**Module - 3 (RDBMS)**

Introduction To DBMS, SQL,PLSQL

What is Database

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

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

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

Database Management System

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

**DBMS allows users the following tasks:**

* **Data Definition:** It is used for creation, modification, and removal of definition that defines the organization of data in the database.
* **Data Updation:** It is used for the insertion, modification, and deletion of the actual data in the database.
* **Data Retrieval:** It is used to retrieve the data from the database which can be used by applications for various purposes.
* **User Administration:** It is used for registering and monitoring users, maintain data integrity, enforcing data security, dealing with concurrency control, monitoring performance and recovering information corrupted by unexpected failure.

Characteristics of DBMS

* It uses a digital repository established on a server to store and manage the information.
* It can provide a clear and logical view of the process that manipulates data.
* DBMS contains automatic backup and recovery procedures.
* It contains ACID properties which maintain data in a healthy state in case of failure.
* It can reduce the complex relationship between data.
* It is used to support manipulation and processing of data.
* It is used to provide security of data.
* It can view the database from different viewpoints according to the requirements of the user.

Advantages of DBMS

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

Disadvantages of DBMS

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

# DBMS vs. File System

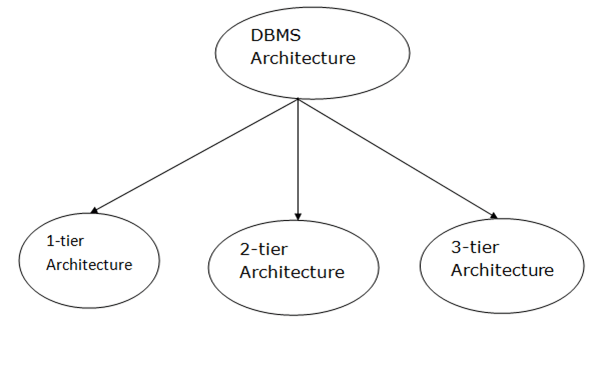
There are following differences between DBMS and File system:

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

# DBMS Architecture

* The DBMS design depends upon its architecture. The basic client/server architecture is used to deal with a large number of PCs, web servers, database servers and other components that are connected with networks.
* The client/server architecture consists of many PCs and a workstation which are connected via the network.
* DBMS architecture depends upon how users are connected to the database to get their request done.

## Types of DBMS Architecture



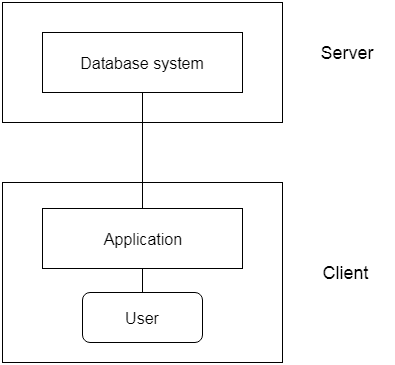
Database architecture can be seen as a single tier or multi-tier. But logically, database architecture is of two types like: **2-tier architecture** and **3-tier architecture**.

### 1-Tier Architecture

* In this architecture, the database is directly available to the user. It means the user can directly sit on the DBMS and uses it.
* Any changes done here will directly be done on the database itself. It doesn't provide a handy tool for end users.
* The 1-Tier architecture is used for development of the local application, where programmers can directly communicate with the database for the quick response.

### 2-Tier Architecture

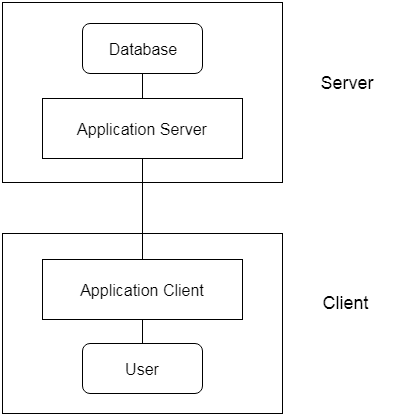
* The 2-Tier architecture is same as basic client-server. In the two-tier architecture, applications on the client end can directly communicate with the database at the server side. For this interaction, API's like: **ODBC**, **JDBC** are used.
* The user interfaces and application programs are run on the client-side.
* The server side is responsible to provide the functionalities like: query processing and transaction management.
* To communicate with the DBMS, client-side application establishes a connection with the server side.



**Fig: 2-tier Architecture**

### 3-Tier Architecture

* The 3-Tier architecture contains another layer between the client and server. In this architecture, client can't directly communicate with the server.
* The application on the client-end interacts with an application server which further communicates with the database system.
* End user has no idea about the existence of the database beyond the application server. The database also has no idea about any other user beyond the application.
* The 3-Tier architecture is used in case of large web application.

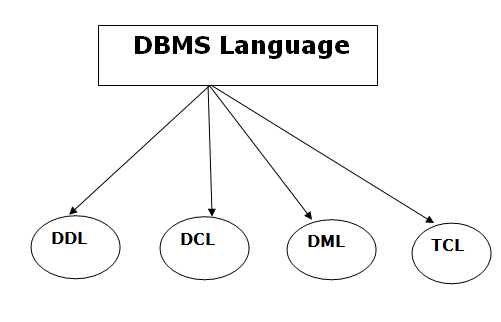


**Fig: 3-tier Architecture**

# Database Language

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

## Types of Database Language



### 1. Data Definition Language

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

Here are some tasks that come under DDL:

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

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

### 2. Data Manipulation Language

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

Here are some tasks that come under DML:

* **Select:** It is used to retrieve data from a database.
* **Insert:** It is used to insert data into a table.
* **Update:** It is used to update existing data within a table.
* **Delete:** It is used to delete all records from a table.
* **Merge:** It performs UPSERT operation, i.e., insert or update operations.
* **Call:** It is used to call a structured query language or a Java subprogram.
* **Explain Plan:** It has the parameter of explaining data.
* **Lock Table:** It controls concurrency.

### 3. Data Control Language

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

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

Here are some tasks that come under DCL:

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

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

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

### 4. Transaction Control Language

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

Here are some tasks that come under TCL:

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

# Relational Model concept

Relational model can represent as a table with columns and rows. Each row is known as a tuple. Each table of the column has a name or attribute.

**Domain:** It contains a set of atomic values that an attribute can take.

**Attribute:** It contains the name of a column in a particular table. Each attribute Ai must have a domain, dom(Ai)

**Relational instance:** In the relational database system, the relational instance is represented by a finite set of tuples. Relation instances do not have duplicate tuples.

**Relational schema:** A relational schema contains the name of the relation and name of all columns or attributes.

**Relational key:** In the relational key, each row has one or more attributes. It can identify the row in the relation uniquely.

**Example: STUDENT Relation**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **ROLL\_NO** | **PHONE\_NO** | **ADDRESS** | **AGE** |
| Ram | 14795 | 7305758992 | Noida | 24 |
| Shyam | 12839 | 9026288936 | Delhi | 35 |
| Laxman | 33289 | 8583287182 | Gurugram | 20 |
| Mahesh | 27857 | 7086819134 | Ghaziabad | 27 |
| Ganesh | 17282 | 9028 9i3988 | Delhi | 40 |

* In the given table, NAME, ROLL\_NO, PHONE\_NO, ADDRESS, and AGE are the attributes.
* The instance of schema STUDENT has 5 tuples.
* t3 = <Laxman, 33289, 8583287182, Gurugram, 20>

## Properties of Relations

* Name of the relation is distinct from all other relations.
* Each relation cell contains exactly one atomic (single) value
* Each attribute contains a distinct name
* Attribute domain has no significance
* tuple has no duplicate value
* Order of tuple can have a different sequence

**Relational Data Model**

The relational model is the theoretical basis of relational databases which is a technique or way of structuring data using relations, which are grid-like mathematical structures consisting of columns and rows. Codd proposed the relational model for IBM, but the idea became extremely vital and prominent that his work would become the basis of relational databases. You might be very familiar with the physical demonstration of a relation in a database - which is known as a table.

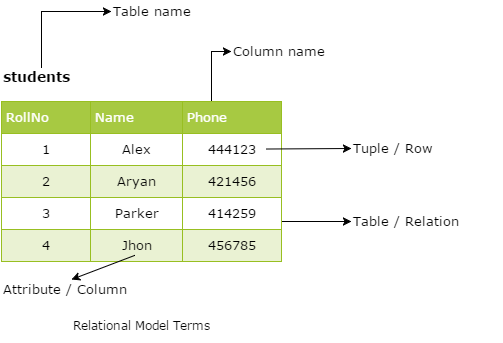
In the relational model, all data is logically structured within relations, i.e., tables, as mentioned above. Each relation has a name and is formed from named attributes or columns of data. Each tuple or row holds one value per attribute. The greatest strength of the relational model is this simple logical structure that it forms. Behind this simple structure is a sophisticated theoretical foundation that is lacking in the first generation of DBMSs.

**Database Schema**

When you talk about the database, you must distinguish between the database schema, which is the logical blueprint of the database, and the database instance, which is a snapshot of the data in the database at a given instant in time. The concept of a relation corresponds to the programming language notion of a variable, while the concept of a relation schema corresponds to the programming languages' notion of the type definition. In other words, a database schema is a skeletal structure which represents the logical view of the complete database. It describes how the data is organized and how the relations among them are associated and formulates all the constraints that are to be applied to the data.

In general, a relation schema consists of a directory of attributes and their corresponding domain.

### Some Common Relational Model Terms

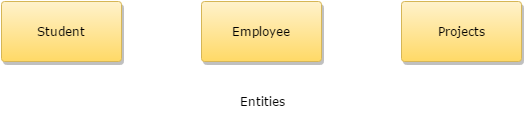


* **Relation:** A relation is a table with columns and rows.
* **Attribute:** An attribute is a named column of a relation.
* **Domain:** A domain is the set of allowable values for one or more attributes.
* **Tuple:** A tuple is a row of a relation.

**ER-Model**

ER-Diagram is a pictorial representation of data that describes how data is communicated and related to each other. Any object, such as entities, attributes of an entity, sets of relationship and other attributes of relationship can be characterized with the help of the ER diagram.

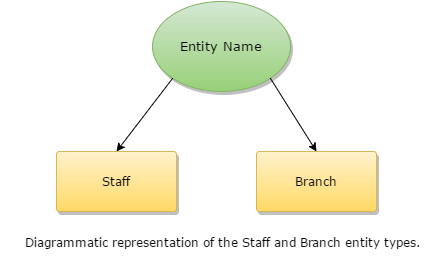
Entities: They are represented using the rectangle shape box. These rectangles are named with the entity set they represent.



ER modeling is a top-down structure to database design that begins with identifying the important data called entities and relationships in combination with the data that must be characterized in the model. Then database model designers can add more details such as the information they want to hold about the entities and relationships which are the attributes and any constraints on the entities, relationships, and attributes. ER modeling is an important technique for any database designer to master and forms the basis of the methodology.

* Entity type: It is a group of objects with the same properties that are identified by the enterprise as having an independent existence. The basic concept of the ER model is the entity type that is used to represent a group of 'objects' in the 'real world' with the same properties. An entity type has an independent existence within a database.
* Entity occurrence: A uniquely identifiable object of an entity type.

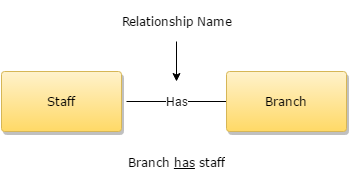
Each entity type is shown as a rectangle labeled with the name of the entity, which is normally a singular noun.



### What is Relationship Type?

A relationship type is a set of associations between one or more participating entity types. Each relationship type is given a name that describes its function.

