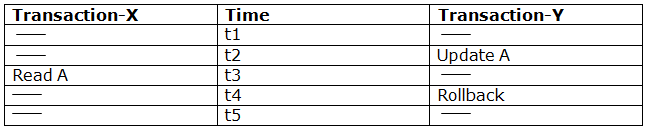
**Here,**

* At time t2, transaction-X reads A's value.
* At time t3, Transaction-Y reads A's value.
* At time t4, Transactions-X writes A's value on the basis of the value seen at time t2.
* At time t5, Transactions-Y writes A's value on the basis of the value seen at time t3.
* So at time T5, the update of Transaction-X is lost because Transaction y overwrites it without looking at its current value.
* Such type of problem is known as Lost Update Problem as update made by one transaction is lost here.

### 2. Dirty Read

* The dirty read occurs in the case when one transaction updates an item of the database, and then the transaction fails for some reason. The updated database item is accessed by another transaction before it is changed back to the original value.
* A transaction T1 updates a record which is read by T2. If T1 aborts then T2 now has values which have never formed part of the stable database.

**Example:**



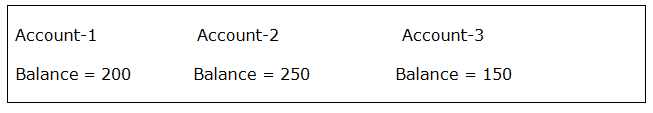
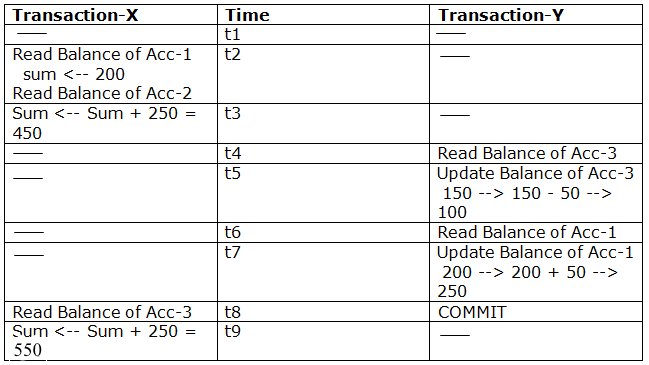
* At time t2, transaction-Y writes A's value.
* At time t3, Transaction-X reads A's value.
* At time t4, Transactions-Y rollbacks. So, it changes A's value back to that of prior to t1.
* So, Transaction-X now contains a value which has never become part of the stable database.
* Such type of problem is known as Dirty Read Problem, as one transaction reads a dirty value which has not been committed.

### 3. Inconsistent Retrievals Problem

* Inconsistent Retrievals Problem is also known as unrepeatable read. When a transaction calculates some group function over a set of data while the other transactions are updating the data, then the Inconsistent Retrievals Problem occurs.
* A transaction T1 reads a record and then does some other processing during which the transaction T2 updates the record. Now when the transaction T1 reads the record, then the new value will be inconsistent with the previous value.

**Example:**

Suppose two transactions operate on three accounts.

* Transaction-X is doing the sum of all balance while transaction-Y is transferring an amount 50 from Account-1 to Account-3.
* Here, transaction-X produces the result of 550 which is incorrect. If we write this produced result in the database, the database will become an inconsistent state because the actual sum is 600.
* Here, transaction-X has seen an inconsistent state of the database.

## Concurrency Control Protocol

Concurrency control protocols ensure atomicity, isolation, and serializability of concurrent transactions. The concurrency control protocol can be divided into three categories:

1. Lock based protocol
2. Time-stamp protocol
3. Validation based protocol

# Lock-Based Protocol

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

**1. Shared lock:**

* It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
* It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

**2. Exclusive lock:**

* In the exclusive lock, the data item can be both reads as well as written by the transaction.
* This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

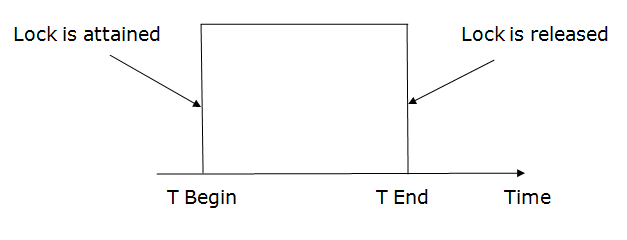
## There are four types of lock protocols available:

### 1. Simplistic lock protocol

It is the simplest way of locking the data while transaction. Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it. It will unlock the data item after completing the transaction.

