DATABASE MANAGEMENT SYSTEM (DBMS)

DBMS stands for **D**ata**b**ase **M**anagement **S**ystem. We can break it like this DBMS = Database + Management System. Database is a collection of data and Management System is a set of programs to store and retrieve those data. Based on this we can **define DBMS** like this: DBMS is a collection of inter-related data and set of programs to store & access those data in an easy and effective manner.

**What is the need of DBMS?**

Database systems are basically developed for large amount of data. When dealing with huge amount of data, there are two things that require optimization: **Storage of data** and **retrieval of data**.

**Storage:** According to the principles of database systems, the data is stored in such a way that it acquires lot less space as the redundant data (duplicate data) has been removed before storage. Let’s take a layman example to understand this:  
In a banking system, suppose a customer is having two accounts, one is saving account and another is salary account. Let’s say bank stores saving account data at one place and salary account data at another place, in that case if the customer information such as customer name, address etc. are stored at both places then this is just a wastage of storage (redundancy/ duplication of data), to organize the data in a better way the information should be stored at one place and both the accounts should be linked to that information somehow. The same thing we achieve in DBMS.

**Fast Retrieval of data**: Along with storing the data in an optimized and systematic manner, it is also important that we retrieve the data quickly when needed. Database systems ensure that the data is retrieved as quickly as possible.

**Purpose of Database Systems**

The main purpose of database systems is to manage the data. Consider a university that keeps the data of students, teachers, courses, books etc. To manage this data we need to store this data somewhere where we can add new data, delete unused data, update outdated data, retrieve data, to perform these operations on data we need a Database management system that allows us to store the data in such a way so that all these operations can be performed on the data efficiently.

Applications where we use Database Management Systems are:

* **Telecom**: There is a database to keeps track of the information regarding calls made, network usage, customer details etc. Without the database systems it is hard to maintain that huge amount of data that keeps updating every millisecond.
* **Industry**: Where it is a manufacturing unit, warehouse or distribution centre, each one needs a database to keep the records of ins and outs. For example distribution centre should keep a track of the product units that supplied into the centre as well as the products that got delivered out from the distribution centre on each day; this is where DBMS comes into picture.
* **Banking System**: For storing customer info, tracking day to day credit and debit transactions, generating bank statements etc. All this work has been done with the help of Database management systems.
* **Sales**: To store customer information, production information and invoice details.
* **Airlines**: To travel though airlines, we make early reservations, this reservation information along with flight schedule is stored in database.
* **Education sector**: Database systems are frequently used in schools and colleges to store and retrieve the data regarding student details, staff details, course details, exam details, payroll data, attendance details, fees details etc. There is a hell lot amount of inter-related data that needs to be stored and retrieved in an efficient manner.
* **Online shopping**: We must be aware of the online shopping websites such as Amazon, Flipkart etc. These sites store the product information, our addresses and preferences, credit details and provide us the relevant list of products based on our query. All this involves a Database management system.

**DBMS Vs File System**

We will discuss what a file processing system is and how Database management systems are better than file processing systems.

**Drawbacks of File system**

* **Data redundancy:** Data redundancy refers to the duplication of data, lets say we are managing the data of a college where a student is enrolled for two courses, the same student details in such case will be stored twice, which will take more storage than needed. Data redundancy often leads to higher storage costs and poor access time.
* **Data inconsistency:** Data redundancy leads to data inconsistency, let’s take the same example that we have taken above, a student is enrolled for two courses and we have student address stored twice, now lets say student requests to change his address, if the address is changed at one place and not on all the records then this can lead to data inconsistency.
* **Data Isolation:** Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.
* **Dependency on application programs:** Changing files would lead to change in application programs.
* **Atomicity issues:** Atomicity of a transaction refers to “All or nothing”, which means either all the operations in a transaction executes or none.

For example: Let’s say Selvam transfers 100$ to Nelson’s account. This transaction consists multiple operations such as debit 100$ from Selvam’s account, credit 100$ to Nelson’s account. Like any other device, a computer system can fail lets say it fails after first operation then in that case Selvam’s account would have been debited by 100$ but the amount was not credited to Nelson’s account, in such case the rollback of operation should occur to maintain the atomicity of transaction. It is **difficult to achieve atomicity in file processing systems**.

* **Data Security:** Data should be secured from unauthorised access, for example a student in a college should not be able to see the payroll details of the teachers, such kind of security constraints are difficult to apply in file processing systems.

**Advantage of DBMS over file system**

There are several advantages of Database management system over file system. Few of them are as follows:

* **No redundant data**: Redundancy removed by data [normalization](https://beginnersbook.com/2015/05/normalization-in-dbms/). No data duplication saves storage and improves access time.
* **Data Consistency and Integrity**: The root cause of data inconsistency is data redundancy, since data normalization takes care of the data redundancy, data inconsistency also been taken care of as part of it
* **Data Security**: It is easier to apply access constraints in database systems so that only authorized user is able to access the data. Each user has a different set of access thus data is secured from the issues such as identity theft, data leaks and misuse of data.
* **Privacy**: Limited access means privacy of data.
* **Easy access to data** – Database systems manages data in such a way so that the data is easily accessible with fast response times.
* **Easy recovery**: Since database systems keeps the backup of data, it is easier to do a full recovery of data in case of a failure.
* **Flexible**: Database systems are more flexible than file processing systems.

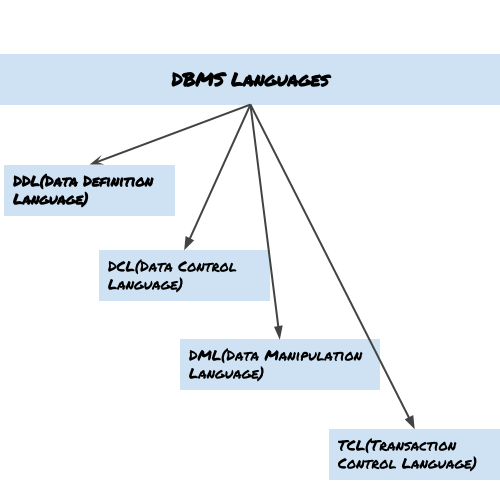
**Disadvantages of DBMS**:

* DBMS implementation cost is high compared to the file system
* Complexity: Database systems are complex to understand
* Performance: Database systems are generic, making them suitable for various applications. However this feature affect their performance for some applications

DBMS languages

Database languages are used to read, update and store data in a database. There are several such languages that can be used for this purpose; one of them is SQL (Structured Query Language).

## Types of DBMS languages:



### Data Definition Language (DDL)

DDL is used for specifying the database schema. It is used for creating tables, schema, indexes, constraints etc. in database. Lets see the operations that we can perform on database using DDL:

* To create the database instance – CREATE
* To alter the structure of database – **ALTER**
* To drop database instances – DROP
* To delete tables in a database instance – **TRUNCATE**
* To rename database instances – **RENAME**
* To drop objects from database such as tables – **DROP**
* To Comment – **Comment**

All of these commands either defines or update the database schema that’s why they come under Data Definition language.

### Data Manipulation Language (DML)

DML is used for accessing and manipulating data in a database. The following operations on database comes under DML:

* To read records from table(s) – SELECT
* To insert record(s) into the table(s) – **INSERT**
* Update the data in table(s) – UPDATE
* Delete all the records from the table – DELETE

### Data Control language (DCL)

DCL is used for granting and revoking user access on a database –

* To grant access to user – GRANT
* To revoke access from user – REVOKE

**In practical data definition language, data manipulation language and data control languages are not separate language, rather they are the parts of a single database language such as SQL.**

### Transaction Control Language(TCL)

The changes in the database that we made using DML commands are either performed or rollbacked using TCL.

* To persist the changes made by DML commands in database – COMMIT
* To rollback the changes made to the database – ROLLBACK

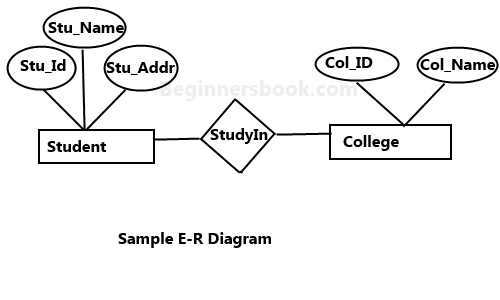
**Entity–relationship model (ER model)**

* An **Entity–relationship model (ER model)** describes the structure of a database with the help of a diagram, which is known as **Entity Relationship Diagram (ER Diagram)**. An ER model is a design or blueprint of a database that can later be implemented as a database. The main components of E-R model are: entity set and relationship set.

## What is an Entity Relationship Diagram (ER Diagram)?

* An ER diagram shows the relationship among entity sets. An entity set is a group of similar entities and these entities can have attributes. In terms of DBMS, an entity is a table or attribute of a table in database, so by showing relationship among tables and their attributes, ER diagram shows the complete logical structure of a database. Lets have a look at a simple ER diagram to understand this concept.

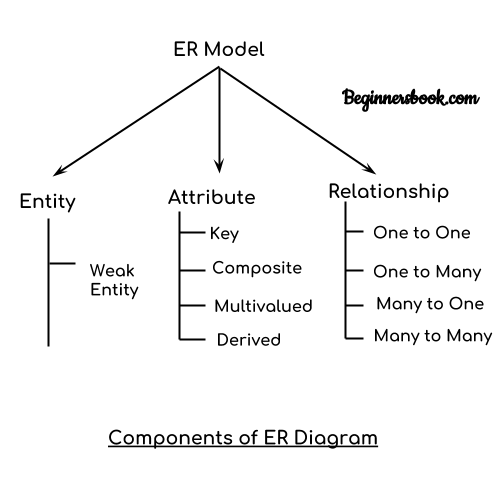
## A simple ER Diagram:

In the following diagram we have two entities Student and College and their relationship. The relationship between Student and College is many to one as a college can have many students however a student cannot study in multiple colleges at the same time. Student entity has attributes such as Stu\_Id, Stu\_Name & Stu\_Addr and College entity has attributes such as Col\_ID & Col\_Name.  


Here are the geometric shapes and their meaning in an E-R Diagram. We will discuss these terms in detail in the next section(Components of a ER Diagram) of this guide so don’t worry too much about these terms now, just go through them once.

**Rectangle**: Represents Entity sets.  
**Ellipses**: Attributes  
**Diamonds**: Relationship Set  
**Lines**: They link attributes to Entity Sets and Entity sets to Relationship Set  
**Double Ellipses:** Multivalued Attributes  
**Dashed Ellipses**: Derived Attributes  
**Double Rectangles**: Weak Entity Sets  
**Double Lines**: Total participation of an entity in a relationship set

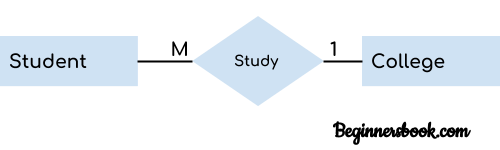
## Components of a ER Diagram



As shown in the above diagram, an ER diagram has three main components:  
1. Entity  
2. Attribute  
3. Relationship

### 1. Entity

An entity is an object or component of data. An entity is represented as rectangle in an ER diagram.  
For example: In the following ER diagram we have two entities Student and College and these two entities have many to one relationship as many students study in a single college. We will read more about relationships later, for now focus on entities.