Here is a diagram showing how relationships are formed in a database.



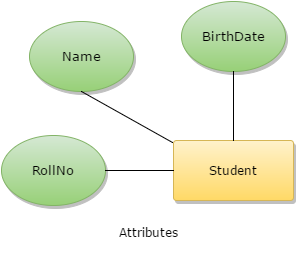
### What is a degree of Relationship?

The entities occupied in a particular relationship type are referred to as participants in that relationship. The number of participants involved in a relationship type is termed as the degree of that relationship.

In the above-figured example "Branch has staff", there is a relationship between two participating entities. A relationship of degree two is called binary degree (relationship).

### What are Attributes?

Attributes are the properties of entities that are represented using ellipse shaped figures. Every elliptical figure represents one attribute and is directly connected to its entity (which is represented as a rectangle).

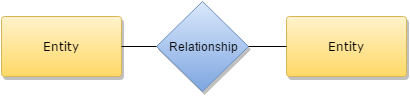


It is to be noted that multi-valued attributes are represented using double ellipse like this:



### Relationships

A diamond-shaped box represents relationships. All the entities (rectangle shaped) participating in a relationship gets connected using a line.



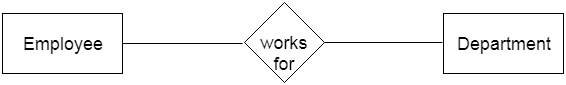
There are four types of relationships. These are:

* **One-to-one:** When only a single instance of an entity is associated with the relationship, it is termed as '1:1'.
* **One-to-many:**When more than one instance of an entity is related and linked with a relationship, it is termed as '1:N'.
* **Many-to-one:** When more than one instance of an entity is linked with the relationship, it is termed as 'N:1'.
* **Many-to-many:**When more than one instance of an entity on the left and more than one instance of an entity on the right can be linked with the relationship, then it is termed as N:N relationship.

### 1. Entity:

An entity may be any object, class, person or place. In the ER diagram, an entity can be represented as rectangles.

Consider an organization as an example- manager, product, employee, department etc. can be taken as an entity.



**a. Weak Entity**

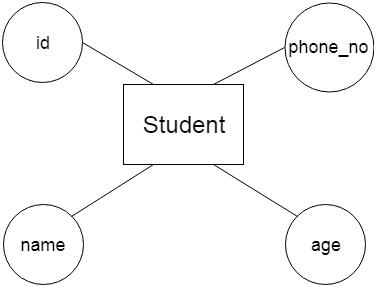
An entity that depends on another entity called a weak entity. The weak entity doesn't contain any key attribute of its own. The weak entity is represented by a double rectangle.



### 2. Attribute

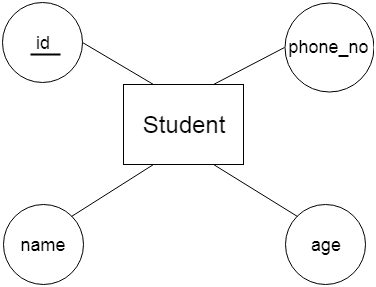
The attribute is used to describe the property of an entity. Eclipse is used to represent an attribute.

**For example,** id, age, contact number, name, etc. can be attributes of a student.



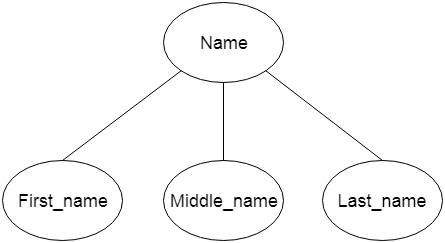
**a. Key Attribute**

The key attribute is used to represent the main characteristics of an entity. It represents a primary key. The key attribute is represented by an ellipse with the text underlined.



**b. Composite Attribute**

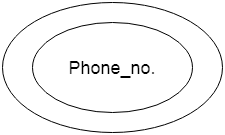
An attribute that composed of many other attributes is known as a composite attribute. The composite attribute is represented by an ellipse, and those ellipses are connected with an ellipse.



**c. Multivalued Attribute**

An attribute can have more than one value. These attributes are known as a multivalued attribute. The double oval is used to represent multivalued attribute.

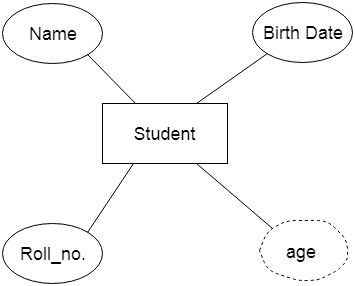
**For example,** a student can have more than one phone number.



**d. Derived Attribute**

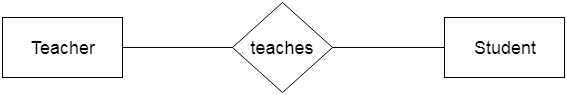
An attribute that can be derived from other attribute is known as a derived attribute. It can be represented by a dashed ellipse.

**For example,** A person's age changes over time and can be derived from another attribute like Date of birth.



### 3. Relationship

A relationship is used to describe the relation between entities. Diamond or rhombus is used to represent the relationship.

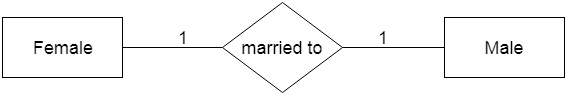


Types of relationship are as follows:

**a. One-to-One Relationship**

When only one instance of an entity is associated with the relationship, then it is known as one to one relationship.

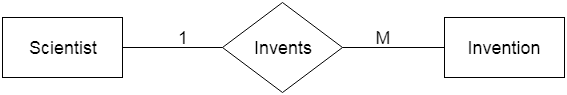
**For example,** A female can marry to one male, and a male can marry to one female.



**b. One-to-many relationship**

When only one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then this is known as a one-to-many relationship.

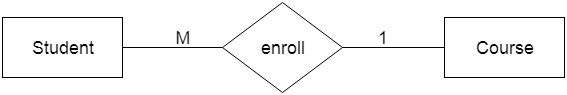
**For example,** Scientist can invent many inventions, but the invention is done by the only specific scientist.



**c. Many-to-one relationship**

When more than one instance of the entity on the left, and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship.

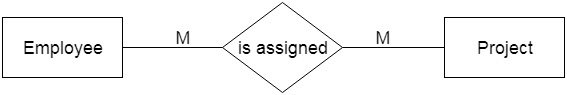
**For example,** Student enrolls for only one course, but a course can have many students.



**d. Many-to-many relationship**

When more than one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship.

**For example,** Employee can assign by many projects and project can have many employees.

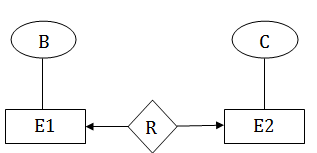


# Mapping Constraints

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

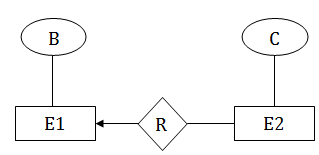
## One-to-one

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



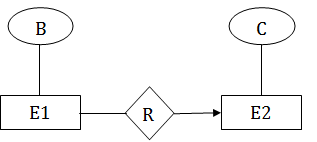
## One-to-many

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



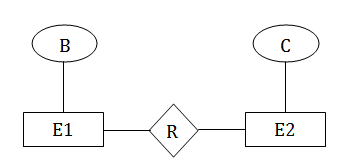
## Many-to-one

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



## Many-to-many

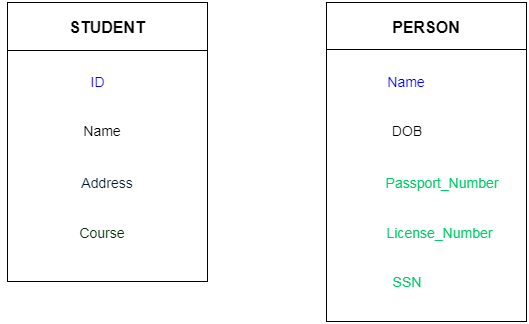
In many-to-many mapping, an entity in E1 is associated with any number of entities in E2, and an entity in E2 is associated with any number of entities in E1.



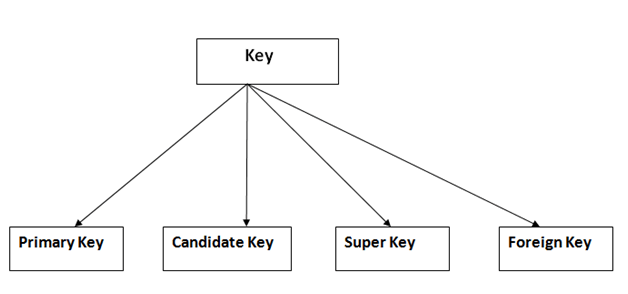
# Keys

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

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

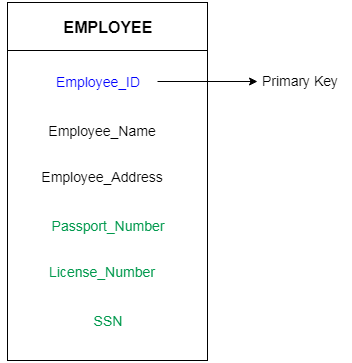


## Types of key:



### 1. Primary key

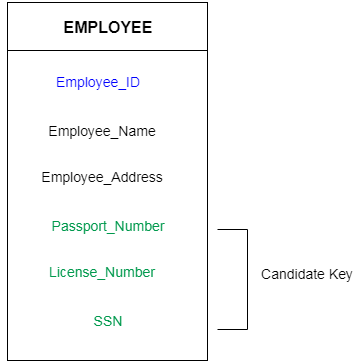
* It is the first key which is used to identify one and only one instance of an entity uniquely. An entity can contain multiple keys as we saw in PERSON table. The key which is most suitable from those lists become a primary key.
* In the EMPLOYEE table, ID can be primary key since it is unique for each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



### 2. Candidate key

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

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



### 3. Super Key

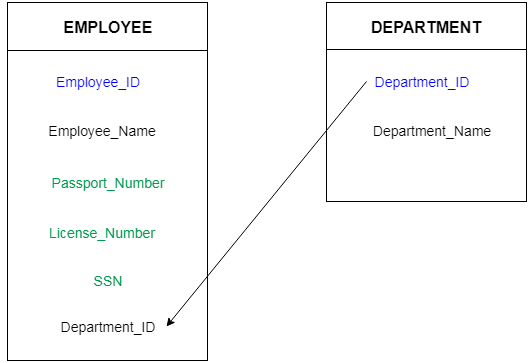
Super key is a set of an attribute which can uniquely identify a tuple. Super key is a superset of a candidate key.

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

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

### 4. Foreign key

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

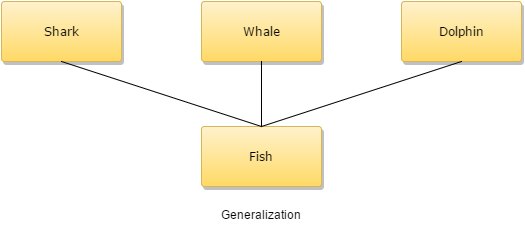


**What is Generalization / Specialization ?**

The concept of generalization (specialization) is associated with special types of entities known as superclasses and subclasses, and the process of attribute inheritance. Database managers begin this section by defining what superclasses and subclasses are and by examining superclass/subclass relationships. The ER Model has the capability of articulating database entities in a conceptual hierarchical manner. As the hierarchy goes up, it generalizes the view of entities, and as you go deep in the hierarchy, it will provide with the detail of every entity included. Going up in this structure is called generalization, where entities are associated together to represent a more generalized view. A generalization is a bottom-up approach.