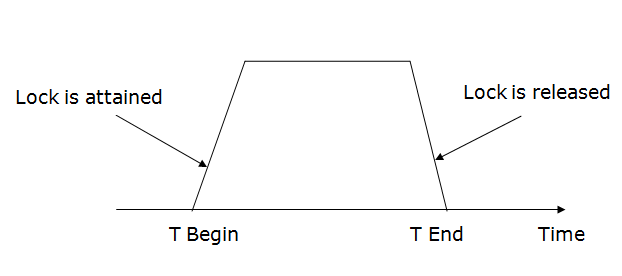
### 2. Pre-claiming Lock Protocol

* Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
* Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
* If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
* If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.



### 3. Two-phase locking (2PL)

* The two-phase locking protocol divides the execution phase of the transaction into three parts.
* In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
* In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
* In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.



There are two phases of 2PL:

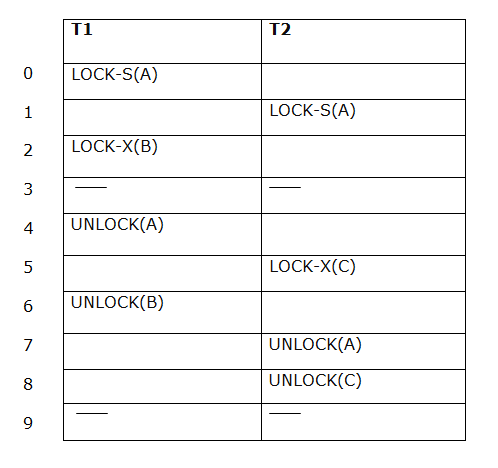
**Growing phase:** In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

**Shrinking phase:** In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

1. Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
2. Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

**Example:**



The following way shows how unlocking and locking work with 2-PL.

**Transaction T1:**

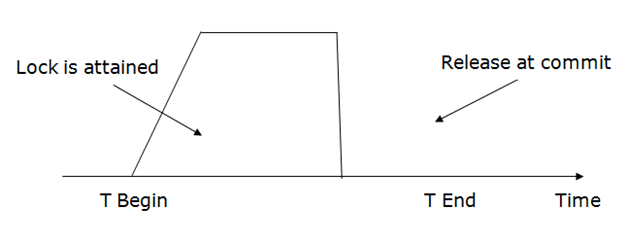
* **Growing phase:** from step 1-3
* **Shrinking phase:** from step 5-7
* **Lock point:** at 3

**Transaction T2:**

* **Growing phase:** from step 2-6
* **Shrinking phase:** from step 8-9
* **Lock point:** at 6

### 4. Strict Two-phase locking (Strict-2PL)

* The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
* The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
* Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
* Strict-2PL protocol does not have shrinking phase of lock release.



It does not have cascading abort as 2PL does.

# Timestamp Ordering Protocol

* The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
* The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
* The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.
* Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.
* The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

**Basic Timestamp ordering protocol works as follows:**

1. Check the following condition whenever a transaction Ti issues a **Read (X)** operation:

* If W\_TS(X) >TS(Ti) then the operation is rejected.
* If W\_TS(X) <= TS(Ti) then the operation is executed.
* Timestamps of all the data items are updated.

2. Check the following condition whenever a transaction Ti issues a **Write(X)** operation:

* If TS(Ti) < R\_TS(X) then the operation is rejected.
* If TS(Ti) < W\_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.

**Where,**

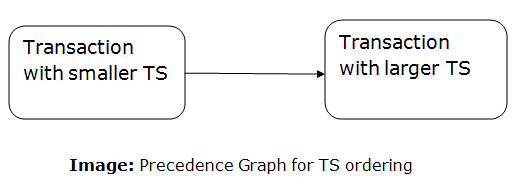
**TS(TI)** denotes the timestamp of the transaction Ti.

**R\_TS(X)** denotes the Read time-stamp of data-item X.

**W\_TS(X)** denotes the Write time-stamp of data-item X.

## Advantages and Disadvantages of TO protocol:

* TO protocol ensures serializability since the precedence graph is as follows:



* TS protocol ensures freedom from deadlock that means no transaction ever waits.
* But the schedule may not be recoverable and may not even be cascade- free.