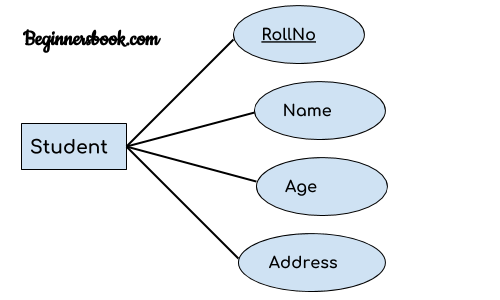
**Weak Entity:**  
An entity that cannot be uniquely identified by its own attributes and relies on the relationship with other entity is called weak entity. The weak entity is represented by a double rectangle. For example – a bank account cannot be uniquely identified without knowing the bank to which the account belongs, so bank account is a weak entity.  


### 2. Attribute

An attribute describes the property of an entity. An attribute is represented as Oval in an ER diagram. There are four types of attributes:

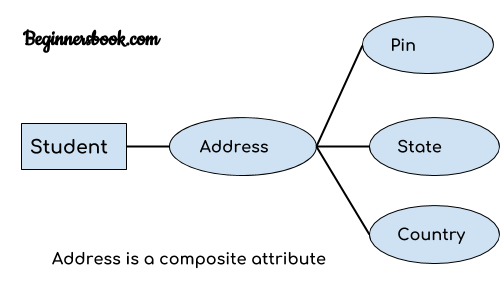
1. Key attribute  
2. Composite attribute  
3. Multivalued attribute  
4. Derived attribute

#### 1. Key attribute:

A key attribute can uniquely identify an entity from an entity set. For example, student roll number can uniquely identify a student from a set of students. Key attribute is represented by oval same as other attributes however the **text of key attribute is underlined**.  


#### 2. Composite attribute:

An attribute that is a combination of other attributes is known as composite attribute. For example, In student entity, the student address is a composite attribute as an address is composed of other attributes such as pin code, state, country.

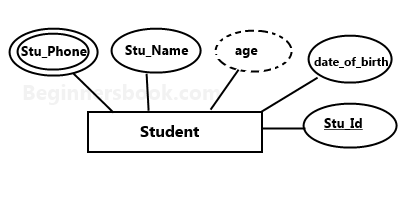


#### 3. Multivalued attribute:

An attribute that can hold multiple values is known as multivalued attribute. It is represented with **double ovals** in an ER Diagram. For example – A person can have more than one phone numbers so the phone number attribute is multivalued.

#### 4. Derived attribute:

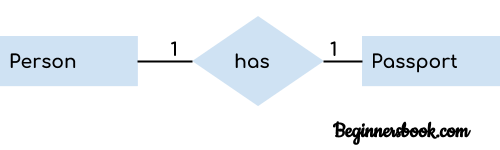
A derived attribute is one whose value is dynamic and derived from another attribute. It is represented by **dashed oval** in an ER Diagram. For example – Person age is a derived attribute as it changes over time and can be derived from another attribute (Date of birth).

**E-R diagram with multivalued and derived attributes**:  


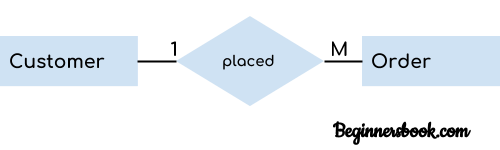
### 3. Relationship

A relationship is represented by diamond shape in ER diagram, it shows the relationship among entities. There are four types of relationships:  
1. One to One  
2. One to Many  
3. Many to One  
4. Many to Many

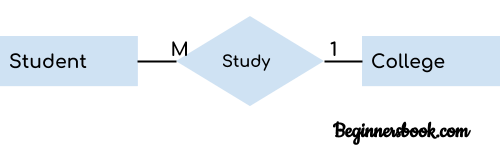
#### 1. One to One Relationship

When a single instance of an entity is associated with a single instance of another entity then it is called one to one relationship. For example, a person has only one passport and a passport is given to one person.  


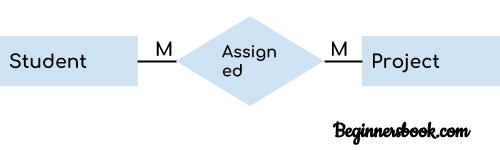
#### 2. One to Many Relationship

When a single instance of an entity is associated with more than one instances of another entity then it is called one to many relationship. For example – a customer can place many orders but a order cannot be placed by many customers.  


#### 3. Many to One Relationship

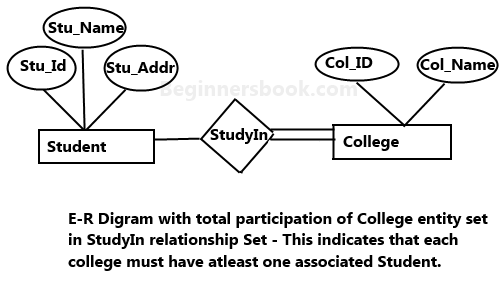
When more than one instances of an entity is associated with a single instance of another entity then it is called many to one relationship. For example – many students can study in a single college but a student cannot study in many colleges at the same time.  


#### 4. Many to Many Relationship

When more than one instances of an entity is associated with more than one instances of another entity then it is called many to many relationship. For example, a can be assigned to many projects and a project can be assigned to many students.  


## Total Participation of an Entity set

A Total participation of an entity set represents that each entity in entity set must have at least one relationship in a relationship set. For example: In the below diagram each college must have at-least one associated Student.



# Data models in DBMS

**Data Model** is a logical structure of Database. It describes the design of database to reflect entities, attributes, relationship among data, constrains etc.

## Types of Data Models

There are several types of data models in DBMS.

**Object based logical Models** – Describe data at the conceptual and view levels.

1. E-R Model
2. Object oriented Model

**Record based logical Models** – Like Object based model, they also describe data at the conceptual and view levels. These models specify logical structure of database with records, fields and attributes.

1. Relational Model
2. Hierarchical Model
3. Network Model – Network Model is same as hierarchical model except that it has graph-like structure rather than a tree-based structure. Unlike hierarchical model, this model allows each record to have more than one parent record.

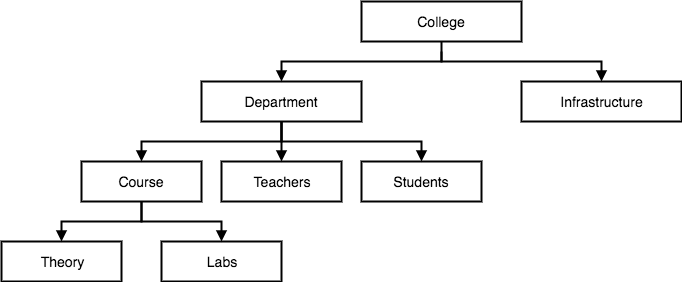
## Hierarchical Model

This database model organises data into a tree-like-structure, with a single root, to which all the other data is linked. The heirarchy starts from the **Root** data, and expands like a tree, adding child nodes to the parent nodes.

In this model, a child node will only have a single parent node.

This model efficiently describes many real-world relationships like index of a book, recipes etc.

In hierarchical model, data is organised into tree-like structure with one one-to-many relationship between two different types of data, for example, one department can have many courses, many professors and of-course many students.

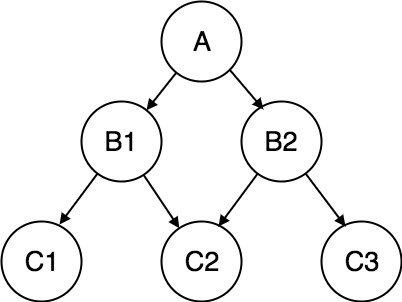


**Network Model**

This is an extension of the Hierarchical model. In this model data is organised more like a graph, and are allowed to have more than one parent node.

In this database model data is more related as more relationships are established in this database model. Also, as the data is more related, hence accessing the data is also easier and fast. This database model was used to map many-to-many data relationships.

This was the most widely used database model, before Relational Model was introduced.



**Relational Model**

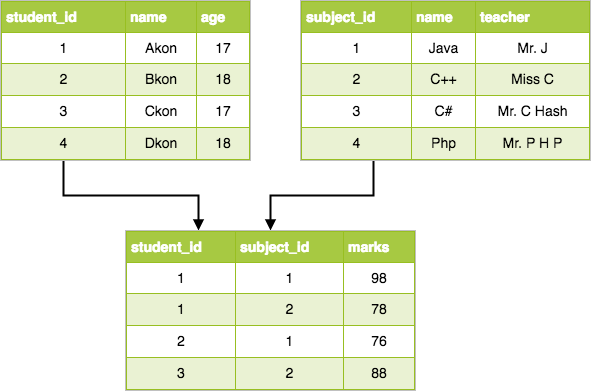
In this model, data is organised in two-dimensional **tables** and the relationship is maintained by storing a common field.

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.



**UNIT – II**

**RELATIONAL DATABASE**

A relational database is al database based on the relational model of data, as proposed by E. F. Codd in 1970. A software system used to maintain relational databases is a relational database management system (RDBMS). Many relational database systems have an option of using the SQL (Structured Query Language) for querying and maintaining the database. A relational database has following major components:  
1. Table  
2. Record or Tuple  
3. Field or Column name or Attribute  
4. Domain  
5. Instance  
6. Schema  
7. Keys

## 1. Table

A table is a collection of data represented in rows and columns. Each table has a name in database. For example, the following table “STUDENT” stores the information of students in database.

**Table: STUDENT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_Id** | **Student\_Name** | **Student\_Addr** | **Student\_Age** |
| 101 | Chaitanya | Dayal Bagh, Agra | 27 |
| 102 | Ajeet | Delhi | 26 |
| 103 | Rahul | Gurgaon | 24 |
| 104 | Shubham | Chennai | 25 |

## 2. Record or Tuple

Each row of a table is known as record. It is also known as tuple. For example, the following row is a record that we have taken from the above table.

|  |  |  |  |
| --- | --- | --- | --- |
| 102 | Ajeet | Delhi | 26 |

## 3. Field or Column name or Attribute

The above table “STUDENT” has four fields (or attributes): Student\_Id, Student\_Name, Student\_Addr & Student\_Age.

## 4. Domain

A domain is a set of permitted values for an attribute in table. For example, a domain of month-of-year can accept January, February,…December as values, a domain of dates can accept all possible valid dates etc. We specify domain of attribute while creating a table.

An attribute cannot accept values that are outside of their domains. For example, In the above table “STUDENT”, the Student\_Id field has integer domain so that field cannot accept values that are not integers for example, Student\_Id cannot has values like, “First”, 10.11

## 5. Keys

Key plays an important role in relational database; it is used for identifying unique rows from table. It also establishes relationship among tables.

**Types of Keys**

Primary Key – A primary is a column or set of columns in a table that uniquely identifies tuples (rows) in that table.

Super Key – A super key is a set of one of more columns (attributes) to uniquely identify rows in a table.

Candidate Key – A super key with no redundant attribute is known as candidate key

Alternate Key – Out of all candidate keys, only one gets selected as primary key, remaining keys are known as alternate or secondary keys.

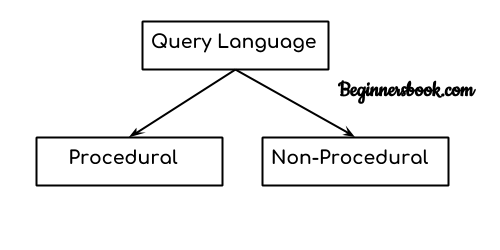
Composite Key – A key that consists of more than one attribute to uniquely identify rows (also known as records & tuples) in a table is called composite key.

Foreign Key – Foreign keys are the columns of a table that points to the primary key of another table. They act as a cross-reference between tables.