In generalization, some entities are accommodated together into one generalized entity or category based on their similar characteristics. In the below-mentioned figure, whale, shark, and dolphin are generalized as fish, i.e., they have been categorized as the fish.

* Super-class: An entity type that includes one or more dissimilar sub-groupings of its occurrences that is required to be represented in a data model.
* Sub-class: A distinct sub-grouping of occurrences of an entity type that require being represented in a data model.



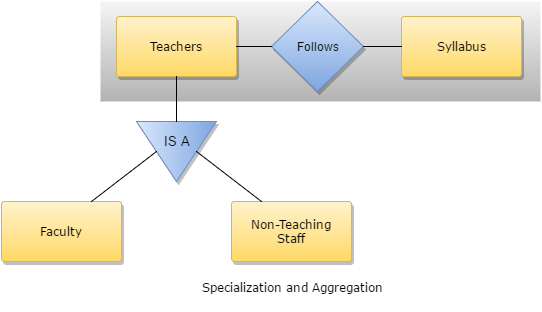
Super-class/Subclass Relationships

Each member of a subclass is also a member of the superclass, i.e., the entity in the subclass is the same entity in the superclass, but has a different role. The relationship between a superclass and a subclass is one-to-one (1:1) and is termed as a super-class/sub-class relationship.

What is Aggregation?

A relationship represents a connection between two entity types that are conceptually at the same level. Sometimes you may want to model a 'has-a,' 'is-a' or 'is-part-of' relationship, in which one entity represents a larger entity (the 'whole') that will consist of smaller entities (the 'parts'). This special kind of relationship is termed as an aggregation. Aggregation does not change the meaning of navigation and routing across the relationship between the whole and its parts.

An example of aggregation is the 'Teacher' entity following the 'syllabus' entity act as a single entity in the relationship. In simple words, aggregation is a process where the relation between two entities is treated as a single entity.



# Join Operations:

A Join operation combines related tuples from different relations, if and only if a given join condition is satisfied. It is denoted by ⋈.

### Example:

**EMPLOYEE**

|  |  |
| --- | --- |
| **EMP\_CODE** | **EMP\_NAME** |
| 101 | Stephan |
| 102 | Jack |
| 103 | Harry |

**SALARY**

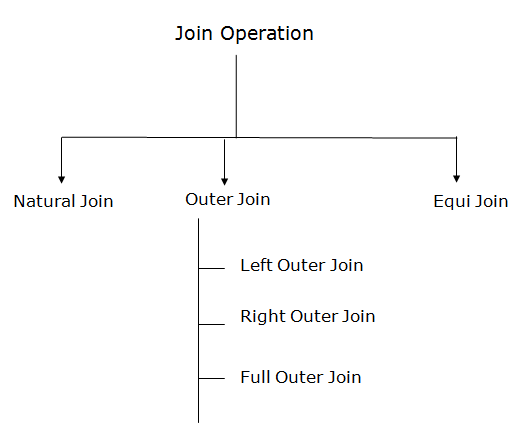
|  |  |
| --- | --- |
| **EMP\_CODE** | **SALARY** |
| 101 | 50000 |
| 102 | 30000 |
| 103 | 25000 |

1. Operation: (EMPLOYEE ⋈ SALARY)

**Result:**

|  |  |  |
| --- | --- | --- |
| **EMP\_CODE** | **EMP\_NAME** | **SALARY** |
| 101 | Stephan | 50000 |
| 102 | Jack | 30000 |
| 103 | Harry | 25000 |

## Types of Join operations:



### 1. Natural Join:

* A natural join is the set of tuples of all combinations in R and S that are equal on their common attribute names.
* It is denoted by ⋈.

**Example:** Let's use the above EMPLOYEE table and SALARY table:

**Input:**

1. ∏EMP\_NAME, SALARY (EMPLOYEE ⋈ SALARY)

**Output:**

|  |  |
| --- | --- |
| **EMP\_NAME** | **SALARY** |
| Stephan | 50000 |
| Jack | 30000 |
| Harry | 25000 |

### 2. Outer Join:

The outer join operation is an extension of the join operation. It is used to deal with missing information.

**Example:**

**EMPLOYEE**

|  |  |  |
| --- | --- | --- |
| **EMP\_NAME** | **STREET** | **CITY** |
| Ram | Civil line | Mumbai |
| Shyam | Park street | Kolkata |
| Ravi | M.G. Street | Delhi |
| Hari | Nehru nagar | Hyderabad |

**FACT\_WORKERS**

|  |  |  |
| --- | --- | --- |
| **EMP\_NAME** | **BRANCH** | **SALARY** |
| Ram | Infosys | 10000 |
| Shyam | Wipro | 20000 |
| Kuber | HCL | 30000 |
| Hari | TCS | 50000 |

**Input:**

1. (EMPLOYEE ⋈ FACT\_WORKERS)

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_NAME** | **STREET** | **CITY** | **BRANCH** | **SALARY** |
| Ram | Civil line | Mumbai | Infosys | 10000 |
| Shyam | Park street | Kolkata | Wipro | 20000 |
| Hari | Nehru nagar | Hyderabad | TCS | 50000 |

An outer join is basically of three types:

1. Left outer join
2. Right outer join
3. Full outer join

### a. Left outer join:

* Left outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.
* In the left outer join, tuples in R have no matching tuples in S.
* It is denoted by ⟕.

**Example:** Using the above EMPLOYEE table and FACT\_WORKERS table

**Input:**

1. EMPLOYEE ⟕ FACT\_WORKERS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_NAME** | **STREET** | **CITY** | **BRANCH** | **SALARY** |
| Ram | Civil line | Mumbai | Infosys | 10000 |
| Shyam | Park street | Kolkata | Wipro | 20000 |
| Hari | Nehru street | Hyderabad | TCS | 50000 |
| Ravi | M.G. Street | Delhi | NULL | NULL |

### b. Right outer join:

* Right outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.
* In right outer join, tuples in S have no matching tuples in R.
* It is denoted by ⟖.

**Example:** Using the above EMPLOYEE table and FACT\_WORKERS Relation

**Input:**

1. EMPLOYEE ⟖ FACT\_WORKERS

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_NAME** | **BRANCH** | **SALARY** | **STREET** | **CITY** |
| Ram | Infosys | 10000 | Civil line | Mumbai |
| Shyam | Wipro | 20000 | Park street | Kolkata |
| Hari | TCS | 50000 | Nehru street | Hyderabad |
| Kuber | HCL | 30000 | NULL | NULL |

### c. Full outer join:

* Full outer join is like a left or right join except that it contains all rows from both tables.
* In full outer join, tuples in R that have no matching tuples in S and tuples in S that have no matching tuples in R in their common attribute name.
* It is denoted by ⟗.

**Example:** Using the above EMPLOYEE table and FACT\_WORKERS table

**Input:**

1. EMPLOYEE ⟗ FACT\_WORKERS

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_NAME** | **STREET** | **CITY** | **BRANCH** | **SALARY** |
| Ram | Civil line | Mumbai | Infosys | 10000 |
| Shyam | Park street | Kolkata | Wipro | 20000 |
| Hari | Nehru street | Hyderabad | TCS | 50000 |
| Ravi | M.G. Street | Delhi | NULL | NULL |
| Kuber | NULL | NULL | HCL | 30000 |

### 3. Equi join:

It is also known as an inner join. It is the most common join. It is based on matched data as per the equality condition. The equi join uses the comparison operator(=).

**Example:**

**CUSTOMER RELATION**

|  |  |
| --- | --- |
| **CLASS\_ID** | **NAME** |
| 1 | John |
| 2 | Harry |
| 3 | Jackson |

**PRODUCT**

|  |  |
| --- | --- |
| **PRODUCT\_ID** | **CITY** |
| 1 | Delhi |
| 2 | Mumbai |
| 3 | Noida |

**Input:**

1. CUSTOMER ⋈ PRODUCT

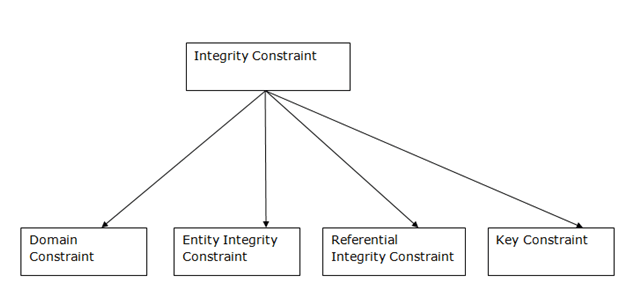
**Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| **CLASS\_ID** | **NAME** | **PRODUCT\_ID** | **CITY** |
| 1 | John | 1 | Delhi |
| 2 | Harry | 2 | Mumbai |
| 3 | Harry | 3 | Noida |

# Integrity Constraints

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

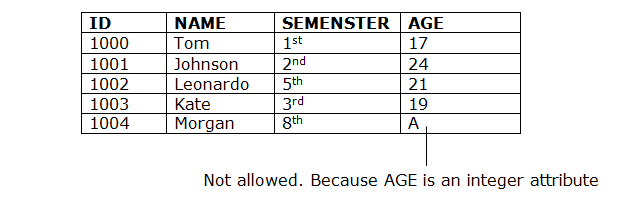
## Types of Integrity Constraint



### 1. Domain constraints

* Domain constraints can be defined as the definition of a valid set of values for an attribute.
* The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

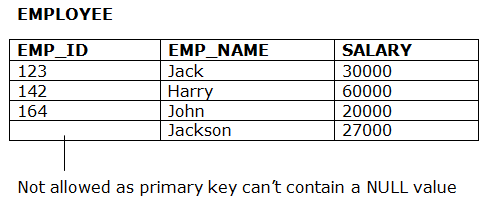
**Example:**



### 2. Entity integrity constraints

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

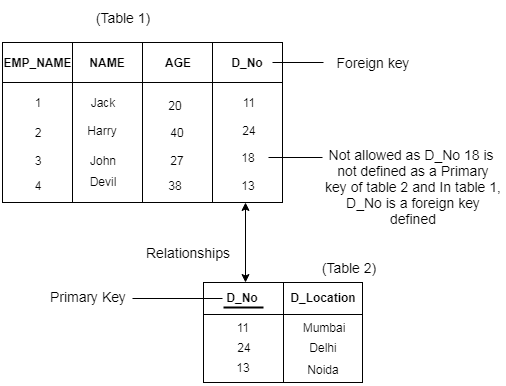
**Example:**



### 3. Referential Integrity Constraints

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

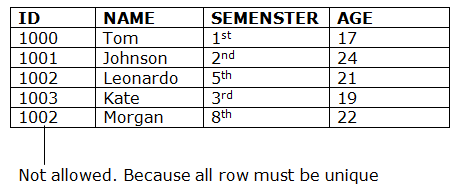
**Example:**



### 4. Key constraints

* Keys are the entity set that is used to identify an entity within its entity set uniquely.
* An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

**Example:**



# Functional Dependency

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

1. X   →   Y

The left side of FD is known as a determinant, the right side of the production is known as a dependent.

**For example:**

Assume we have an employee table with attributes: Emp\_Id, Emp\_Name, Emp\_Address.

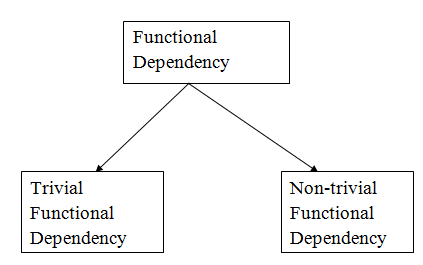
Here Emp\_Id attribute can uniquely identify the Emp\_Name attribute of employee table because if we know the Emp\_Id, we can tell that employee name associated with it.