# Validation Based Protocol

Validation phase is also known as optimistic concurrency control technique. In the validation based protocol, the transaction is executed in the following three phases:

1. **Read phase:** In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.
2. **Validation phase:** In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.
3. **Write phase:** If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

Here each phase has the following different timestamps:

**Start(Ti):** It contains the time when Ti started its execution.

**Validation (Ti):** It contains the time when Ti finishes its read phase and starts its validation phase.

**Finish(Ti):** It contains the time when Ti finishes its write phase.

* This protocol is used to determine the time stamp for the transaction for serialization using the time stamp of the validation phase, as it is the actual phase which determines if the transaction will commit or rollback.
* Hence TS(T) = validation(T).
* The serializability is determined during the validation process. It can't be decided in advance.
* While executing the transaction, it ensures a greater degree of concurrency and also less number of conflicts.
* Thus it contains transactions which have less number of rollbacks.

Chapter 8

# Failure Classification

To find that where the problem has occurred, we generalize a failure into the following categories:

1. Transaction failure
2. System crash
3. Disk failure

### 1. Transaction failure

The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for a transaction failure could be -

* 1. **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.
  2. **Syntax error:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. **For example,** The system aborts an active transaction, in case of deadlock or resource unavailability.

### 2. System Crash

* 1. System failure can occur due to power failure or other hardware or software failure. **Example:** Operating system error.

**Fail-stop assumption:** In the system crash, non-volatile storage is assumed not to be corrupted.

### 3. Disk Failure

* 1. It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.
  2. Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

Storage Structure

the storage structure can be divided into two categories −

* **Volatile storage** − As the name suggests, a volatile storage cannot survive system crashes. Volatile storage devices are placed very close to the CPU; normally they are embedded onto the chipset itself. For example, main memory and cache memory are examples of volatile storage. They are fast but can store only a small amount of information.
* **Non-volatile storage** − These memories are made to survive system crashes. They are huge in data storage capacity, but slower in accessibility. Examples may include hard-disks, magnetic tapes, flash memory, and non-volatile (battery backed up) RAM.

# Log-Based Recovery

* The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.
* If any operation is performed on the database, then it will be recorded in the log.
* But the process of storing the logs should be done before the actual transaction is applied in the database.

Let's assume there is a transaction to modify the City of a student. The following logs are written for this transaction.

* When the transaction is initiated, then it writes 'start' log.
  1. <Tn, Start>
* When the transaction modifies the City from 'Noida' to 'Bangalore', then another log is written to the file.
  1. <Tn, City, 'Noida', 'Bangalore' >
* When the transaction is finished, then it writes another log to indicate the end of the transaction.
  1. <Tn, Commit>

There are two approaches to modify the database:

### 1. Deferred database modification:

* The deferred modification technique occurs if the transaction does not modify the database until it has committed.
* In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.

### 2. Immediate database modification:

* The Immediate modification technique occurs if database modification occurs while the transaction is still active.
* In this technique, the database is modified immediately after every operation. It follows an actual database modification.

## Recovery using Log records

When the system is crashed, then the system consults the log to find which transactions need to be undone and which need to be redone.

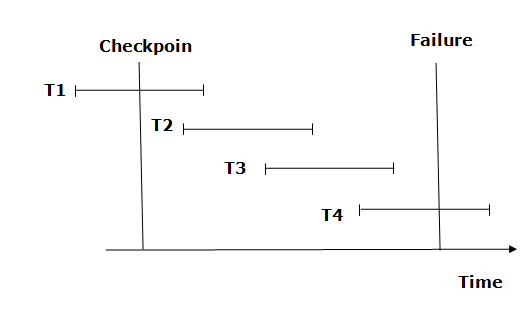
1. If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.
2. If log contains record<Tn, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone.

# Checkpoint

* The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
* The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.
* When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.
* The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

## Recovery using Checkpoint

In the following manner, a recovery system recovers the database from this failure:



* The recovery system reads log files from the end to start. It reads log files from T4 to T1.
* Recovery system maintains two lists, a redo-list, and an undo-list.
* The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.
* **For example:** In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.
* The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.
* **For example:** Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.