## Query Language

In simple words, a Language which is used to store and retrieve data from database is known as query language. For example – **SQL**

There are two types of query language:  
1.Procedural Query language  
2.Non-procedural query language



### 1. Procedural Query language:

In procedural query language, user instructs the system to perform a series of operations to produce the desired results. Here users tells what data to be retrieved from database and how to retrieve it.

**For example –** Let’s take a real world example to understand the procedural language, we are asking our younger brother to make a cup of tea, if we are just telling him to make a tea and not telling the process then it is a non-procedural language, however if we are telling the step by step process like switch on the stove, boil the water, add milk etc. then it is a procedural language.

### 2. Non-procedural query language:

In Non-procedural query language, user instructs the system to produce the desired result without telling the step by step process. Here users tells what data to be retrieved from database but doesn’t tell how to retrieve it.

Now let’s back to our main topic of relational algebra and relational calculus.

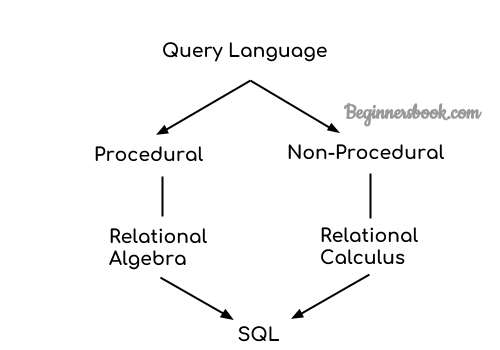
## Relational Algebra:

Relational algebra is a conceptual procedural query language used on relational model.

## Relational Calculus:

Relational calculus is a conceptual non-procedural query language used on relational model.

**Note:**  
I have used word conceptual while describing relational algebra and relational calculus, because they are theoretical mathematical system or query language, they are not the practical implementation, SQL is a practical implementation of relational algebra and relational calculus.



## Relational Algebra

Relational algebra is a **procedural** query language that works on relational model. The purpose of a query language is to retrieve data from database or perform various operations such as insert, update, delete on the data. When I say that relational algebra is a procedural query language, it means that it tells what data to be retrieved and how to be retrieved.

On the other hand relational calculus is a non-procedural query language, which means it tells what data to be retrieved but doesn’t tell how to retrieve it.

## Types of operations in relational algebra

We have divided these operations in two categories:  
1. Basic Operations  
2. Derived Operations

### Basic/Fundamental Operations:

1. Select (σ)  
2. Project (∏)  
3. Union (∪)  
4. Set Difference (-)  
5. Cartesian product (X)  
6. Rename (ρ)

### Derived Operations:

1. Natural Join (⋈)  
2. Left, Right, Full outer join (⟕, ⟖, ⟗)  
3. Intersection (∩)  
4. Division (÷)

Lets discuss these operations one by one with the help of examples.

## Select Operator (σ)

Select Operator is denoted by sigma (σ) and it is used to find the tuples (or rows) in a relation (or table) which satisfy the given condition.

**Syntax of Select Operator (σ)**

σ Condition/Predicate(Relation/Table name)

### Select Operator (σ) Example

Table: CUSTOMER

---------------

Customer\_Id Customer\_Name Customer\_City

----------- ------------- -------------

C10100 Steve Agra

C10111 Raghu Agra

C10115 Chaitanya Noida

C10117 Ajeet Delhi

C10118 Carl Delhi

**Query:**

σ Customer\_City="Agra" (CUSTOMER)

**Output:**

Customer\_Id Customer\_Name Customer\_City

----------- ------------- -------------

C10100 Steve Agra

C10111 Raghu Agra

## Project Operator (∏)

Project operator is denoted by ∏ symbol and it is used to select desired columns (or attributes) from a table (or relation).

Project operator in relational algebra is similar to the [Select statement in SQL](https://beginnersbook.com/2018/11/sql-select/).

**Syntax of Project Operator (∏)**

∏ column\_name1, column\_name2, ...., column\_nameN(table\_name)

### Project Operator (∏) Example

In this example, we have a table CUSTOMER with three columns, we want to fetch only two columns of the table, which we can do with the help of Project Operator ∏.

Table: CUSTOMER

Customer\_Id Customer\_Name Customer\_City

----------- ------------- -------------

C10100 Steve Agra

C10111 Raghu Agra

C10115 Chaitanya Noida

C10117 Ajeet Delhi

C10118 Carl Delhi

**Query:**

∏ Customer\_Name, Customer\_City (CUSTOMER)

**Output:**

Customer\_Name Customer\_City

------------- -------------

Steve Agra

Raghu Agra

Chaitanya Noida

Ajeet Delhi

Carl Delhi

## Union Operator (∪)

Union operator is denoted by ∪ symbol and it is used to select all the rows (tuples) from two tables (relations).

Lets discuss union operator a bit more. Lets say we have two relations R1 and R2 both have same columns and we want to select all the tuples(rows) from these relations then we can apply the union operator on these relations.

**Note:** The rows (tuples) that are present in both the tables will only appear once in the union set. In short we can say that there are no duplicates present after the union operation.

**Syntax of Union Operator (∪)**

table\_name1 ∪ table\_name2

### Union Operator (∪) Example

Table 1: COURSE

Course\_Id Student\_Name Student\_Id

--------- ------------ ----------

C101 Aditya S901

C104 Aditya S901

C106 Steve S911

C109 Paul S921

C115 Lucy S931

Table 2: STUDENT

Student\_Id Student\_Name Student\_Age

------------ ---------- -----------

S901 Aditya 19

S911 Steve 18

S921 Paul 19

S931 Lucy 17

S941 Carl 16

S951 Rick 18

**Query:**

∏ Student\_Name (COURSE) ∪ ∏ Student\_Name (STUDENT)

**Output:**

Student\_Name

------------

Aditya

Carl

Paul

Lucy

Rick

Steve

**Note:** As we can see there are no duplicate names present in the output even though we had few common names in both the tables, also in the COURSE table we had the duplicate name itself.

## Intersection Operator (∩)

Intersection operator is denoted by ∩ symbol and it is used to select common rows (tuples) from two tables (relations).

Lets say we have two relations R1 and R2 both have same columns and we want to select all those tuples(rows) that are present in both the relations, then in that case we can apply intersection operation on these two relations R1 ∩ R2.

**Note:** Only those rows that are present in both the tables will appear in the result set.

**Syntax of Intersection Operator (∩)**

table\_name1 ∩ table\_name2

### Intersection Operator (∩) Example

Lets take the same example that we have taken above.  
Table 1: COURSE

Course\_Id Student\_Name Student\_Id

--------- ------------ ----------

C101 Aditya S901

C104 Aditya S901

C106 Steve S911

C109 Paul S921

C115 Lucy S931

Table 2: STUDENT

Student\_Id Student\_Name Student\_Age

------------ ---------- -----------

S901 Aditya 19

S911 Steve 18

S921 Paul 19

S931 Lucy 17

S941 Carl 16

S951 Rick 18

**Query:**

∏ Student\_Name (COURSE) ∩ ∏ Student\_Name (STUDENT)

**Output:**

Student\_Name

------------

Aditya

Steve

Paul

Lucy

## Set Difference (-)

Set Difference is denoted by – symbol. Lets say we have two relations R1 and R2 and we want to select all those tuples(rows) that are present in Relation R1 but **not** present in Relation R2, this can be done using Set difference R1 – R2.

**Syntax of Set Difference (-)**

table\_name1 - table\_name2

### Set Difference (-) Example

Lets take the same tables COURSE and STUDENT that we have seen above.

**Query:**  
Lets write a query to select those student names that are present in STUDENT table but not present in COURSE table.

∏ Student\_Name (STUDENT) - ∏ Student\_Name (COURSE)

**Output:**

Student\_Name

------------

Carl

Rick

## Cartesian product (X)

Cartesian Product is denoted by X symbol. Lets say we have two relations R1 and R2 then the cartesian product of these two relations (R1 X R2) would combine each tuple of first relation R1 with the each tuple of second relation R2. I know it sounds confusing but once we take an example of this, we will be able to understand this.

**Syntax of Cartesian product (X)**

R1 X R2

### Cartesian product (X) Example

Table 1: R

Col\_A Col\_B

----- ------

AA 100

BB 200

CC 300

Table 2: S

Col\_X Col\_Y

----- -----

XX 99

YY 11

ZZ 101

**Query:**  
Lets find the cartesian product of table R and S.

R X S

**Output:**

Col\_A Col\_B Col\_X Col\_Y

----- ------ ------ ------

AA 100 XX 99

AA 100 YY 11

AA 100 ZZ 101

BB 200 XX 99

BB 200 YY 11

BB 200 ZZ 101

CC 300 XX 99

CC 300 YY 11

CC 300 ZZ 101

**Note:** The number of rows in the output will always be the cross product of number of rows in each table. In our example table 1 has 3 rows and table 2 has 3 rows so the output has 3×3 = 9 rows.

## Rename (ρ)

Rename (ρ) operation can be used to rename a relation or an attribute of a relation.  
**Rename (ρ) Syntax:**  
ρ(new\_relation\_name, old\_relation\_name)

### Rename (ρ) Example

Lets say we have a table customer, we are fetching customer names and we are renaming the resulted relation to CUST\_NAMES.

Table: CUSTOMER

Customer\_Id Customer\_Name Customer\_City

----------- ------------- -------------

C10100 Steve Agra

C10111 Raghu Agra

C10115 Chaitanya Noida

C10117 Ajeet Delhi

C10118 Carl Delhi

**Query:**

ρ(CUST\_NAMES, ∏(Customer\_Name)(CUSTOMER))

**Output:**

CUST\_NAMES

----------

Steve

Raghu

Chaitanya

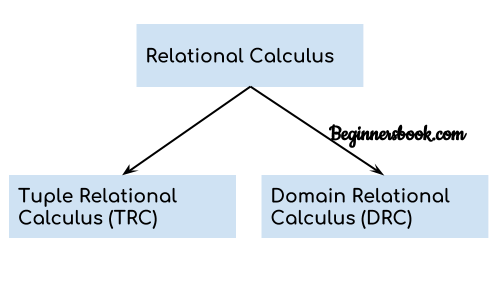
Ajeet

Carl

**What is Relational Calculus?**

Relational calculus is a non-procedural query language that tells the system what data to be retrieved but doesn’t tell how to retrieve it.

**Types of Relational Calculus**



**1. Tuple Relational Calculus (TRC)**

Tuple relational calculus is used for selecting those tuples that satisfy the given condition.  
Table: Student

First\_Name Last\_Name Age

---------- --------- ----

Ajeet Singh 30

Chaitanya Singh 31

Rajeev Bhatia 27

Carl Pratap 28

Lets write relational calculus queries.

Query to display the last name of those students where age is greater than 30

{ t.Last\_Name | Student(t) AND t.age > 30 }

In the above query we can see two parts separated by | symbol. The second part is where we define the condition and in the first part we specify the fields which we want to display for the selected tuples.