Functional dependency can be written as:

1. Emp\_Id → Emp\_Name

We can say that Emp\_Name is functionally dependent on Emp\_Id.

## Types of Functional dependency



### 1. Trivial functional dependency

* A → B has trivial functional dependency if B is a subset of A.
* The following dependencies are also trivial like: A → A, B → B

**Example:**

1. Consider a table with two columns Employee\_Id and Employee\_Name.
2. {Employee\_id, Employee\_Name}   →    Employee\_Id is a trivial functional dependency as
3. Employee\_Id is a subset of {Employee\_Id, Employee\_Name}.
4. Also, Employee\_Id → Employee\_Id and Employee\_Name   →    Employee\_Name are trivial dependencies too.

### 2. Non-trivial functional dependency

* A → B has a non-trivial functional dependency if B is not a subset of A.
* When A intersection B is NULL, then A → B is called as complete non-trivial.

**Example:**

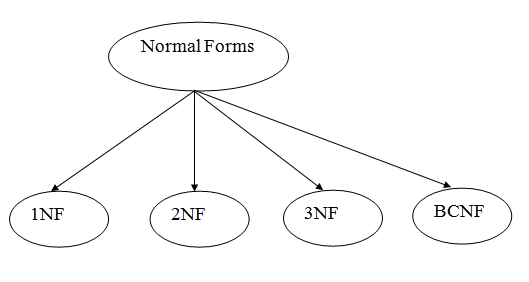
1. ID   →    Name,
2. Name   →    DOB

# Normalization

* Normalization is the process of organizing the data in the database.
* Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
* Normalization divides the larger table into the smaller table and links them using relationship.
* The normal form is used to reduce redundancy from the database table.

## Types of Normal Forms

There are the four types of normal forms:



|  |  |
| --- | --- |
| **Normal Form** | **Description** |
| [1NF](https://www.javatpoint.com/dbms-first-normal-form) | A relation is in 1NF if it contains an atomic value. |
| [2NF](https://www.javatpoint.com/dbms-second-normal-form) | A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key. |
| [3NF](https://www.javatpoint.com/dbms-third-normal-form) | A relation will be in 3NF if it is in 2NF and no transition dependency exists. |
| [4NF](https://www.javatpoint.com/dbms-forth-normal-form) | A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency. |
| [5NF](https://www.javatpoint.com/dbms-fifth-normal-form) | A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless. |

# First Normal Form (1NF)

* A relation will be 1NF if it contains an atomic value.
* It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
* First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

**Example:** Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP\_PHONE.

**EMPLOYEE table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_PHONE** | **EMP\_STATE** |
| 14 | John | 7272826385, 9064738238 | UP |
| 20 | Harry | 8574783832 | Bihar |
| 12 | Sam | 7390372389, 8589830302 | Punjab |

The decomposition of the EMPLOYEE table into 1NF has been shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_PHONE** | **EMP\_STATE** |
| 14 | John | 7272826385 | UP |
| 14 | John | 9064738238 | UP |
| 20 | Harry | 8574783832 | Bihar |
| 12 | Sam | 7390372389 | Punjab |
| 12 | Sam | 8589830302 | Punjab |

# Second Normal Form (2NF)

* In the 2NF, relational must be in 1NF.
* In the second normal form, all non-key attributes are fully functional dependent on the primary key

**Example:** Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

**TEACHER table**

|  |  |  |
| --- | --- | --- |
| **TEACHER\_ID** | **SUBJECT** | **TEACHER\_AGE** |
| 25 | Chemistry | 30 |
| 25 | Biology | 30 |
| 47 | English | 35 |
| 83 | Math | 38 |
| 83 | Computer | 38 |

In the given table, non-prime attribute TEACHER\_AGE is dependent on TEACHER\_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

**TEACHER\_DETAIL table:**

|  |  |
| --- | --- |
| **TEACHER\_ID** | **TEACHER\_AGE** |
| 25 | 30 |
| 47 | 35 |
| 83 | 38 |

**TEACHER\_SUBJECT table:**

|  |  |
| --- | --- |
| **TEACHER\_ID** | **SUBJECT** |
| 25 | Chemistry |
| 25 | Biology |
| 47 | English |
| 83 | Math |
| 83 | Computer |

# Third Normal Form (3NF)

* A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
* 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
* If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

1. X is a super key.
2. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

**Example:**

**EMPLOYEE\_DETAIL table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_ZIP** | **EMP\_STATE** | **EMP\_CITY** |
| 222 | Harry | 201010 | UP | Noida |
| 333 | Stephan | 02228 | US | Boston |
| 444 | Lan | 60007 | US | Chicago |
| 555 | Katharine | 06389 | UK | Norwich |
| 666 | John | 462007 | MP | Bhopal |

**Super key in the table above:**

* 1. {EMP\_ID}, {EMP\_ID, EMP\_NAME}, {EMP\_ID, EMP\_NAME, EMP\_ZIP}....so on

**Candidate key:** {EMP\_ID}

**Non-prime attributes:** In the given table, all attributes except EMP\_ID are non-prime.

Here, EMP\_STATE & EMP\_CITY dependent on EMP\_ZIP and EMP\_ZIP dependent on EMP\_ID. The non-prime attributes (EMP\_STATE, EMP\_CITY) transitively dependent on super key(EMP\_ID). It violates the rule of third normal form.

That's why we need to move the EMP\_CITY and EMP\_STATE to the new <EMPLOYEE\_ZIP> table, with EMP\_ZIP as a Primary key.

**EMPLOYEE table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_ZIP** |
| 222 | Harry | 201010 |
| 333 | Stephan | 02228 |
| 444 | Lan | 60007 |
| 555 | Katharine | 06389 |
| 666 | John | 462007 |

**EMPLOYEE\_ZIP table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_ZIP** | **EMP\_STATE** | **EMP\_CITY** |
| 201010 | UP | Noida |
| 02228 | US | Boston |
| 60007 | US | Chicago |
| 06389 | UK | Norwich |
| 462007 | MP | Bhopal |

# Boyce Codd normal form (BCNF)

* BCNF is the advance version of 3NF. It is stricter than 3NF.
* A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
* For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

**Example:** Let's assume there is a company where employees work in more than one department.

**EMPLOYEE table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** | **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| 264 | India | Designing | D394 | 283 |
| 264 | India | Testing | D394 | 300 |
| 364 | UK | Stores | D283 | 232 |
| 364 | UK | Developing | D283 | 549 |

**In the above table Functional dependencies are as follows:**

1. EMP\_ID  →  EMP\_COUNTRY
2. EMP\_DEPT  →   {DEPT\_TYPE, EMP\_DEPT\_NO}

**Candidate key: {EMP-ID, EMP-DEPT}**

The table is not in BCNF because neither EMP\_DEPT nor EMP\_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

**EMP\_COUNTRY table:**

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** |
| 264 | India |
| 264 | India |

**EMP\_DEPT table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| Designing | D394 | 283 |
| Testing | D394 | 300 |
| Stores | D283 | 232 |
| Developing | D283 | 549 |

**EMP\_DEPT\_MAPPING table:**

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_DEPT** |
| D394 | 283 |
| D394 | 300 |
| D283 | 232 |
| D283 | 549 |

**Functional dependencies:**

1. EMP\_ID   →    EMP\_COUNTRY
2. EMP\_DEPT   →   {DEPT\_TYPE, EMP\_DEPT\_NO}

**Candidate keys:**

**For the first table:** EMP\_ID  
**For the second table:** EMP\_DEPT  
**For the third table:** {EMP\_ID, EMP\_DEPT}

Now, this is in BCNF because left side part of both the functional dependencies is a key.

# SQL Tutorial

Top of Form

Bottom of Form

**SQL** is a database computer language designed for the retrieval and management of data in a relational database. **SQL** stands for **Structured Query Language**. This tutorial will give you a quick start to SQL. It covers most of the topics required for a basic understanding of SQL and to get a feel of how it works.

## Why to Learn SQL?

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in a relational database.

SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server use SQL as their standard database language.

Also, they are using different dialects, such as −

* MS SQL Server using T-SQL,
* Oracle using PL/SQL,
* MS Access version of SQL is called JET SQL (native format) etc.

## Applications of SQL

As mentioned before, SQL is one of the most widely used query language over the databases. I'm going to list few of them here:

* Allows users to access data in the relational database management systems.
* Allows users to describe the data.
* Allows users to define the data in a database and manipulate that data.
* Allows to embed within other languages using SQL modules, libraries & pre-compilers.
* Allows users to create and drop databases and tables.
* Allows users to create view, stored procedure, functions in a database.
* Allows users to set permissions on tables, procedures and views.

## What is SQL?

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in a relational database.

SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server use SQL as their standard database language.

Also, they are using different dialects, such as −

* MS SQL Server using T-SQL,
* Oracle using PL/SQL,
* MS Access version of SQL is called JET SQL (native format) etc.

## Why SQL?

SQL is widely popular because it offers the following advantages −

* Allows users to access data in the relational database management systems.
* Allows users to describe the data.
* Allows users to define the data in a database and manipulate that data.
* Allows to embed within other languages using SQL modules, libraries & pre-compilers.
* Allows users to create and drop databases and tables.
* Allows users to create view, stored procedure, functions in a database.
* Allows users to set permissions on tables, procedures and views.

## A Brief History of SQL

* **1970** − Dr. Edgar F. "Ted" Codd of IBM is known as the father of relational databases. He described a relational model for databases.
* **1974** − Structured Query Language appeared.
* **1978** − IBM worked to develop Codd's ideas and released a product named System/R.
* **1986** − IBM developed the first prototype of relational database and standardized by ANSI. The first relational database was released by Relational Software which later came to be known as Oracle.

## SQL Process

When you are executing an SQL command for any RDBMS, the system determines the best way to carry out your request and SQL engine figures out how to interpret the task.

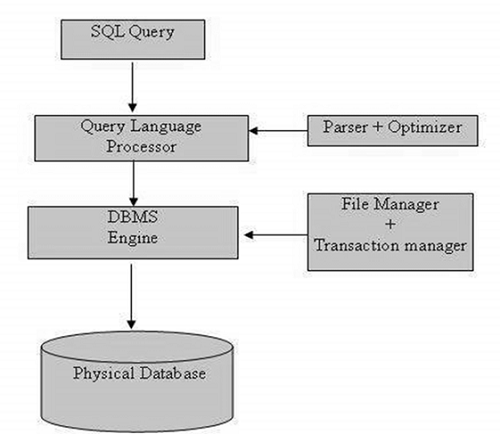
There are various components included in this process.

These components are −

* Query Dispatcher
* Optimization Engines
* Classic Query Engine
* SQL Query Engine, etc.

A classic query engine handles all the non-SQL queries, but a SQL query engine won't handle logical files.

Following is a simple diagram showing the SQL Architecture −



## SQL Commands

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified into the following groups based on their nature −

### DDL - Data Definition Language

|  |  |
| --- | --- |
| **Sr.No.** | **Command & Description** |
| 1 | **CREATE**  Creates a new table, a view of a table, or other object in the database. |
| 2 | **ALTER**  Modifies an existing database object, such as a table. |
| 3 | **DROP**  Deletes an entire table, a view of a table or other objects in the database. |

### DML - Data Manipulation Language

|  |  |
| --- | --- |
| **Sr.No.** | **Command & Description** |
| 1 | **SELECT**  Retrieves certain records from one or more tables. |
| 2 | **INSERT**  Creates a record. |
| 3 | **UPDATE**  Modifies records. |
| 4 | **DELETE**  Deletes records. |

### DCL - Data Control Language

|  |  |
| --- | --- |
| **Sr.No.** | **Command & Description** |
| 1 | **GRANT**  Gives a privilege to user. |
| 2 | **REVOKE**  Takes back privileges granted from user. |

## What is RDBMS?

RDBMS stands for **R**elational **D**atabase **M**anagement **S**ystem. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

A Relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd.

## What is a table?

The data in an RDBMS is stored in database objects which are called as **tables**. This table is basically a collection of related data entries and it consists of numerous columns and rows.

Remember, a table is the most common and simplest form of data storage in a relational database. The following program is an example of a CUSTOMERS table −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

## What is a field?

Every table is broken up into smaller entities called fields. The fields in the CUSTOMERS table consist of ID, NAME, AGE, ADDRESS and SALARY.

A field is a column in a table that is designed to maintain specific information about every record in the table.

## What is a Record or a Row?

A record is also called as a row of data is each individual entry that exists in a table. For example, there are 7 records in the above CUSTOMERS table. Following is a single row of data or record in the CUSTOMERS table −

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

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

A record is a horizontal entity in a table.

## What is a column?

A column is a vertical entity in a table that contains all information associated with a specific field in a table.

For example, a column in the CUSTOMERS table is ADDRESS, which represents location description and would be as shown below −

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

| ADDRESS |

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

| Ahmedabad |

| Delhi |

| Kota |

| Mumbai |

| Bhopal |

| MP |

| Indore |

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

## What is a NULL value?

A NULL value in a table is a value in a field that appears to be blank, which means a field with a NULL value is a field with no value.

It is very important to understand that a NULL value is different than a zero value or a field that contains spaces. A field with a NULL value is the one that has been left blank during a record creation.

## SQL Constraints

Constraints are the rules enforced on data columns on a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints can either be column level or table level. Column level constraints are applied only to one column whereas, table level constraints are applied to the entire table.

Following are some of the most commonly used constraints available in SQL −

* [NOT NULL Constraint](https://www.tutorialspoint.com/sql/sql-not-null.htm) − Ensures that a column cannot have a NULL value.
* [DEFAULT Constraint](https://www.tutorialspoint.com/sql/sql-default.htm) − Provides a default value for a column when none is specified.
* [UNIQUE Constraint](https://www.tutorialspoint.com/sql/sql-unique.htm) − Ensures that all the values in a column are different.
* [PRIMARY Key](https://www.tutorialspoint.com/sql/sql-primary-key.htm) − Uniquely identifies each row/record in a database table.
* [FOREIGN Key](https://www.tutorialspoint.com/sql/sql-foreign-key.htm) − Uniquely identifies a row/record in any another database table.
* [CHECK Constraint](https://www.tutorialspoint.com/sql/sql-check.htm) − The CHECK constraint ensures that all values in a column satisfy certain conditions.
* [INDEX](https://www.tutorialspoint.com/sql/sql-index.htm) − Used to create and retrieve data from the database very quickly.

## Data Integrity

The following categories of data integrity exist with each RDBMS −

* **Entity Integrity −** There are no duplicate rows in a table.
* **Domain Integrity −** Enforces valid entries for a given column by restricting the type, the format, or the range of values.
* **Referential integrity −** Rows cannot be deleted, which are used by other records.
* **User-Defined Integrity −** Enforces some specific business rules that do not fall into entity, domain or referential integrity.

## Database Normalization

Database normalization is the process of efficiently organizing data in a database. There are two reasons of this normalization process −

* Eliminating redundant data, for example, storing the same data in more than one table.
* Ensuring data dependencies make sense.

Both these reasons are worthy goals as they reduce the amount of space a database consumes and ensures that data is logically stored. Normalization consists of a series of guidelines that help guide you in creating a good database structure.

Normalization guidelines are divided into normal forms; think of a form as the format or the way a database structure is laid out. The aim of normal forms is to organize the database structure, so that it complies with the rules of first normal form, then second normal form and finally the third normal form.

It is your choice to take it further and go to the fourth normal form, fifth normal form and so on, but in general, the third normal form is more than enough.

* [First Normal Form (1NF)](https://www.tutorialspoint.com/sql/first-normal-form.htm)
* [Second Normal Form (2NF)](https://www.tutorialspoint.com/sql/second-normal-form.htm)
* [Third Normal Form (3NF)](https://www.tutorialspoint.com/sql/third-normal-form.htm)

**SQL Syntax**

SQL is followed by a unique set of rules and guidelines called Syntax. This tutorial gives you a quick start with SQL by listing all the basic SQL Syntax.

All the SQL statements start with any of the keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW and all the statements end with a semicolon (;).

The most important point to be noted here is that SQL is case insensitive, which means SELECT and select have same meaning in SQL statements. Whereas, MySQL makes difference in table names. So, if you are working with MySQL, then you need to give table names as they exist in the database.

## Various Syntax in SQL

All the examples given in this tutorial have been tested with a MySQL server.

### SQL SELECT Statement

SELECT column1, column2....columnN

FROM table\_name;

### SQL DISTINCT Clause

SELECT DISTINCT column1, column2....columnN

FROM table\_name;

### SQL WHERE Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION;

### SQL AND/OR Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION-1 {AND|OR} CONDITION-2;

### SQL IN Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name IN (val-1, val-2,...val-N);

### SQL BETWEEN Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name BETWEEN val-1 AND val-2;

### SQL LIKE Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name LIKE { PATTERN };

### SQL ORDER BY Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION

ORDER BY column\_name {ASC|DESC};

### SQL GROUP BY Clause

SELECT SUM(column\_name)

FROM table\_name

WHERE CONDITION

GROUP BY column\_name;

### SQL COUNT Clause

SELECT COUNT(column\_name)

FROM table\_name

WHERE CONDITION;

### SQL HAVING Clause

SELECT SUM(column\_name)

FROM table\_name

WHERE CONDITION

GROUP BY column\_name

HAVING (arithematic function condition);

### SQL CREATE TABLE Statement

CREATE TABLE table\_name(

column1 datatype,

column2 datatype,

column3 datatype,

.....

columnN datatype,

PRIMARY KEY( one or more columns )

);

### SQL DROP TABLE Statement

DROP TABLE table\_name;

### SQL CREATE INDEX Statement

CREATE UNIQUE INDEX index\_name

ON table\_name ( column1, column2,...columnN);

### SQL DROP INDEX Statement

ALTER TABLE table\_name

DROP INDEX index\_name;

### SQL DESC Statement

DESC table\_name;

### SQL TRUNCATE TABLE Statement

TRUNCATE TABLE table\_name;

### SQL ALTER TABLE Statement

ALTER TABLE table\_name {ADD|DROP|MODIFY} column\_name {data\_ype};

### SQL ALTER TABLE Statement (Rename)

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

### SQL INSERT INTO Statement

INSERT INTO table\_name( column1, column2....columnN)

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

### SQL UPDATE Statement

UPDATE table\_name

SET column1 = value1, column2 = value2....columnN=valueN

[ WHERE CONDITION ];

### SQL DELETE Statement

DELETE FROM table\_name

WHERE {CONDITION};

### SQL CREATE DATABASE Statement

CREATE DATABASE database\_name;

### SQL DROP DATABASE Statement

DROP DATABASE database\_name;

### SQL USE Statement

USE database\_name;

### SQL COMMIT Statement

COMMIT;

### SQL ROLLBACK Statement

ROLLBACK;

**Data Type**

SQL Data Type is an attribute that specifies the type of data of any object. Each column, variable and expression has a related data type in SQL. You can use these data types while creating your tables. You can choose a data type for a table column based on your requirement.

SQL Server offers six categories of data types for your use which are listed below −

Exact Numeric Data Types

|  |  |  |
| --- | --- | --- |
| **DATA TYPE** | **FROM** | **TO** |
| bigint | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |
| int | -2,147,483,648 | 2,147,483,647 |
| smallint | -32,768 | 32,767 |
| tinyint | 0 | 255 |
| bit | 0 | 1 |
| decimal | -10^38 +1 | 10^38 -1 |
| numeric | -10^38 +1 | 10^38 -1 |
| money | -922,337,203,685,477.5808 | +922,337,203,685,477.5807 |
| smallmoney | -214,748.3648 | +214,748.3647 |

Approximate Numeric Data Types

|  |  |  |
| --- | --- | --- |
| **DATA TYPE** | **FROM** | **TO** |
| float | -1.79E + 308 | 1.79E + 308 |
| real | -3.40E + 38 | 3.40E + 38 |

Date and Time Data Types

|  |  |  |
| --- | --- | --- |
| **DATA TYPE** | **FROM** | **TO** |
| datetime | Jan 1, 1753 | Dec 31, 9999 |
| smalldatetime | Jan 1, 1900 | Jun 6, 2079 |
| date | Stores a date like June 30, 1991 | |
| time | Stores a time of day like 12:30 P.M. | |

**Note** − Here, datetime has 3.33 milliseconds accuracy where as smalldatetime has 1 minute accuracy.

Character Strings Data Types

|  |  |
| --- | --- |
| **Sr.No.** | **DATA TYPE & Description** |
| 1 | **char**  Maximum length of 8,000 characters.( Fixed length non-Unicode characters) |
| 2 | **varchar**  Maximum of 8,000 characters.(Variable-length non-Unicode data). |
| 3 | **varchar(max)**  Maximum length of 2E + 31 characters, Variable-length non-Unicode data (SQL Server 2005 only). |
| 4 | **text**  Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters. |

## What is an Operator in SQL?

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. These Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

* Arithmetic operators
* Comparison operators
* Logical operators
* Operators used to negate conditions

## SQL Arithmetic Operators

Assume **'variable a'** holds 10 and **'variable b'** holds 20, then −

[Show Examples](https://www.tutorialspoint.com/sql/sql-arithmetic-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator. | a + b will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand. | a - b will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator. | a \* b will give 200 |
| / (Division) | Divides left hand operand by right hand operand. | b / a will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder. | b % a will give 0 |

## SQL Comparison Operators

Assume **'variable a'** holds 10 and **'variable b'** holds 20, then −

[Show Examples](https://www.tutorialspoint.com/sql/sql-comparison-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (a = b) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a != b) is true. |
| <> | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a <> b) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (a > b) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (a < b) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (a >= b) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (a <= b) is true. |
| !< | Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true. | (a !< b) is false. |
| !> | Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true. | (a !> b) is true. |

## SQL Logical Operators

Here is a list of all the logical operators available in SQL.

[Show Examples](https://www.tutorialspoint.com/sql/sql-logical-operators.htm)

|  |  |
| --- | --- |
| **Sr.No.** | **Operator & Description** |
| 1 | **ALL**  The ALL operator is used to compare a value to all values in another value set. |
| 2 | **AND**  The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause. |
| 3 | **ANY**  The ANY operator is used to compare a value to any applicable value in the list as per the condition. |
| 4 | **BETWEEN**  The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value. |
| 5 | **EXISTS**  The EXISTS operator is used to search for the presence of a row in a specified table that meets a certain criterion. |
| 6 | **IN**  The IN operator is used to compare a value to a list of literal values that have been specified. |
| 7 | **LIKE**  The LIKE operator is used to compare a value to similar values using wildcard operators. |
| 8 | **NOT**  The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.** |
| 9 | **OR**  The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause. |
| 10 | **IS NULL**  The NULL operator is used to compare a value with a NULL value. |
| 11 | **UNIQUE**  The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates). |

## The SQL SELECT Statement

The SELECT statement is used to select data from a database.