The result of the above query would be:

Last\_Name

---------

Singh

Query to display all the details of students where Last name is ‘Singh’

{ t | Student(t) AND t.Last\_Name = 'Singh' }

**Output:**

First\_Name Last\_Name Age

---------- --------- ----

Ajeet Singh 30

Chaitanya Singh 31

**2. Domain Relational Calculus (DRC)**

In domain relational calculus the records are filtered based on the domains.  
Again we take the same table to understand how DRC works.  
Table: Student

First\_Name Last\_Name Age

---------- --------- ----

Ajeet Singh 30

Chaitanya Singh 31

Rajeev Bhatia 27

Carl Pratap 28

Query to find the first name and age of students where student age is greater than 27

{< First\_Name, Age > | ∈ Student ∧ Age > 27}

**Note:**  
The symbols used for logical operators are: ∧ for AND, ∨ for OR and ┓ for NOT.

**Output:**

First\_Name Age

---------- ----

Ajeet 30

Chaitanya 31

Carl 28

## NULL Value

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value. A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

## How to Test for NULL Values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

### IS NULL Syntax

SELECT column\_namesFROM table\_name  
WHERE column\_name IS NULL;

### IS NOT NULL Syntax

SELECT column\_namesFROM table\_name  
WHERE column\_name IS NOT NULL;

**Modification of the Databases**

The **Database Modification** is the generation of SQL script that leads database to change some of its values in its current state. Some of SQL commands are used for storing, retrieving, modifying, and deleting data. These SQL commands are used for managing changes affecting the data.

## The SQL INSERT INTO Statement

The INSERT INTO statement is used to insert new records in a table.

### INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways.

The first way specifies both the column names and the values to be inserted:

INSERT INTO table\_name (column1, column2, column3, ...)  
VALUES (value1, value2, value3, ...);

If we are adding values for all the columns of the table, we do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. The INSERT INTO syntax would be as follows:

INSERT INTO table\_name  
VALUES (value1, value2, value3, ...);

## INSERT INTO Example

The following SQL statement inserts a new record in the "Customers" table:

### Example

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)  
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

## Insert Data Only in Specified Columns

It is also possible to only insert data in specific columns.

The following SQL statement will insert a new record, but only insert data in the "CustomerName", "City", and "Country" columns (CustomerID will be updated automatically)**:**

### Example

INSERT INTO Customers (CustomerName, City, Country)  
VALUES ('Cardinal', 'Stavanger', 'Norway');

## The SQL UPDATE Statement

The UPDATE statement is used to modify the existing records in a table.

### UPDATE Syntax

UPDATE table\_name  
SET column1 = value1, column2 = value2, ...  
WHERE condition;

**Note:** Be careful when updating records in a table! Notice the WHERE clause in the UPDATE statement. The WHERE clause specifies which record(s) that should be updated. If we omit the WHERE clause, all records in the table will be updated!

The following SQL statement updates the first customer (CustomerID = 1) with a new contact person and a new city.

### Example

UPDATE Customers  
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'  
It is the WHERE clause that determines how many records will be updated.

The following SQL statement will update the contactname to "Juan" for all records where country is "Mexico":

### Example

UPDATE Customers  
SET ContactName='Juan'  
WHERE Country='Mexico'

WHERE CustomerID = 1;

## The SQL DELETE Statement

The DELETE statement is used to delete existing records in a table.

### DELETE Syntax

DELETE FROM table\_name WHERE condition;

**Note:** Be careful when deleting records in a table! Notice the WHERE clause in the DELETE statement. The WHERE clause specifies which record(s) should be deleted. If we omit the WHERE clause, all records in the table will be deleted!

The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:

### Example

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

DELETE FROM table\_name;

The following SQL statement deletes all rows in the "Customers" table, without deleting the table:

### Example

DELETE FROM Customers;

**UNIT – III**

**NORMALIZATION**

**Normalization** is a process of organizing the data in database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly. Let’s discuss about anomalies first then we will discuss normal forms with examples.

## Anomalies in DBMS

There are three types of anomalies that occur when the database is not normalized. These are – Insertion, update and deletion anomaly. Let’s take an example to understand this.

**Example**: Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp\_id for storing employee’s id, emp\_name for storing employee’s name, emp\_address for storing employee’s address and emp\_dept for storing the department details in which the employee works. At some point of time the table looks like this:

|  |  |  |  |
| --- | --- | --- | --- |
| **emp\_id** | **emp\_name** | **emp\_address** | **emp\_dept** |
| 101 | Rick | Delhi | D001 |
| 101 | Rick | Delhi | D002 |
| 123 | Maggie | Agra | D890 |
| 166 | Glenn | Chennai | D900 |
| 166 | Glenn | Chennai | D004 |

The above table is not normalized. We will see the problems that we face when a table is not normalized.

**Update anomaly**: In the above table we have two rows for employee Rick as he belongs to two departments of the company. If we want to update the address of Rick then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, Rick would be having two different addresses, which is not correct and would lead to inconsistent data.

**Insert anomaly**: Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp\_dept field doesn’t allow nulls.

**Delete anomaly**: Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp\_dept as D890 would also delete the information of employee Maggie since she is assigned only to this department.

To overcome these anomalies we need to normalize the data. In the next section we will discuss about normalization.

## Normalization

Here are the most commonly used normal forms:

* First normal form(1NF)
* Second normal form(2NF)
* Third normal form(3NF)
* Boyce & Codd normal form (BCNF)

## First normal form (1NF)

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

**Example**: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

|  |  |  |  |
| --- | --- | --- | --- |
| **emp\_id** | **emp\_name** | **emp\_address** | **emp\_mobile** |
| 101 | Herschel | New Delhi | 8912312390 |
| 102 | Jon | Kanpur | 8812121212  9900012222 |
| 103 | Ron | Chennai | 7778881212 |
| 104 | Lester | Bangalore | 9990000123  8123450987 |

Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as we can see in the table above.

This table is **not in 1NF**as the rule says “each attribute of a table must have atomic (single) values”, the emp\_mobile values for employees Jon & Lester violates that rule.

To make the table complies with 1NF we should have the data like this:

|  |  |  |  |
| --- | --- | --- | --- |
| **emp\_id** | **emp\_name** | **emp\_address** | **emp\_mobile** |
| 101 | Herschel | New Delhi | 8912312390 |
| 102 | Jon | Kanpur | 8812121212 |
| 102 | Jon | Kanpur | 9900012222 |
| 103 | Ron | Chennai | 7778881212 |
| 104 | Lester | Bangalore | 9990000123 |
| 104 | Lester | Bangalore | 8123450987 |

## Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

* Table is in 1NF (First normal form)
* No non-prime attribute is dependent on the proper subset of any candidate key of table.

An attribute that is not part of any candidate key is known as non-prime attribute.

**Example**: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

|  |  |  |
| --- | --- | --- |
| teacher\_id | subject | teacher\_age |
| 111 | Maths | 38 |
| 111 | Physics | 38 |
| 222 | Biology | 38 |
| 333 | Physics | 40 |
| 333 | Chemistry | 40 |

**Candidate Keys**: {teacher\_id, subject}  
**Non prime attribute**: teacher\_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher age is dependent on teacher\_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says “**no** non-prime attribute is dependent on the proper subset of any candidate key of the table”.

To make the table complies with 2NF we can break it in two tables like this:  
**teacher\_details table:**

|  |  |
| --- | --- |
| teacher\_id | teacher\_age |
| 111 | 38 |
| 222 | 38 |
| 333 | 40 |

**teacher\_subject table:**

|  |  |
| --- | --- |
| teacher\_id | Subject |
| 111 | Maths |
| 111 | Physics |
| 222 | Biology |
| 333 | Physics |
| 333 | Chemistry |

Now the tables comply with Second normal form (2NF).

## Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

* Table must be in 2NF
* [Transitive functional dependency](https://beginnersbook.com/2015/04/transitive-dependency-in-dbms/) of non-prime attribute on any super key should be removed.

An attribute that is not part of any [candidate key](https://beginnersbook.com/2015/04/candidate-key-in-dbms/) is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency X-> Y at least one of the following conditions hold:

* X is a [super key](https://beginnersbook.com/2015/04/super-key-in-dbms/) of table
* Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

**Example**: Suppose a company wants to store the complete address of each employee, they create a table named employee\_details that looks like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **emp\_id** | **emp\_name** | **emp\_zip** | **emp\_state** | **emp\_city** | **emp\_district** |
| 1001 | John | 282005 | UP | Agra | Dayal Bagh |
| 1002 | Ajeet | 222008 | TN | Chennai | M-City |
| 1006 | Lora | 282007 | TN | Chennai | Urrapakkam |
| 1101 | Lilly | 292008 | UK | Pauri | Bhagwan |
| 1201 | Steve | 222999 | MP | Gwalior | Ratan |

**Super keys**: {emp\_id}, {emp\_id, emp\_name}, {emp\_id, emp\_name, emp\_zip}…so on  
**Candidate Keys**: {emp\_id}  
**Non-prime attributes**: all attributes except emp\_id are non-prime as they are not part of any candidate keys.

Here, emp\_state, emp\_city & emp\_district dependent on emp\_zip. And, emp\_zip is dependent on emp\_id that makes non-prime attributes (emp\_state, emp\_city & emp\_district) transitively dependent on super key (emp\_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

**employee table:**

|  |  |  |
| --- | --- | --- |
| **emp\_id** | **emp\_name** | **emp\_zip** |
| 1001 | John | 282005 |
| 1002 | Ajeet | 222008 |
| 1006 | Lora | 282007 |
| 1101 | Lilly | 292008 |
| 1201 | Steve | 222999 |

**employee\_zip table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **emp\_zip** | **emp\_state** | **emp\_city** | **emp\_district** |
| 282005 | UP | Agra | Dayal Bagh |
| 222008 | TN | Chennai | M-City |
| 282007 | TN | Chennai | Urrapakkam |
| 292008 | UK | Pauri | Bhagwan |
| 222999 | MP | Gwalior | Ratan |

## Boyce Codd normal form (BCNF)

It is an advance version of 3NF that’s why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every [functional dependency](https://beginnersbook.com/2015/04/functional-dependency-in-dbms/) X->Y, X should be the super key of the table.

**Example**: Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **emp\_id** | **emp\_nationality** | **emp\_dept** | **dept\_type** | **dept\_no\_of\_emp** |
| 1001 | Austrian | Production and planning | D001 | 200 |
| 1001 | Austrian | stores | D001 | 250 |
| 1002 | American | design and technical support | D134 | 100 |
| 1002 | American | Purchasing department | D134 | 600 |

**Functional dependencies in the table above**:  
emp\_id -> emp\_nationality  
emp\_dept -> {dept\_type, dept\_no\_of\_emp}

**Candidate key**: {emp\_id, emp\_dept}

The table is not in BCNF as neither emp\_id nor emp\_dept alone are keys.

To make the table comply with BCNF we can break the table in three tables like this:  
**emp\_nationality table:**

|  |  |
| --- | --- |
| **emp\_id** | **emp\_nationality** |
| 1001 | Austrian |
| 1002 | American |

**emp\_dept table:**

|  |  |  |
| --- | --- | --- |
| **emp\_dept** | **dept\_type** | **dept\_no\_of\_emp** |
| Production and planning | D001 | 200 |
| Stores | D001 | 250 |
| design and technical support | D134 | 100 |
| Purchasing department | D134 | 600 |

**emp\_dept\_mapping table:**

|  |  |
| --- | --- |
| emp\_id | emp\_dept |
| 1001 | Production and planning |
| 1001 | stores |
| 1002 | design and technical support |
| 1002 | Purchasing department |

**Functional dependencies**:  
emp\_id -> emp\_nationality  
emp\_dept -> {dept\_type, dept\_no\_of\_emp}

**Candidate keys**:  
For first table: emp\_id  
For second table: emp\_dept  
For third table: {emp\_id, emp\_dept}

This is now in BCNF as in both the functional dependencies left side part is a key.

**UNIT – IV**

**STRUCTURED QUERY LANGUAGE (SQL)**

**Introduction**

SQL stands for Structured Query Language. It is used for storing and managing data in relational database management system (RDMS).It is a standard language for Relational Database System. It enables a user to create, read, update and delete relational databases and tables. All the RDBMS like MySQL, Informix, Oracle, MS Access and SQL Server use SQL as their standard database language.SQL allows users to query the database in a number of ways, using English-like statements.

SQL follows the following rules:

* Structure query language is not case sensitive. Generally, keywords of SQL are written in uppercase.
* Statements of SQL are dependent on text lines. We can use a single SQL statement on one or multiple text line.
* Using the SQL statements, we can perform most of the actions in a database.
* SQL depends on tuple relational calculus and relational algebra.

### SQL process

When an SQL command is executing for any RDBMS, then the system figure out the best way to carry out the request and the SQL engine determines that how to interpret the task. In the process, various components are included. These components can be optimization Engine, Query engine, Query dispatcher, classic, etc. All the non-SQL queries are handled by the classic query engine, but SQL query engine won't handle logical files.

**Characteristics / Features of SQL**

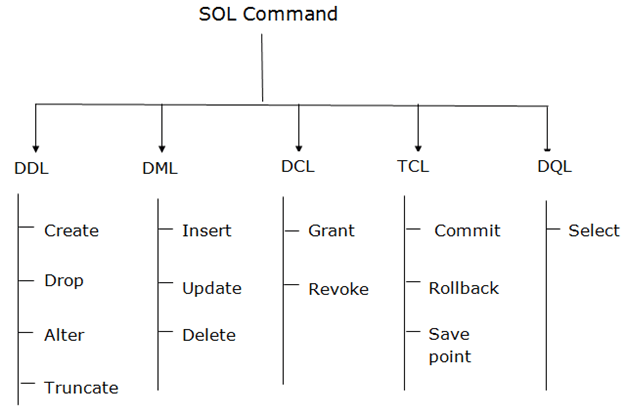
* SQL is easy to learn.
* SQL is used to access data from relational database management systems.
* SQL can execute queries against the database.
* SQL is used to describe the data.
* SQL is used to define the data in the database and manipulate it when needed.
* SQL is used to create and drop the database and table.
* SQL is used to create a view, stored procedure, function in a database.
* SQL allows users to set permissions on tables, procedures, and views.

# SQL Commands

* SQL commands are instructions. It is used to communicate with the database. It is also used to perform specific tasks, functions, and queries of data.
* SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.

## Types of SQL Commands

There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.



### 1. Data Definition Language (DDL)

* DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.
* All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

* CREATE
* ALTER
* DROP
* TRUNCATE

**a. CREATE** It is used to create a new table in the database.

**Syntax:**

CREATE TABLE TABLE\_NAME (COLUMN\_NAME DATATYPES[,....]);

**Example:**

CREATE TABLE EMPLOYEE(Name VARCHAR2(20), Email VARCHAR2(100), DOB DATE);

**b. DROP:** It is used to delete both the structure and record stored in the table.

**Syntax**

DROP TABLE ;

**Example**

DROP TABLE EMPLOYEE;

**c. ALTER:** It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.

**Syntax:**

To add a new column in the table

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

To modify existing column in the table:

ALTER TABLE MODIFY(COLUMN DEFINITION....);

**EXAMPLE**

ALTER TABLE STU\_DETAILS ADD(ADDRESS VARCHAR2(20));

ALTER TABLE STU\_DETAILS MODIFY (NAME VARCHAR2(30));

**d. TRUNCATE:** It is used to delete all the rows from the table and free the space containing the table.

**Syntax:**

TRUNCATE TABLE table\_name;

**Example:**

TRUNCATE TABLE EMPLOYEE;

DESC EMPLOYEE

### 2. Data Manipulation Language

* DML commands are used to modify the database. It is responsible for all form of changes in the database.
* The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rollback.

Here are some commands that come under DML:

* INSERT
* UPDATE
* DELETE

**a. INSERT:** The INSERT statement is a SQL query. It is used to insert data into the row of a table.

**Syntax:**

INSERT INTO TABLE\_NAME

(col1, col2, col3,.... col N)

VALUES (value1, value2, value3, .... valueN);

Or

INSERT INTO TABLE\_NAME

VALUES (value1, value2, value3, .... valueN);

**For example:**

INSERT INTO javatpoint (Author, Subject) VALUES ("Sonoo", "DBMS");

INTERT INTO JAVAPOINT VALUES (“KRISHNA”, 980055,”KAANOO”,”,RDBMS”):

**b. UPDATE:** This command is used to update or modify the value of a column in the table.

**Syntax:**

UPDATE table\_name SET [column\_name1= value1,...column\_nameN = valueN] [WHERE CONDITION]

**For example:**

UPDATE students

SET User\_Name = 'Sonoo'

WHERE Student\_Id = '3'

**c. DELETE:** It is used to remove one or more row from a table.

**Syntax:**

DELETE FROM table\_name [WHERE condition];

**For example:**

DELETE FROM javatpoint

WHERE Author="Sonoo";

### 3. Data Control Language

DCL commands are used to grant and take back authority from any database user.

Here are some commands that come under DCL:

* Grant
* Revoke

**a. Grant:** It is used to give user access privileges to a database.

**Example**

GRANT SELECT, UPDATE ON MY\_TABLE TO SOME\_USER, ANOTHER\_USER;

**b. Revoke:** It is used to take back permissions from the user.

**Example**

REVOKE SELECT, UPDATE ON MY\_TABLE FROM USER1, USER2;

### 4. Transaction Control Language

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only.

These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Here are some commands that come under TCL:

* COMMIT
* ROLLBACK
* SAVEPOINT

**a. Commit:** Commit command is used to save all the transactions to the database.

**Syntax:**

COMMIT;

**Example:**

DELETE FROM CUSTOMERS

WHERE AGE = 25;

COMMIT;

**b. Rollback:** Rollback command is used to undo transactions that have not already been saved to the database.

**Syntax:**

ROLLBACK;

**Example:**

DELETE FROM CUSTOMERS

WHERE AGE = 25;

ROLLBACK;

**c. SAVEPOINT:** It is used to roll the transaction back to a certain point without rolling back the entire transaction.

**Syntax:**

SAVEPOINT SAVEPOINT\_NAME;

### 5. Data Query Language

DQL is used to fetch the data from the database.

It uses only one command:

* SELECT

**a. SELECT:** This is the same as the projection operation of relational algebra. It is used to select the attribute based on the condition described by WHERE clause.

**Syntax:**

SELECT expressions

FROM TABLES

WHERE conditions;

**For example:**

SELECT emp\_name

FROM employee

WHERE age > 20;

**Select SQL Operations**

In SQL, the SELECT statement is used to query or retrieve data from a table in the database. The returns data is stored in a table, and the result table is known as result-set.

**Syntax**

SELECT column1, column2, ...

FROM table\_name;

Here, the expression is the field name of the table that we want to select data from.

Use the following syntax to select all the fields available in the table:

SELECT  \*  FROM table\_name;

Below is a selection from the "Customers" table in the Northwind sample database:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

The following SQL statement selects the "CustomerName" and "City" columns from the "Customers" table:

SELECT CustomerName, City FROM Customers;

The following SQL statement selects only the DISTINCT values from the "Country" column in the "Customers" table:

SELECT DISTINCT Country FROM Customers;

The following SQL statement lists the number of different (distinct) customer countries:

SELECT COUNT(DISTINCT Country) FROM Customers;

## The SQL WHERE Clause

The WHERE clause is used to filter records.

The WHERE clause is used to extract only those records that fulfill a specified condition.

### WHERE Syntax

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition;

**Note:** The WHERE clause is not only used in SELECT statement, it is also used in UPDATE, DELETE statement, etc.!

## Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| <> | Not equal. **Note:** In some versions of SQL this operator may be written as != |
| BETWEEN | Between a certain range |
| LIKE | Search for a pattern |
| IN | To specify multiple possible values for a column |

## The SQL AND, OR and NOT Operators

The WHERE clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition:

* The AND operator displays a record if all the conditions separated by AND are TRUE.
* The OR operator displays a record if any of the conditions separated by OR is TRUE.

The NOT operator displays a record if the condition(s) is NOT TRUE.

### AND Syntax

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition1 AND condition2 AND condition3 ...;

### OR Syntax

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition1 OR condition2 OR condition3 ...;

### NOT Syntax

SELECT column1, column2, ...  
FROM table\_name  
WHERE NOT condition;

## AND Example

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city is "Berlin":

### Example

SELECT \* FROM Customers  
WHERE Country='Germany' AND City='Berlin';

## OR Example

The following SQL statement selects all fields from "Customers" where city is "Berlin" OR "München":

### Example

SELECT \* FROM Customers  
WHERE City='Berlin' OR City='München';

The following SQL statement selects all fields from "Customers" where country is "Germany" OR "Spain":

### Example

SELECT \* FROM Customers  
WHERE Country='Germany' OR Country='Spain';

## NOT Example

The following SQL statement selects all fields from "Customers" where country is NOT "Germany":

### Example

SELECT \* FROM Customers  
WHERE NOT Country='Germany';

## Combining AND, OR and NOT

We can also combine the AND, OR and NOT operators.

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city must be "Berlin" OR "München" (use parenthesis to form complex expressions):

### Example

SELECT \* FROM Customers  
WHERE Country='Germany' AND (City='Berlin' OR City='München');

The following SQL statement selects all fields from "Customers" where country is NOT "Germany" and NOT "USA":

### Example

SELECT \* FROM Customers  
WHERE NOT Country='Germany' AND NOT Country='USA';

## The SQL ORDER BY Keyword

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

### ORDER BY Syntax

SELECT column1, column2, ...  
FROM table\_name  
ORDER BY column1, column2, ... ASC|DESC;

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" column:

### Example

SELECT \* FROM Customers  
ORDER BY Country;

The following SQL statement selects all customers from the "Customers" table, sorted DESCENDING by the "Country" column:

### Example

SELECT \* FROM Customers  
ORDER BY Country DESC;

Grouping the Output of the Query

In SQL, The **Group By** statement is used for organizing similar data into groups. The data is further organized with the help of equivalent function. It means, if different rows in a precise column have the same values, it will arrange those rows in a group.

* **The SELECT** statement is used with the **GROUP BY** clause in the SQL query.
* **WHERE** clause is placed before the **GROUP BY** clause in **SQL**.
* **ORDER BY** clause is placed after the **GROUP BY** clause in **SQL**.

### Syntax:

**SELECT** column1, function\_name(column2)

**FROM** table\_name

**WHERE** condition

**GROUP** **BY** column1, column2

**ORDER** **BY** column1, column2;

function\_name: **Table** **name**.

Condition: which we used.

### Sample Table:

### Employee

|  |  |  |  |
| --- | --- | --- | --- |
| **S.no** | **Name** | **AGE** | **Salary** |
| **1** | **John** | **24** | **25000** |
| **2** | **Nick** | 22 | 22000 |
| **3** | **Amara** | 25 | 15000 |
| **4** | **Nick** | 22 | 22000 |
| **5** | **John** | 24 | 25000 |

### Student

|  |  |  |
| --- | --- | --- |
| **SUBJECT** | **YEAR** | **NAME** |
| **C language** | **2** | **John** |
| **C language** | **2** | Ginny |
| **C language** | **2** | Jasmeen |
| **C language** | **3** | Nick |
| **C language** | **3** | Amara |
| **Java** | **1** | Sifa |
| **Java** | **1** | Dolly |

### Example:

**Group By single column: Group By** single column is used to place all the rows with the same value. These values are of that specified column in one group. It signifies that all rows will put an equal amount through a single column, which is of one appropriate column in one group.