The data returned is stored in a result table, called the result-set.

### SELECT Syntax

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

Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table, use the following syntax:

SELECT \* FROM table\_name;

## Demo Database

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 |

## SELECT Column Example

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

### Example

SELECT CustomerName, City FROM Customers;

## SELECT \* Example

The following SQL statement selects all the columns from the "Customers" table:

### Example

SELECT \* FROM Customers;

## The SQL SELECT DISTINCT Statement

The SELECT DISTINCT statement is used to return only distinct (different) values.

Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

### SELECT DISTINCT Syntax

SELECT DISTINCT column1, column2, ...  
FROM table\_name;

## Demo Database

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 |

## SELECT Example Without DISTINCT

The following SQL statement selects ALL (including the duplicates) values from the "Country" column in the "Customers" table:

### Example

SELECT Country FROM Customers;

## SELECT DISTINCT Examples

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

### Example

SELECT 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.!

## Demo Database

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 |

## WHERE Clause Example

The following SQL statement selects all the customers from the country "Mexico", in the "Customers" table:

### Example

SELECT \* FROM Customers  
WHERE Country='Mexico';

## Text Fields vs. Numeric Fields

SQL requires single quotes around text values (most database systems will also allow double quotes).

However, numeric fields should not be enclosed in quotes:

### Example

SELECT \* FROM Customers  
WHERE CustomerID=1;

## Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Equal | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_equal_to) |
| > | Greater than | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_greater_than) |
| < | Less than | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_less_than) |
| >= | Greater than or equal | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_greater_than2) |
| <= | Less than or equal | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_less_than2) |
| <> | Not equal. **Note:** In some versions of SQL this operator may be written as != | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_not_equal_to) |
| BETWEEN | Between a certain range | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_between) |
| LIKE | Search for a pattern | [Try it](https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_like) |
| 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;

## Demo Database

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 |

## 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';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_where_and)

## 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';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_where_or)

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';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_where_or2)

## NOT Example

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

### Example

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

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_where_not)

## Combining AND, OR and NOT

You 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';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_where_not_and)

## 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;

## Demo Database

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 |

## ORDER BY Example

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

### Example

SELECT \* FROM Customers  
ORDER BY Country;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_orderby)

## ORDER BY DESC Example

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;

## 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 you are adding values for all the columns of the table, you 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, ...);

## Demo Database

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 89 | White Clover Markets | Karl Jablonski | 305 - 14th Ave. S. Suite 3B | Seattle | 98128 | USA |
| 90 | Wilman Kala | Matti Karttunen | Keskuskatu 45 | Helsinki | 21240 | Finland |
| 91 | Wolski | Zbyszek | ul. Filtrowa 68 | Walla | 01-012 | Poland |

## 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');

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_insert_colname)

The selection from the "Customers" table will now look like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 89 | White Clover Markets | Karl Jablonski | 305 - 14th Ave. S. Suite 3B | Seattle | 98128 | USA |
| 90 | Wilman Kala | Matti Karttunen | Keskuskatu 45 | Helsinki | 21240 | Finland |
| 91 | Wolski | Zbyszek | ul. Filtrowa 68 | Walla | 01-012 | Poland |
| 92 | Cardinal | Tom B. Erichsen | Skagen 21 | Stavanger | 4006 | Norway |

**Did you notice that we did not insert any number into the CustomerID field?**  
The CustomerID column is an [auto-increment](https://www.w3schools.com/sql/sql_autoincrement.asp) field and will be generated automatically when a new record is inserted into the table.

## 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');

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_insert_cols)

## What is a 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.

**Note:** 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;

## Demo Database

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 IS NULL Operator

The IS NULL operator is used to test for empty values (NULL values).

The following SQL lists all customers with a NULL value in the "Address" field:

### Example

SELECT CustomerName, ContactName, Address  
FROM Customers  
WHERE Address IS NULL;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_is_null)

**Tip:** Always use IS NULL to look for NULL values.

## The IS NOT NULL Operator

The IS NOT NULL operator is used to test for non-empty values (NOT NULL values).

The following SQL lists all customers with a value in the "Address" field:

### Example

SELECT CustomerName, ContactName, Address  
FROM Customers  
WHERE Address IS NOT NULL;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_is_not_null)

## 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 you omit the WHERE clause, all records in the table will be updated!

## Demo Database

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 |

## UPDATE Table

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'  
WHERE CustomerID = 1;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_update_2)

The selection from the "Customers" table will now look like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Alfred Schmidt | Obere Str. 57 | Frankfurt | 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 |

## UPDATE Multiple Records

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';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_update_3)

The selection from the "Customers" table will now look like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Alfred Schmidt | Obere Str. 57 | Frankfurt | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Juan | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Juan | 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 |

## Update Warning!

Be careful when updating records. If you omit the WHERE clause, ALL records will be updated!

### Example

UPDATE Customers  
SET ContactName='Juan';

## 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 you omit the WHERE clause, all records in the table will be deleted!

## Demo Database

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 |

## SQL DELETE Example

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

### Example

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_delete)

The "Customers" table will now look like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 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 |

## Delete All Records

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;

## The SQL MIN() and MAX() Functions

The MIN() function returns the smallest value of the selected column.

The MAX() function returns the largest value of the selected column.

### MIN() Syntax

SELECT MIN(column\_name)  
FROM table\_name  
WHERE condition;

### MAX() Syntax

SELECT MAX(column\_name)  
FROM table\_name  
WHERE condition;

## Demo Database

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ProductID** | **ProductName** | **SupplierID** | **CategoryID** | **Unit** | **Price** |
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 22 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 21.35 |

## MIN() Example

The following SQL statement finds the price of the cheapest product:

### Example

SELECT MIN(Price) AS SmallestPrice  
FROM Products;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_min)

## MAX() Example

The following SQL statement finds the price of the most expensive product:

### Example

SELECT MAX(Price) AS LargestPrice  
FROM Products;

## The SQL COUNT(), AVG() and SUM() Functions

The COUNT() function returns the number of rows that matches a specified criteria.

The AVG() function returns the average value of a numeric column.

The SUM() function returns the total sum of a numeric column.

### COUNT() Syntax

SELECT COUNT(column\_name)  
FROM table\_name  
WHERE condition;

### AVG() Syntax

SELECT AVG(column\_name)  
FROM table\_name  
WHERE condition;

### SUM() Syntax

SELECT SUM(column\_name)  
FROM table\_name  
WHERE condition;

## Demo Database

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ProductID** | **ProductName** | **SupplierID** | **CategoryID** | **Unit** | **Price** |
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 22 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 21.35 |

## COUNT() Example

The following SQL statement finds the number of products:

### Example

SELECT COUNT(ProductID)  
FROM Products;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_count)

**Note:** NULL values are not counted.

## AVG() Example

The following SQL statement finds the average price of all products:

### Example

SELECT AVG(Price)  
FROM Products;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_avg)

**Note:** NULL values are ignored.

## Demo Database

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

|  |  |  |  |
| --- | --- | --- | --- |
| **OrderDetailID** | **OrderID** | **ProductID** | **Quantity** |
| 1 | 10248 | 11 | 12 |
| 2 | 10248 | 42 | 10 |
| 3 | 10248 | 72 | 5 |
| 4 | 10249 | 14 | 9 |
| 5 | 10249 | 51 | 40 |

## SUM() Example

The following SQL statement finds the sum of the "Quantity" fields in the "OrderDetails" table:

### Example

SELECT SUM(Quantity)  
FROM OrderDetails;

## The SQL LIKE Operator

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the LIKE operator:

* % - The percent sign represents zero, one, or multiple characters
* \_ - The underscore represents a single character

**Note:** MS Access uses an asterisk (\*) instead of the percent sign (%), and a question mark (?) instead of the underscore (\_).

The percent sign and the underscore can also be used in combinations!

### LIKE Syntax

SELECT column1, column2, ...  
FROM table\_name  
WHERE columnN LIKE pattern;

**Tip:** You can also combine any number of conditions using AND or OR operators.

Here are some examples showing different LIKE operators with '%' and '\_' wildcards:

|  |  |
| --- | --- |
| **LIKE Operator** | **Description** |
| WHERE CustomerName LIKE 'a%' | Finds any values that start with "a" |
| WHERE CustomerName LIKE '%a' | Finds any values that end with "a" |
| WHERE CustomerName LIKE '%or%' | Finds any values that have "or" in any position |
| WHERE CustomerName LIKE '\_r%' | Finds any values that have "r" in the second position |
| WHERE CustomerName LIKE 'a\_\_%' | Finds any values that start with "a" and are at least 3 characters in length |
| WHERE ContactName LIKE 'a%o' | Finds any values that start with "a" and ends with "o" |

## Demo Database

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 |

## SQL LIKE Examples

The following SQL statement selects all customers with a CustomerName starting with "a":

### Example

SELECT \* FROM Customers  
WHERE CustomerName LIKE 'a%';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_like)

The following SQL statement selects all customers with a CustomerName ending with "a":

### Example

SELECT \* FROM Customers  
WHERE CustomerName LIKE '%a';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_like_ending)

The following SQL statement selects all customers with a CustomerName that have "or" in any position:

### Example

SELECT \* FROM Customers  
WHERE CustomerName LIKE '%or%';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_like_pattern)

The following SQL statement selects all customers with a CustomerName that have "r" in the second position:

### Example

SELECT \* FROM Customers  
WHERE CustomerName LIKE '\_r%';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_like_underscore)

The following SQL statement selects all customers with a CustomerName that starts with "a" and are at least 3 characters in length:

### Example

SELECT \* FROM Customers  
WHERE CustomerName LIKE 'a\_\_%';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_like_start_least)

The following SQL statement selects all customers with a ContactName that starts with "a" and ends with "o":

### Example

SELECT \* FROM Customers  
WHERE ContactName LIKE 'a%o';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_like_start_end)

The following SQL statement selects all customers with a CustomerName that does NOT start with "a":

### Example

SELECT \* FROM Customers  
WHERE CustomerName NOT LIKE 'a%';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_like_not)

## SQL ALTER TABLE Statement

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

## ALTER TABLE - ADD Column

To add a column in a table, use the following syntax:

ALTER TABLE table\_name  
ADD column\_name datatype;

The following SQL adds an "Email" column to the "Customers" table:

### Example

ALTER TABLE Customers  
ADD Email varchar(255);

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_alter_table)

## ALTER TABLE - DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

ALTER TABLE table\_name  
DROP COLUMN column\_name;

The following SQL deletes the "Email" column from the "Customers" table:

### Example

ALTER TABLE Customers  
DROP COLUMN Email;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_alter_table2)

## ALTER TABLE - ALTER/MODIFY COLUMN

To change the data type of a column in a table, use the following syntax:

**SQL Server / MS Access:**

ALTER TABLE table\_name  
ALTER COLUMN column\_name datatype;

**My SQL / Oracle (prior version 10G):**

ALTER TABLE table\_name  
MODIFY COLUMN column\_name datatype;

**Oracle 10G and later:**

ALTER TABLE table\_name  
MODIFY column\_name datatype;

## SQL ALTER TABLE Example

Look at the "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

Now we want to add a column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

ALTER TABLE Persons  
ADD DateOfBirth date;

Notice that the new column, "DateOfBirth", is of type date and is going to hold a date. The data type specifies what type of data the column can hold. For a complete reference of all the data types available in MS Access, MySQL, and SQL Server, go to our complete [Data Types reference](https://www.w3schools.com/sql/sql_datatypes.asp).

The "Persons" table will now look like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **LastName** | **FirstName** | **Address** | **City** | **DateOfBirth** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |  |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |  |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |  |

## Change Data Type Example

Now we want to change the data type of the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

ALTER TABLE Persons  
ALTER COLUMN DateOfBirth year;

Notice that the "DateOfBirth" column is now of type year and is going to hold a year in a two- or four-digit format.

## DROP COLUMN Example

Next, we want to delete the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

ALTER TABLE Persons  
DROP COLUMN DateOfBirth;

The "Persons" table will now look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

## The SQL GROUP BY Statement

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

### GROUP BY Syntax

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)ORDER BY column\_name(s);

## Demo Database

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 |

## SQL GROUP BY Examples

The following SQL statement lists the number of customers in each country:

### Example

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_groupby)

The following SQL statement lists the number of customers in each country, sorted high to low:

### Example

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
ORDER BY COUNT(CustomerID) DESC;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_groupby_orderby)

## Demo Database

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10248 | 90 | 5 | 1996-07-04 | 3 |
| 10249 | 81 | 6 | 1996-07-05 | 1 |
| 10250 | 34 | 4 | 1996-07-08 | 2 |

And a selection from the "Shippers" table:

|  |  |
| --- | --- |
| **ShipperID** | **ShipperName** |
| 1 | Speedy Express |
| 2 | United Package |
| 3 | Federal Shipping |

## GROUP BY With JOIN Example

The following SQL statement lists the number of orders sent by each shipper:

### Example

SELECT Shippers.ShipperName, COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders  
LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID  
GROUP BY ShipperName;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_groupby1)

## The SQL HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

### HAVING Syntax

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)HAVING conditionORDER BY column\_name(s);

## Demo Database

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 |

## SQL HAVING Examples

The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers:

### Example

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
HAVING COUNT(CustomerID) > 5;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_having)

The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):

### Example

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
HAVING COUNT(CustomerID) > 5  
ORDER BY COUNT(CustomerID) DESC;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_having_orderby)

## Demo Database

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10248 | 90 | 5 | 1996-07-04 | 3 |
| 10249 | 81 | 6 | 1996-07-05 | 1 |
| 10250 | 34 | 4 | 1996-07-08 | 2 |

And a selection from the "Employees" table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EmployeeID** | **LastName** | **FirstName** | **BirthDate** | **Photo** | **Notes** |
| 1 | Davolio | Nancy | 1968-12-08 | EmpID1.pic | Education includes a BA.... |
| 2 | Fuller | Andrew | 1952-02-19 | EmpID2.pic | Andrew received his BTS.... |
| 3 | Leverling | Janet | 1963-08-30 | EmpID3.pic | Janet has a BS degree.... |

## More HAVING Examples

The following SQL statement lists the employees that have registered more than 10 orders:

### Example

SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders  
FROM (Orders  
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID)  
GROUP BY LastName  
HAVING COUNT(Orders.OrderID) > 10;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_having2)

The following SQL statement lists if the employees "Davolio" or "Fuller" have registered more than 25 orders:

### Example

SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders  
FROM Orders  
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID  
WHERE LastName = 'Davolio' OR LastName = 'Fuller'  
GROUP BY LastName  
HAVING COUNT(Orders.OrderID) > 25;

## The SQL SELECT INTO Statement

The SELECT INTO statement copies data from one table into a new table.

### SELECT INTO Syntax

Copy all columns into a new table:

SELECT \*  
INTO newtable [IN externaldb]  
FROM oldtableWHERE condition;

Copy only some columns into a new table:

SELECT column1, column2, column3, ...  
INTO newtable [IN externaldb]  
FROM oldtableWHERE condition;

The new table will be created with the column-names and types as defined in the old table. You can create new column names using the AS clause.

## SQL SELECT INTO Examples

The following SQL statement creates a backup copy of Customers:

SELECT \* INTO CustomersBackup2017  
FROM Customers;

The following SQL statement uses the IN clause to copy the table into a new table in another database:

SELECT \* INTO CustomersBackup2017 IN 'Backup.mdb'  
FROM Customers;

The following SQL statement copies only a few columns into a new table:

SELECT CustomerName, ContactName INTO CustomersBackup2017  
FROM Customers;

The following SQL statement copies only the German customers into a new table:

SELECT \* INTO CustomersGermany  
FROM Customers  
WHERE Country = 'Germany';

The following SQL statement copies data from more than one table into a new table:

SELECT Customers.CustomerName, Orders.OrderID  
INTO CustomersOrderBackup2017  
FROM Customers  
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;

**Tip:** SELECT INTO can also be used to create a new, empty table using the schema of another. Just add a WHERE clause that causes the query to return no data:

SELECT \* INTO newtable  
FROM oldtable  
WHERE 1 = 0;

## SQL Aliases

SQL aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

An alias only exists for the duration of the query.

### Alias Column Syntax

SELECT column\_name AS alias\_name  
FROM table\_name;

### Alias Table Syntax

SELECT column\_name(s)  
FROM table\_name AS alias\_name;

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 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 |

And a selection from the "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10354 | 58 | 8 | 1996-11-14 | 3 |
| 10355 | 4 | 6 | 1996-11-15 | 1 |
| 10356 | 86 | 6 | 1996-11-18 | 2 |

## Alias for Columns Examples

The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column:

### Example

SELECT CustomerID AS ID, CustomerName AS Customer  
FROM Customers;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_alias_column0)

The following SQL statement creates two aliases, one for the CustomerName column and one for the ContactName column. **Note:** It requires double quotation marks or square brackets if the alias name contains spaces:

### Example

SELECT CustomerName AS Customer, ContactName AS [Contact Person]  
FROM Customers;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_alias_column)

The following SQL statement creates an alias named "Address" that combine four columns (Address, PostalCode, City and Country):

### Example

SELECT CustomerName, Address + ', ' + PostalCode + ' ' + City + ', ' + Country AS Address  
FROM Customers;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_alias_column2&ss=-1)

**Note:** To get the SQL statement above to work in MySQL use the following:

SELECT CustomerName, CONCAT(Address,', ',PostalCode,', ',City,', ',Country) AS Address  
FROM Customers;

## Alias for Tables Example

The following SQL statement selects all the orders from the customer with CustomerID=4 (Around the Horn). We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively (Here we use aliases to make the SQL shorter):

### Example

SELECT o.OrderID, o.OrderDate, c.CustomerName  
FROM Customers AS c, Orders AS o  
WHERE c.CustomerName="Around the Horn" AND c.CustomerID=o.CustomerID;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_alias_table)

The following SQL statement is the same as above, but without aliases:

### Example

SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName  
FROM Customers, Orders  
WHERE Customers.CustomerName="Around the Horn" AND Customers.CustomerID=Orders.CustomerID;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_alias_no)

Aliases can be useful when:

* There are more than one table involved in a query
* Functions are used in the query
* Column names are big or not very readable
* Two or more columns are combined together

## SQL ALTER TABLE Statement

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

## ALTER TABLE - ADD Column

To add a column in a table, use the following syntax:

ALTER TABLE table\_name  
ADD column\_name datatype;

The following SQL adds an "Email" column to the "Customers" table:

### Example

ALTER TABLE Customers  
ADD Email varchar(255);

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_alter_table)

## ALTER TABLE - DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

ALTER TABLE table\_name  
DROP COLUMN column\_name;

The following SQL deletes the "Email" column from the "Customers" table:

### Example

ALTER TABLE Customers  
DROP COLUMN Email;

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_alter_table2)

## ALTER TABLE - ALTER/MODIFY COLUMN

To change the data type of a column in a table, use the following syntax:

**SQL Server / MS Access:**

ALTER TABLE table\_name  
ALTER COLUMN column\_name datatype;

**My SQL / Oracle (prior version 10G):**

ALTER TABLE table\_name  
MODIFY COLUMN column\_name datatype;

**Oracle 10G and later:**

ALTER TABLE table\_name  
MODIFY column\_name datatype;

## SQL ALTER TABLE Example

Look at the "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

Now we want to add a column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

ALTER TABLE Persons  
ADD DateOfBirth date;

Notice that the new column, "DateOfBirth", is of type date and is going to hold a date. The data type specifies what type of data the column can hold. For a complete reference of all the data types available in MS Access, MySQL, and SQL Server, go to our complete [Data Types reference](https://www.w3schools.com/sql/sql_datatypes.asp).

The "Persons" table will now look like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **LastName** | **FirstName** | **Address** | **City** | **DateOfBirth** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |  |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |  |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |  |

## Change Data Type Example

Now we want to change the data type of the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

ALTER TABLE Persons  
ALTER COLUMN DateOfBirth year;

Notice that the "DateOfBirth" column is now of type year and is going to hold a year in a two- or four-digit format.

## DROP COLUMN Example

Next, we want to delete the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

ALTER TABLE Persons  
DROP COLUMN DateOfBirth;

The "Persons" table will now look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

## SQL Wildcard Characters

A wildcard character is used to substitute one or more characters in a string.

Wildcard characters are used with the [SQL LIKE](https://www.w3schools.com/sql/sql_like.asp) operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

### Wildcard Characters in MS Access

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Description** | **Example** |
| \* | Represents zero or more characters | bl\* finds bl, black, blue, and blob |
| ? | Represents a single character | h?t finds hot, hat, and hit |
| [] | Represents any single character within the brackets | h[oa]t finds hot and hat, but not hit |
| ! | Represents any character not in the brackets | h[!oa]t finds hit, but not hot and hat |
| - | Represents a range of characters | c[a-b]t finds cat and cbt |
| # | Represents any single numeric character | 2#5 finds 205, 215, 225, 235, 245, 255, 265, 275, 285, and 295 |

### Wildcard Characters in SQL Server

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Description** | **Example** |
| % | Represents zero or more characters | bl% finds bl, black, blue, and blob |
| \_ | Represents a single character | h\_t finds hot, hat, and hit |
| [] | Represents any single character within the brackets | h[oa]t finds hot and hat, but not hit |
| ^ | Represents any character not in the brackets | h[^oa]t finds hit, but not hot and hat |
| - | Represents a range of characters | c[a-b]t finds cat and cbt |

All the wildcards can also be used in combinations!

Here are some examples showing different LIKE operators with '%' and '\_' wildcards:

|  |  |
| --- | --- |
| **LIKE Operator** | **Description** |
| WHERE CustomerName LIKE 'a%' | Finds any values that starts with "a" |
| WHERE CustomerName LIKE '%a' | Finds any values that ends with "a" |
| WHERE CustomerName LIKE '%or%' | Finds any values that have "or" in any position |
| WHERE CustomerName LIKE '\_r%' | Finds any values that have "r" in the second position |
| WHERE CustomerName LIKE 'a\_%\_%' | Finds any values that starts with "a" and are at least 3 characters in length |
| WHERE ContactName LIKE 'a%o' | Finds any values that starts with "a" and ends with "o" |

## Demo Database

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 |

## Using the % Wildcard

The following SQL statement selects all customers with a City starting with "ber":

### Example

SELECT \* FROM Customers  
WHERE City LIKE 'ber%';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_wildcard_percent)

The following SQL statement selects all customers with a City containing the pattern "es":

### Example

SELECT \* FROM Customers  
WHERE City LIKE '%es%';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_wildcard_percent_pattern)

## Using the \_ Wildcard

The following SQL statement selects all customers with a City starting with any character, followed by "ondon":

### Example

SELECT \* FROM Customers  
WHERE City LIKE '\_ondon';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_wildcard_underscore)

The following SQL statement selects all customers with a City starting with "L", followed by any character, followed by "n", followed by any character, followed by "on":

### Example

SELECT \* FROM Customers  
WHERE City LIKE 'L\_n\_on';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_wildcard_underscore2)

## Using the [charlist] Wildcard

The following SQL statement selects all customers with a City starting with "b", "s", or "p":

### Example

SELECT \* FROM Customers  
WHERE City LIKE '[bsp]%';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_wildcard_charlist&ss=-1)

The following SQL statement selects all customers with a City starting with "a", "b", or "c":

### Example

SELECT \* FROM Customers  
WHERE City LIKE '[a-c]%';

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_wildcard_charlist2&ss=-1)

# SQL - NOT NULL Constraint

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such a constraint on this column specifying that NULL is now not allowed for that column.