Consider the below query:

**SELECT** **NAME**, SUM (SALARY) **FROM** Employee

**GROUP** **BY** **NAME**;

The output of the query is:

|  |  |
| --- | --- |
| **NAME** | **SALARY** |
| **John** | 50000 |
| **Nick** | 44000 |
| **Amara** | 15000 |

In the output, the rows which hold duplicate **NAME** are grouped under a similar NAME, and their corresponding SALARY is the sum of the SALARY of the duplicate rows.

* **Groups based on several columns:** A group of some columns are **GROUP BY** **column 1**, **column2**, **etc**. Here, we are placing all rows in a group with the similar values of both **column 1** and **column 2**.

Consider the below query:

**SELECT** SUBJECT, YEAR, Count (\*)

**FROM** Student

**Group** **BY** SUBJECT, YEAR;

**Output:**

|  |  |  |
| --- | --- | --- |
| **SUBJECT** | **YEAR** | **Count** |
| **C language** | 2 | 3 |
| **C language** | 3 | 2 |
| **Java** | 1 | 2 |

In the above output, the student with similar **SUBJECT** and **YEAR** are grouped in the same place. The students who have only one thing in common belongs to different groups. For example, if the NAME is same and the YEAR is different.

Now, we have to group the table according to more than one column or two columns.

**HAVING Clause**

**WHERE** clause is used for deciding purpose. It is used to place conditions on the columns to determine the part of the last result-set of the group. Here, we are not required to use the combined functions like **COUNT (), SUM (),** etc. with the**WHERE** clause. After that, we need to use a **HAVING** clause.

### Having clause Syntax:

**SELECT** column1, function\_name(column2)

**FROM** table\_name

**WHERE** condition

**GROUP** **BY** column1, column2

**HAVING** condition

**ORDER** **BY** column1, column2;

function\_name:  Mainly used **for** **name** **of** the **function**, SUM(), AVG().

table\_name: Used **for** **name** **of** the **table**.

condition: Condition used.

### Example:

**SELECT** **NAME**, SUM(SALARY) **FROM** Employee

**GROUP** **BY** **NAME**

**HAVING** SUM(SALARY)>23000;

**Output:**

|  |  |
| --- | --- |
| **Name** | **SUM(SALARY)** |
| **John** | 50000 |

According to the above output, only one name in the NAME column has been listed in the result because there is only one data in the database whose sum of salary is more than 50000.

It should be placed on groups, not on the columns.

### Points:

* The **GROUP BY** Clause is used to group the rows, which have the same values.
* The **SELECT** statement in SQL is used with the GROUP BY clause.
* In the **Group BY** clause, the SELECT statement can use**constants,** **aggregate functions, expressions,**and**column names**.
* The **GROUP BY** Clause is called when the HAVING clause is used to reduce the results.

# Querying from Multiple Tables SQL SELECT from Multiple Tables

This statement is used to retrieve fields from multiple tables. To do so, we need to use join query to get data from multiple tables.

Let's see the example for the select from multiple tables:

**SELECT** orders.order\_id, suppliers.**name**

**FROM** suppliers

**INNER** JOIN orders

**ON** suppliers.supplier\_id = orders.supplier\_id

**ORDER** **BY** order\_id;

Let us take three tables, two tables of customers named customer1 and customer2 and the third table is product table.

**Customer1 table**

|  |  |
| --- | --- |
| **Cus\_id** | **Name1** |
| 1 | Jack |
| 2 | Jill |

**Customer2 table**

|  |  |
| --- | --- |
| **Cus\_id** | **Name2** |
| 1 | Sandy |
| 2 | Venus |

**Product table**

|  |  |  |
| --- | --- | --- |
| **P\_id** | **Cus\_id** | **P\_name** |
| 1 | 1 | Laptop |
| 2 | 2 | Phone |
| 3 | P1 | Pen |
| 4 | P2 | Notebook |

**Example syntax to select from multiple tables:**

**SELECT** p. p\_id, p.cus\_id, p.p\_name, c1.name1, c2.name2

**FROM** product **AS** p

LEFT JOIN customer1 **AS** c1

**ON** p.cus\_id=c1.cus\_id

LEFT JOIN customer2 **AS** c2

**ON** p.cus\_id = c2.cus\_id

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_id** | **Cus\_id** | **P\_name** | **P\_name** | **P\_name** |
| 1 | 1 | Laptop | Jack | NULL |
| 2 | 2 | Phone | Jill | NULL |
| 3 | P1 | Pen | NULL | Sandy |
| 4 | P2 | Notebook | NULL |  |

# SET Operations in SQL

SQL supports few Set operations which can be performed on the table data. These are used to get meaningful results from data stored in the table, under different special conditions. The SQL Set operation is used to combine the two or more SQL SELECT statements.

## Types of Set Operation

1. Union
2. UnionAll
3. Intersect
4. Minus



### 1. Union

* The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
* In the union operation, all the number of datatype and columns must be same in both the tables on which UNION operation is being applied.
* The union operation eliminates the duplicate rows from its resultset.

**Syntax**

SELECT column\_name FROM table1

UNION

SELECT column\_name FROM table2;

**Example:**

**The First table**

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Jack |
| 2 | Harry |
| 3 | Jackson |

**The Second table**

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 3 | Jackson |
| 4 | Stephan |
| 5 | David |

Union SQL query will be:

SELECT \* FROM First

UNION

SELECT \* FROM Second;

The resultset table will look like:

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Jack |
| 2 | Harry |
| 3 | Jackson |
| 4 | Stephan |
| 5 | David |

### 2. Union All

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

**Syntax:**

SELECT column\_name FROM table1

UNION ALL

SELECT column\_name FROM table2;

**Example:** Using the above First and Second table.

Union All query will be like:

SELECT \* FROM First

UNION ALL

SELECT \* FROM Second;

The resultset table will look like:

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Jack |
| 2 | Harry |
| 3 | Jackson |
| 3 | Jackson |
| 4 | Stephan |
| 5 | David |

### 3. Intersect

* It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
* In the Intersect operation, the number of datatype and columns must be the same.
* It has no duplicates and it arranges the data in ascending order by default.

**Syntax**

SELECT column\_name FROM table1

INTERSECT

SELECT column\_name FROM table2;

**Example:**

**Using the above First and Second table.**

Intersect query will be:

SELECT \* FROM First

INTERSECT

SELECT \* FROM Second;

The resultset table will look like:

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 3 | Jackson |

### 4. Minus

It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.

It has no duplicates and data arranged in ascending order by default.

**Syntax:**

SELECT column\_name FROM table1

MINUS

SELECT column\_name FROM table2;

**Example**

**Using the above First and Second table.**

Minus query will be:

SELECT \* FROM First

MINUS

SELECT \* FROM Second;

The resultset table will look like:

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Jack |
| 2 | Harry |

Nested Queries/ Sub Queries

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

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

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that subqueries must follow −

* Subqueries must be enclosed within parentheses.
* A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
* An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
* Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
* The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
* A subquery cannot be immediately enclosed in a set function.
* The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

## Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows −

SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

WHERE column\_name OPERATOR

(SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

[WHERE])

### Example

Consider the CUSTOMERS table having the following records −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 35 | 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 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

Now, let us check the following subquery with a SELECT statement.

SQL> SELECT \*

FROM CUSTOMERS

WHERE ID IN (SELECT ID

FROM CUSTOMERS

WHERE SALARY > 4500) ;

This would produce the following result.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

## Subqueries with the INSERT Statement

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

The basic syntax is as follows.

INSERT INTO table\_name [ (column1 [, column2 ]) ]

SELECT [ \*|column1 [, column2 ]

FROM table1 [, table2 ]

[ WHERE VALUE OPERATOR ]

### Example

Consider a table CUSTOMERS\_BKP with similar structure as CUSTOMERS table. Now to copy the complete CUSTOMERS table into the CUSTOMERS\_BKP table, we can use the following syntax.

SQL> INSERT INTO CUSTOMERS\_BKP

SELECT \* FROM CUSTOMERS

WHERE ID IN (SELECT ID

FROM CUSTOMERS) ;

## Subqueries with the UPDATE Statement

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

The basic syntax is as follows.

UPDATE table

SET column\_name = new\_value

[ WHERE OPERATOR [ VALUE ]

(SELECT COLUMN\_NAME

FROM TABLE\_NAME)

[ WHERE) ]

### Example

Assuming, we have CUSTOMERS\_BKP table available which is backup of CUSTOMERS table. The following example updates SALARY by 0.25 times in the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

SQL> UPDATE CUSTOMERS

SET SALARY = SALARY \* 0.25

WHERE AGE IN (SELECT AGE FROM CUSTOMERS\_BKP

WHERE AGE >= 27 );

This would impact two rows and finally CUSTOMERS table would have the following records.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 35 | Ahmedabad | 125.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 2125.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

## Subqueries with the DELETE Statement

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

The basic syntax is as follows.

DELETE FROM TABLE\_NAME

[ WHERE OPERATOR [ VALUE ]

(SELECT COLUMN\_NAME

FROM TABLE\_NAME)

[ WHERE) ]

### Example

Assuming, we have a CUSTOMERS\_BKP table available which is a backup of the CUSTOMERS table. The following example deletes the records from the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

SQL> DELETE FROM CUSTOMERS

WHERE AGE IN (SELECT AGE FROM CUSTOMERS\_BKP

WHERE AGE >= 27 );

This would impact two rows and finally the CUSTOMERS table would have the following records.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

**T-SQL (Transact-SQL)**

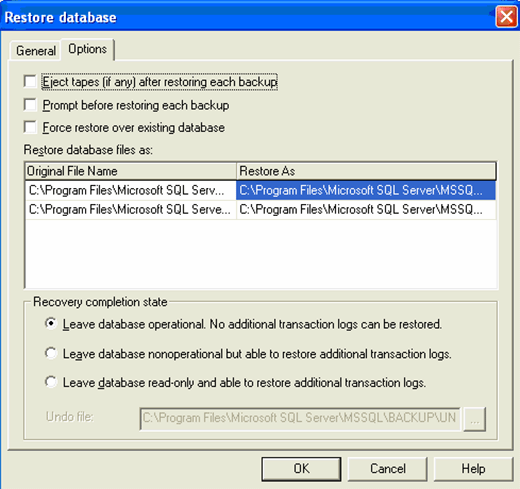
T-SQL (Transact-SQL) is a set of programming extensions from Sybase and Microsoft that add several features to the Structured Query Language ([SQL](https://searchsqlserver.techtarget.com/definition/SQL)), including transaction control, exception and error handling, row processing and declared variables.

All applications that communicate with SQL Server do so by sending T-SQL statements to the server. T-SQL queries include the SELECT statement, selecting columns, labeling output columns, restricting rows and modifying a search condition.

T-SQL identifiers, meanwhile, are used in all databases, servers, and database objects in SQL Server. These include the following tables, constraints, stored procedures, views, columns and [data types](https://searchapparchitecture.techtarget.com/definition/data-type). T-SQL identifiers must each have a unique name, are assigned when an object is created and are used to identify an object.

The most popular T-SQL statement is the stored procedure, which is a compiled and stored T-SQL code. Similar to views, [stored procedures](https://searchoracle.techtarget.com/definition/stored-procedure) generate an execution plan when called the first time. The difference is stored procedures can select data and execute any T-SQL code within any parameters.

User-defined functions are another example of T-SQL statements. User-defined functions take input parameters, perform an action and return the results to the call.



Restore a database with T-SQL.

Another example is a [trigger](https://searchsqlserver.techtarget.com/definition/trigger), which is a stored T-SQL script that runs when a statement other than SELECT is issued against a table or view. The two common triggers are AFTER triggers and INSTEAD OF triggers.

Programming T-SQL statements enables IT pros to build applications contained within SQL Server. These applications -- or objects -- can insert, update, delete or read data stored in a database.

Common language runtime ([CLR](https://whatis.techtarget.com/definition/Common-Language-Runtime-CLR)) integration is the final T-SQL statement example. Since SQL Server 2005, IT pros can integrate with the [.NET](https://searchwindevelopment.techtarget.com/definition/NET) Framework CLR. This enables us to use .NET programming languages within SQL Server objects to create stored procedures, user-defined functions and triggers.

### T-SQL Functions

In addition to SQL Server's built-in functions, users can define functions using T-SQL.

Types of T-SQL functions include:

* Aggregate functions, which operate on a collection of values, but return one summary value.
* Ranking functions, which return a ranking value for every row within a partition.
* Rowset functions, which return an object that can be used as a table reference in SQL statements.
* [Scalar](https://whatis.techtarget.com/definition/scalar) functions, which operate on a single value and return a single value.

SQL Server also supports analytical functions in T-SQL to depict complex analytical tasks. These analytical functions enable IT pros to perform common analysis, such as ranking, percentiles, moving averages and cumulative sums to be expressed in a single SQL statement.

### Difference between T-SQL and SQL

There are three distinct differences between the two.

* While T-SQL is an extension to SQL, SQL is a programming language.
* T-SQL contains procedural programming and local variable, while SQL does not.
* T-SQL is proprietary, while SQL is an open format.

### Joins in T-SQL

Joins in T-SQL are clauses used to combine rows from two or more tables, based on a related column between them. Joins specify how SQL should use data from one table to select the rows in another table. Several operators -- such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT -- can be used to join tables.

Different types of joins are available in T-SQL. They include, for example, inner joins and outer joins. An inner join, which returns rows when there is a match in both tables, can be specified in either the FROM or WHERE clauses. Outer joins, which can be specified in the FROM clause only, finds and returns matching data and some dissimilar data from tables.

**Unit- V**

**Procedural Language for SQL (PL/SQL)**

PL/SQL is a block structured language that enables developers to combine the power of SQL with procedural statements.All the statements of a block are passed to oracle engine all at once which increases processing speed and decreases the traffic.

**Disadvantages of SQL:**

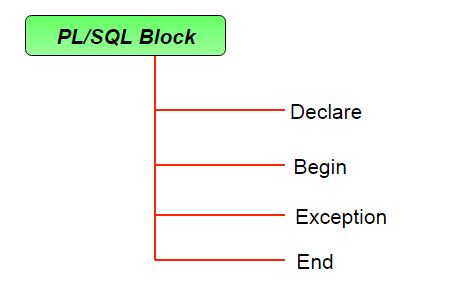
* SQL doesn’t provide the programmers with a technique of condition checking, looping and branching.
* SQL statements are passed to Oracle engine one at a time which increases traffic and decreases speed.
* SQL has no facility of error checking during manipulation of data.

**Features of PL/SQL:**

1. PL/SQL is basically a procedural language, which provides the functionality of decision making, iteration and many more features of procedural programming languages.
2. PL/SQL can execute a number of queries in one block using single command.
3. One can create a PL/SQL unit such as procedures, functions, packages, triggers, and types, which are stored in the database for reuse by applications.
4. PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.
5. Applications written in PL/SQL are portable to computer hardware or operating system where Oracle is operational.
6. PL/SQL Offers extensive error checking.

**Structure of PL/SQL Block:**

PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL. The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other.



Typically, each block performs a logical action in the program. A block has the following structure:

**DECLARE**

declaration statements;

**BEGIN**

executable statements

**EXCEPTIONS**

exception handling statements

**END;**

* Declare section starts with **DECLARE** keyword in which variables, constants, records as cursors can be declared which stores data temporarily. It basically consists definition of PL/SQL identifiers. This part of the code is optional.
* Execution section starts with **BEGIN** and ends with **END** keyword.This is a mandatory section and here the program logic is written to perform any task like loops and conditional statements. It supports all [DML](https://en.wikipedia.org/wiki/Data_manipulation_language) commands, [DDL](https://en.wikipedia.org/wiki/Data_definition_language) commands and SQL\*PLUS built-in functions as well.
* Exception section starts with **EXCEPTION** keyword.This section is optional which contains statements that are executed when a run-time error occurs. Any exceptions can be handled in this section.

# PL/SQL Cursor

When an SQL statement is processed, Oracle creates a memory area known as context area. A cursor is a pointer to this context area. It contains all information needed for processing the statement. In PL/SQL, the context area is controlled by Cursor. A cursor contains information on a select statement and the rows of data accessed by it.

A cursor is used to referred to 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

## 1) PL/SQL Implicit Cursors

The implicit cursors are automatically generated by Oracle while an SQL statement is executed, if we don't use an explicit cursor for the statement.

These are created by default to process the statements when DML statements like INSERT, UPDATE, DELETE etc. are executed.

Orcale provides some attributes known as Implicit cursor's attributes to check the status of DML operations. Some of them are: %FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.

**For example:**When we execute the SQL statements like INSERT, UPDATE, DELETE then the cursor attributes tell whether any rows are affected and how many have been affected. If we run a SELECT INTO statement in PL/SQL block, the implicit cursor attribute can be used to find out whether any row has been returned by the SELECT statement. It will return an error if there no data is selected.

The following table soecifies the status of the cursor with each of its attribute.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| %FOUND | Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect at least one row or more rows or a SELECT INTO statement returned one or more rows. Otherwise it returns FALSE. |
| %NOTFOUND | Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect no row, or a SELECT INTO statement return no rows. Otherwise it returns FALSE. It is a just opposite of %FOUND. |
| %ISOPEN | It always returns FALSE for implicit cursors, because the SQL cursor is automatically closed after executing its associated SQL statements. |
| %ROWCOUNT | It returns the number of rows affected by DML statements like INSERT, DELETE, and UPDATE or returned by a SELECT INTO statement. |

## PL/SQL Implicit Cursor Example

**Create customers table and have records:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 20000 |
| 2 | Suresh | 22 | Kanpur | 22000 |
| 3 | Mahesh | 24 | Ghaziabad | 24000 |
| 4 | Chandan | 25 | Noida | 26000 |
| 5 | Alex | 21 | Paris | 28000 |
| 6 | Sunita | 20 | Delhi | 30000 |

Let's execute the following program to update the table and increase salary of each customer by 5000. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected:

**Create procedure:**

1. **DECLARE**
2. total\_rows number(2);
3. **BEGIN**
4. **UPDATE**  customers
5. **SET** salary = salary + 5000;
6. IF sql%notfound **THEN**
7. dbms\_output.put\_line('no customers updated');
8. ELSIF sql%found **THEN**
9. total\_rows := sql%rowcount;
10. dbms\_output.put\_line( total\_rows || ' customers updated ');
11. **END** IF;
12. **END**;
13. /

Output:

6 customers updated

PL/SQL procedure successfully completed.

Now, if we check the records in customer table, we will find that the rows are updated.

1. **select** \* **from** customers;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 25000 |
| 2 | Suresh | 22 | Kanpur | 27000 |
| 3 | Mahesh | 24 | Ghaziabad | 29000 |
| 4 | Chandan | 25 | Noida | 31000 |
| 5 | Alex | 21 | Paris | 33000 |
| 6 | Sunita | 20 | Delhi | 35000 |

## 2) PL/SQL Explicit Cursors

The Explicit cursors are defined by the programmers to gain more control over the context area. These cursors 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.

Following is the syntax to create an explicit cursor:

## Syntax of explicit cursor

Following is the syntax to create an explicit cursor:

1. **CURSOR** cursor\_name **IS** select\_statement;;

## Steps:

We must follow these steps while working with an explicit cursor.

1. Declare the cursor to initialize in the memory.
2. Open the cursor to allocate memory.
3. Fetch the cursor to retrieve data.
4. Close the cursor to release allocated memory.

## 1) Declare the cursor:

It defines the cursor with a name and the associated SELECT statement.

**Syntax for explicit cursor decleration**

1. **CURSOR** **name** **IS**
2. **SELECT** statement;

## 2) Open the cursor:

It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.

**Syntax for cursor open:**

1. **OPEN** cursor\_name;

## 3) Fetch the cursor:

It is used to access one row at a time. We can fetch rows from the above-opened cursor as follows:

**Syntax for cursor fetch:**

1. **FETCH** cursor\_name **INTO** variable\_list;

## 4) Close the cursor:

It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.

**Syntax for cursor close:**

1. **Close** cursor\_name;

## PL/SQL Explicit Cursor Example

Explicit cursors are defined by programmers to gain more control over the context area. It is defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

Let's take an example to demonstrate the use of explicit cursor. In this example, we are using the already created CUSTOMERS table.

**Create customers table and have records:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 20000 |
| 2 | Suresh | 22 | Kanpur | 22000 |
| 3 | Mahesh | 24 | Ghaziabad | 24000 |
| 4 | Chandan | 25 | Noida | 26000 |
| 5 | Alex | 21 | Paris | 28000 |
| 6 | Sunita | 20 | Delhi | 30000 |

**Create procedure:**

Execute the following program to retrieve the customer name and address.

1. **DECLARE**
2. c\_id customers.id%type;
3. c\_name customers.**name**%type;
4. c\_addr customers.address%type;
5. **CURSOR** c\_customers **is**
6. **SELECT** id, **name**, address **FROM** customers;
7. **BEGIN**
8. **OPEN** c\_customers;
9. LOOP
10. **FETCH** c\_customers **into** c\_id, c\_name, c\_addr;
11. EXIT **WHEN** c\_customers%notfound;
12. dbms\_output.put\_line(c\_id || ' ' || c\_name || ' ' || c\_addr);
13. **END** LOOP;
14. **CLOSE** c\_customers;
15. **END**;
16. /

Output:

1 Ramesh Allahabad

2 Suresh Kanpur

3 Mahesh Ghaziabad

4 Chandan Noida

5 Alex Paris

6 Sunita Delhi

PL/SQL procedure successfully completed.

# PL/SQL Exception Handling

## What is Exception

An error occurs during the program execution is called Exception in PL/SQL.

PL/SQL facilitates programmers to catch such conditions using exception block in the program and an appropriate action is taken against the error condition.

There are two type of exceptions:

* System-defined Exceptions
* User-defined Exceptions

## PL/SQL Exception Handling

**Syntax for exception handling:**

Following is a general syntax for exception handling:

1. **DECLARE**
2. <declarations **section**>
3. **BEGIN**
4. <executable command(s)>
5. EXCEPTION
6. <exception handling goes here >
7. **WHEN** exception1 **THEN**
8. exception1-handling-statements
9. **WHEN** exception2  **THEN**
10. exception2-handling-statements
11. **WHEN** exception3 **THEN**
12. exception3-handling-statements
13. ........
14. **WHEN** others **THEN**
15. exception3-handling-statements
16. **END**;

## Example of exception handling

Let's take a simple example to demonstrate the concept of exception handling. Here we are using the already created CUSTOMERS table.

SELECT\* FROM COUSTOMERS;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 20000 |
| 2 | Suresh | 22 | Kanpur | 22000 |
| 3 | Mahesh | 24 | Ghaziabad | 24000 |
| 4 | Chandan | 25 | Noida | 26000 |
| 5 | Alex | 21 | Paris | 28000 |
| 6 | Sunita | 20 | Delhi | 30000 |

1. **DECLARE**
2. c\_id customers.id%type := 8;
3. c\_name  customers.**name**%type;
4. c\_addr customers.address%type;
5. **BEGIN**
6. **SELECT**  **name**, address **INTO**  c\_name, c\_addr
7. **FROM** customers
8. **WHERE** id = c\_id;
9. DBMS\_OUTPUT.PUT\_LINE ('Name: '||  c\_name);
10. DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);
11. EXCEPTION
12. **WHEN** no\_data\_found **THEN**
13. dbms\_output.put\_line('No such customer!');
14. **WHEN** others **THEN**
15. dbms\_output.put\_line('Error!');
16. **END**;
17. /

After the execution of above code at SQL Prompt, it produces the following result:

No such customer!

PL/SQL procedure successfully completed.

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

#### Note: We get the result "No such customer" because the customer\_id used in the above example is 8 and there is no cutomer having id value 8 in that table.

If we use the id defined in the above table (i.e. 1 to 6), we will get a certain result. For a demo example: here, we are using the id 5.

1. **DECLARE**
2. c\_id customers.id%type := 5;
3. c\_name  customers.**name**%type;
4. c\_addr customers.address%type;
5. **BEGIN**
6. **SELECT**  **name**, address **INTO**  c\_name, c\_addr
7. **FROM** customers
8. **WHERE** id = c\_id;
9. DBMS\_OUTPUT.PUT\_LINE ('Name: '||  c\_name);
10. DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);
11. EXCEPTION
12. **WHEN** no\_data\_found **THEN**
13. dbms\_output.put\_line('No such customer!');
14. **WHEN** others **THEN**
15. dbms\_output.put\_line('Error!');
16. **END**;
17. /

After the execution of above code at SQL prompt, we will get the following result:

Name: alex

Address: paris

PL/SQL procedure successfully completed.

## Raising Exceptions

In the case of any internal database error, exceptions are raised by the database server automatically. But it can also be raised explicitly by programmer by using command RAISE.

**Syntax for raising an exception:**

1. **DECLARE**
2. exception\_name EXCEPTION;
3. **BEGIN**
4. IF condition **THEN**
5. RAISE exception\_name;
6. **END** IF;
7. EXCEPTION
8. **WHEN** exception\_name **THEN**
9. statement;
10. **END**;

## PL/SQL User-defined Exceptions

PL/SQL facilitates their users to define their own exceptions according to the need of the program. A user-defined exception can be raised explicitly, using either a RAISE statement or the procedure DBMS\_STANDARD.RAISE\_APPLICATION\_ERROR.

**Syntax for user define exceptions**

1. **DECLARE**
2. my-exception EXCEPTION;

## PL/SQL Pre-defined Exceptions

There are many pre-defined exception in PL/SQL which are executed when any database rule is violated by the programs.

**For example:**NO\_DATA\_FOUND is a pre-defined exception which is raised when a SELECT INTO statement returns no rows.

Following is a list of some important pre-defined exceptions:

|  |  |  |  |
| --- | --- | --- | --- |
| **Exception** | **Oracle Error** | **SQL Code** | **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" clauses 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 s 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 size-constraint error occurs. |
| ZERO\_DIVIDE | 01476 | 1476 | It is raised when an attempt is made to divide a number by zero. |

# PL/SQL Procedure

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

* **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
* **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

## How to pass parameters in procedure:

When we want to create a procedure or function, we have to define parameters .There is three ways to pass parameters in procedure:

1. **IN parameters:**The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:**The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:**The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

#### A procedure may or may not return any value.

## PL/SQL Create Procedure

**Syntax for creating procedure:**

**CREATE** [OR REPLACE] **PROCEDURE** procedure\_name

    [ (parameter [,parameter]) ]

**IS**

    [declaration\_section]

**BEGIN**

    executable\_section

[EXCEPTION

    exception\_section]

**END** [procedure\_name];

## Create procedure example

In this example, we are going to insert record in user table. So we need to create user table first.

**Table creation:**

**create** **table** user(id number(10) **primary** **key**,**name** varchar2(100));

Now write the procedure code to insert record in user table.

**Procedure Code:**

**create** or replace **procedure** "INSERTUSER"

(id IN NUMBER,

**name** IN VARCHAR2)

**is**

**begin**

**insert** **into** user **values**(id,**name**);

**end**;

/

Output:

Procedure created.

## PL/SQL program to call procedure

Let's see the code to call above created procedure.

**BEGIN**

   insertuser(101,'Rahul');

   dbms\_output.put\_line('record inserted successfully');

**END**;

/

Now, see the "USER" table, we will see one record is inserted.

|  |  |
| --- | --- |
| **ID** | **Name** |
| 101 | Rahul |

## PL/SQL Drop Procedure

**Syntax for drop procedure**

**DROP** **PROCEDURE** procedure\_name;

## Example of drop procedure

**DROP** **PROCEDURE** pro1;

# PL/SQL Function

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

**Syntax to create a function:**

1. **CREATE** [OR REPLACE] **FUNCTION** function\_name [parameters]
2. [(parameter\_name [IN | **OUT** | IN **OUT**] type [, ...])]
3. **RETURN** return\_datatype
4. {**IS** | **AS**}
5. **BEGIN**
6. < function\_body >
7. **END** [function\_name];

**Here:**

* **Function\_name:** specifies the name of the function.
* **[OR REPLACE]** option allows modifying an existing function.
* The **optional parameter list** contains name, mode and types of the parameters.
* **IN** represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.

### The function must contain a return statement.

* RETURN clause specifies that data type we 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.

## PL/SQL Function Example

Let's see a simple example to **create a function**.

1. **create** or replace **function** adder(n1 in number, n2 in number)
2. **return** number
3. **is**
4. n3 number(8);
5. **begin**
6. n3 :=n1+n2;
7. **return** n3;
8. **end**;
9. /

Now write another program to **call the function**.

1. **DECLARE**
2. n3 number(2);
3. **BEGIN**
4. n3 := adder(11,22);
5. dbms\_output.put\_line('Addition is: ' || n3);
6. **END**;
7. /

**Output:**

Addition is: 33

Statement processed.

0.05 seconds

## Another PL/SQL Function Example

Let's take an example to demonstrate Declaring, Defining and Invoking a simple PL/SQL function which will compute and return the maximum of two values.

1. **DECLARE**
2. a number;
3. b number;
4. c number;
5. **FUNCTION** findMax(x IN number, y IN number)
6. **RETURN** number
7. **IS**
8. z number;
9. **BEGIN**
10. IF x > y **THEN**
11. z:= x;
12. **ELSE**
13. Z:= y;
14. **END** IF;
16. **RETURN** z;
17. **END**;
18. **BEGIN**
19. a:= 23;
20. b:= 45;
22. c := findMax(a, b);
23. dbms\_output.put\_line(' Maximum of (23,45): ' || c);
24. **END**;
25. /

**Output:**

Maximum of (23,45): 45

Statement processed.

0.02 seconds

## PL/SQL function example using table

Let's take a customer table. This example illustrates creating and calling a standalone function. This function will return the total number of CUSTOMERS in the customers table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Customers** | | | |
| **Id** | **Name** | **Department** | **Salary** |
| 1 | alex | web developer | 35000 |
| 2 | ricky | program developer | 45000 |
| 3 | mohan | web designer | 35000 |
| 4 | dilshad | database manager | 44000 |

**Create Function:**

**CREATE** OR REPLACE **FUNCTION** totalCustomers

**RETURN** number **IS**

   total number(2) := 0;

**BEGIN**

**SELECT** count(\*) **into** total

**FROM** customers;

**RETURN** total;

**END**;

/

After the execution of above code, we will get the following result.

Function created.

**Calling PL/SQL Function:**

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

After the successful completion of the defined task, the call function returns program control back to the main program.

To call a function we have to pass the required parameters along with function name and if function returns a value then we can store 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**;

/

After the execution of above code in SQL prompt, we will get the following result.

Total no. of Customers: 4

PL/SQL procedure successfully completed.

## PL/SQL Drop Function

**Syntax for removing wer created function:**

If we want to remove we created function from the database, we should use the following syntax.

**DROP** **FUNCTION** function\_name;

# PL/SQL Trigger

Trigger is invoked by Oracle engine automatically whenever a specified event occurs.Trigger is stored into database and invoked repeatedly, when specific condition match.

Triggers are stored programs, which are automatically executed or fired when some event occurs.

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

## Advantages of Triggers

These are the following advantages of Triggers:

* Trigger generates some derived column values automatically
* Enforces referential integrity
* Event logging and storing information on table access
* Auditing
* Synchronous replication of tables
* Imposing security authorizations
* Preventing invalid transactions

## Creating a trigger:

**Syntax for creating trigger:**

**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**;

**Here,**

* CREATE [OR REPLACE] TRIGGER trigger\_name: It creates or replaces an existing trigger with the trigger\_name.
* {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would 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 would 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 we to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
* [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would 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.

## PL/SQL Trigger Example

Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table:

**Create table and have records:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 20000 |
| 2 | Suresh | 22 | Kanpur | 22000 |
| 3 | Mahesh | 24 | Ghaziabad | 24000 |
| 4 | Chandan | 25 | Noida | 26000 |
| 5 | Alex | 21 | Paris | 28000 |
| 6 | Sunita | 20 | Delhi | 30000 |

**Create trigger:**

Let's take a program to create 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**;

/

After the execution of the above code at SQL Prompt, it produces the following result.

Trigger created.

**Check the salary difference by procedure:**

Use the following code to get the old salary, new salary and salary difference after the trigger created.

**DECLARE**

   total\_rows number(2);

**BEGIN**

**UPDATE**  customers

**SET** salary = salary + 5000;

   IF sql%notfound **THEN**

      dbms\_output.put\_line('no customers updated');

   ELSIF sql%found **THEN**

      total\_rows := sql%rowcount;

      dbms\_output.put\_line( total\_rows || ' customers updated ');

**END** IF;

**END**;

/

Output:

Old salary: 20000

New salary: 25000

Salary difference: 5000

Old salary: 22000

New salary: 27000

Salary difference: 5000

Old salary: 24000

New salary: 29000

Salary difference: 5000

Old salary: 26000

New salary: 31000

Salary difference: 5000

Old salary: 28000

New salary: 33000

Salary difference: 5000

Old salary: 30000

New salary: 35000

Salary difference: 5000

6 customers updated

**Note:** As many times we executed this code, the old and new both salary is incremented by 5000 and hence the salary difference is always 5000.

After the execution of above code again, we will get the following result.

Old salary: 25000

New salary: 30000

Salary difference: 5000

Old salary: 27000

New salary: 32000

Salary difference: 5000

Old salary: 29000

New salary: 34000

Salary difference: 5000

Old salary: 31000

New salary: 36000

Salary difference: 5000

Old salary: 33000

New salary: 38000

Salary difference: 5000

Old salary: 35000

New salary: 40000

Salary difference: 5000

6 customers updated

## Important Points

Following are the two very important point and should be noted carefully.

* OLD and NEW references are used for record level triggers these are not avialable for table level triggers.
* If we want to query the table in the same trigger, then we 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.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* E N D \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***