A NULL is not the same as no data, rather, it represents unknown data.

## Example

For example, the following SQL query creates a new table called CUSTOMERS and adds five columns, three of which, are ID NAME and AGE, In this we specify not to accept NULLs −

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to the SALARY column in Oracle and MySQL, you would write a query like the one that is shown in the following code block.

ALTER TABLE CUSTOMERS

MODIFY SALARY DECIMAL (18, 2) NOT NULL;

# SQL - DEFAULT Constraint

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.

## Example

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, the SALARY column is set to 5000.00 by default, so in case the INSERT INTO statement does not provide a value for this column, then by default this column would be set to 5000.00.

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2) DEFAULT 5000.00,

PRIMARY KEY (ID)

);

If the CUSTOMERS table has already been created, then to add a DEFAULT constraint to the SALARY column, you would write a query like the one which is shown in the code block below.

# SQL - UNIQUE Constraint

The UNIQUE Constraint prevents two records from having identical values in a column. In the CUSTOMERS table, for example, you might want to prevent two or more people from having an identical age.

### Example

For example, the following SQL query creates a new table called CUSTOMERS and adds five columns. Here, the AGE column is set to UNIQUE, so that you cannot have two records with the same age.

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL UNIQUE,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

# SQL - Primary Key

A primary key is a field in a table which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.

A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a composite key.

If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

**Note** − You would use these concepts while creating database tables.

## Create Primary Key

Here is the syntax to define the ID attribute as a primary key in a CUSTOMERS table.

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

To create a PRIMARY KEY constraint on the "ID" column when the CUSTOMERS table already exists, use the following SQL syntax −

ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID);

# SQL - Foreign Key

A foreign key is a key used to link two tables together. This is sometimes also called as a referencing key.

A Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.

**The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.**

If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

### Example

Consider the structure of the following two tables.

**CUSTOMERS table**

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

**ORDERS table**

CREATE TABLE ORDERS (

ID INT NOT NULL,

DATE DATETIME,

CUSTOMER\_ID INT references CUSTOMERS(ID),

AMOUNT double,

PRIMARY KEY (ID)

);

# SQL - CHECK Constraint

The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and isn't entered the table.

### Example

For example, the following program creates a new table called CUSTOMERS and adds five columns. Here, we add a CHECK with AGE column, so that you cannot have any CUSTOMER who is below 18 years.

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL CHECK (AGE >= 18),

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

Dropping Constraints

Any constraint that you have defined can be dropped using the ALTER TABLE command with the DROP CONSTRAINT option.

For example, to drop the primary key constraint in the EMPLOYEES table, you can use the following command.

ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES\_PK;

Some implementations may provide shortcuts for dropping certain constraints. For example, to drop the primary key constraint for a table in Oracle, you can use the following command.

ALTER TABLE EMPLOYEES DROP PRIMARY KEY;

# SQL - INNER JOINS

The most important and frequently used of the joins is the **INNER JOIN**. They are also referred to as an **EQUIJOIN**.

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

## Syntax

The basic syntax of the **INNER JOIN** is as follows.

SELECT table1.column1, table2.column2...

FROM table1

INNER JOIN table2

ON table1.common\_field = table2.common\_field;

## Example

Consider the following two tables.

**Table 1** − CUSTOMERS Table is as follows.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

**Table 2** − ORDERS Table is as follows.

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

| OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Now, let us join these two tables using the INNER JOIN as follows −

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

INNER JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result.

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

| ID | NAME | AMOUNT | DATE |

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

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

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

# SQL - LEFT JOINS

The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

## Syntax

The basic syntax of a **LEFT JOIN** is as follows.

SELECT table1.column1, table2.column2...

FROM table1

LEFT JOIN table2

ON table1.common\_field = table2.common\_field;

Here, the given condition could be any given expression based on your requirement.

## Example

Consider the following two tables,

**Table 1** − CUSTOMERS Table is as follows.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

**Table 2** − Orders Table is as follows.

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

| OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Now, let us join these two tables using the LEFT JOIN as follows.

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result −

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

| ID | NAME | AMOUNT | DATE |

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

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

# SQL - RIGHT JOINS

The SQL **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

## Syntax

The basic syntax of a **RIGHT JOIN** is as follow.

SELECT table1.column1, table2.column2...

FROM table1

RIGHT JOIN table2

ON table1.common\_field = table2.common\_field;

## Example

Consider the following two tables,

**Table 1** − CUSTOMERS Table is as follows.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

**Table 2** − ORDERS Table is as follows.

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

|OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Now, let us join these two tables using the RIGHT JOIN as follows.

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result −

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

| ID | NAME | AMOUNT | DATE |

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

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

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

# SQL - FULL JOINS

The SQL **FULL JOIN** combines the results of both left and right outer joins.

The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side.

## Syntax

The basic syntax of a **FULL JOIN** is as follows −

SELECT table1.column1, table2.column2...

FROM table1

FULL JOIN table2

ON table1.common\_field = table2.common\_field;

Here, the given condition could be any given expression based on your requirement.

## Example

Consider the following two tables.

**Table 1** − CUSTOMERS Table is as follows.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

**Table 2** − ORDERS Table is as follows.

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

|OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Now, let us join these two tables using FULL JOIN as follows.

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

FULL JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result −

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

| ID | NAME | AMOUNT | DATE |

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

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

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

# SQL - CARTESIAN or CROSS JOINS

The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to either True or where the join-condition is absent from the statement.

## Syntax

The basic syntax of the **CARTESIAN JOIN** or the **CROSS JOIN** is as follows −

SELECT table1.column1, table2.column2...

FROM table1, table2 [, table3 ]

## Example

Consider the following two tables.

**Table 1** − CUSTOMERS table is as follows.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

Table 2: ORDERS Table is as follows −

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

|OID | DATE | CUSTOMER\_ID | AMOUNT |

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

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

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

Now, let us join these two tables using CARTESIAN JOIN as follows −

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS, ORDERS;

This would produce the following result −

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

| ID | NAME | AMOUNT | DATE |

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

| 1 | Ramesh | 3000 | 2009-10-08 00:00:00 |

| 1 | Ramesh | 1500 | 2009-10-08 00:00:00 |

| 1 | Ramesh | 1560 | 2009-11-20 00:00:00 |

| 1 | Ramesh | 2060 | 2008-05-20 00:00:00 |

| 2 | Khilan | 3000 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 2 | Khilan | 2060 | 2008-05-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 2060 | 2008-05-20 00:00:00 |

| 4 | Chaitali | 3000 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | 3000 | 2009-10-08 00:00:00 |

| 5 | Hardik | 1500 | 2009-10-08 00:00:00 |

| 5 | Hardik | 1560 | 2009-11-20 00:00:00 |

| 5 | Hardik | 2060 | 2008-05-20 00:00:00 |

| 6 | Komal | 3000 | 2009-10-08 00:00:00 |

| 6 | Komal | 1500 | 2009-10-08 00:00:00 |

| 6 | Komal | 1560 | 2009-11-20 00:00:00 |

| 6 | Komal | 2060 | 2008-05-20 00:00:00 |

| 7 | Muffy | 3000 | 2009-10-08 00:00:00 |

| 7 | Muffy | 1500 | 2009-10-08 00:00:00 |

| 7 | Muffy | 1560 | 2009-11-20 00:00:00 |

| 7 | Muffy | 2060 | 2008-05-20 00:00:00 |

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

# What are SQL Functions?

SQL provides many built-in functions to perform operations on data. These functions are useful while performing mathematical calculations, string concatenations, sub-strings etc. SQL functions are divided into two categories,

1. Aggregate Functions
2. Scalar Functions

## Aggregate Functions

These functions **return a single value** after performing calculations on a group of values. Following are some of the frequently used Aggregrate functions.

### AVG() Function

Average returns average value after calculating it from values in a numeric column.

Its general **syntax** is,

SELECT AVG(column\_name) FROM table\_name

#### Using AVG() function

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to find average salary will be,

SELECT avg(salary) from Emp;

Result of the above query will be,

|  |
| --- |
| **avg(salary)** |
| 8200 |

### COUNT() Function

Count returns the number of rows present in the table either based on some condition or without condition.

Its general **syntax** is,

SELECT COUNT(column\_name) FROM table-name

#### Using COUNT() function

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to count employees, satisfying specified condition is,

SELECT COUNT(name) FROM Emp WHERE salary = 8000;

Result of the above query will be,

|  |
| --- |
| **count(name)** |
| 2 |

#### Example of COUNT(distinct)

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query is,

SELECT COUNT(DISTINCT salary) FROM emp;

Result of the above query will be,

|  |
| --- |
| **count(distinct salary)** |
| 4 |

### FIRST() Function

First function returns first value of a selected column

**Syntax** for FIRST function is,

SELECT FIRST(column\_name) FROM table-name;

#### Using FIRST() function

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query will be,

SELECT FIRST(salary) FROM Emp;

and the result will be,

|  |
| --- |
| **first(salary)** |
| 9000 |

### LAST() Function

LAST function returns the return last value of the selected column.

**Syntax** of LAST function is,

SELECT LAST(column\_name) FROM table-name;

#### Using LAST() function

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query will be,

SELECT LAST(salary) FROM emp;

Result of the above query will be,

|  |
| --- |
| **last(salary)** |
| 8000 |

### MAX() Function

MAX function returns maximum value from selected column of the table.

**Syntax** of MAX function is,

SELECT MAX(column\_name) from table-name;

#### Using MAX() function

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to find the Maximum salary will be,

SELECT MAX(salary) FROM emp;

Result of the above query will be,

|  |
| --- |
| **MAX(salary)** |
| 10000 |

### MIN() Function

MIN function returns minimum value from a selected column of the table.

**Syntax** for MIN function is,

SELECT MIN(column\_name) from table-name;

#### Using MIN() function

Consider the following **Emp** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to find minimum salary is,

SELECT MIN(salary) FROM emp;

Result will be,

|  |
| --- |
| **MIN(salary)** |
| 6000 |

### SUM() Function

SUM function returns total sum of a selected columns numeric values.

**Syntax** for SUM is,

SELECT SUM(column\_name) from table-name;

#### Using SUM() function

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to find sum of salaries will be,

SELECT SUM(salary) FROM emp;

Result of above query is,

|  |
| --- |
| **SUM(salary)** |
| 41000 |

## Scalar Functions

Scalar functions return a single value from an input value. Following are some frequently used Scalar Functions in SQL.

### UCASE() Function

UCASE function is used to convert value of string column to Uppercase characters.

**Syntax** of UCASE,

SELECT UCASE(column\_name) from table-name;

#### Using UCASE() function

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | anu | 22 | 9000 |
| 402 | shane | 29 | 8000 |
| 403 | rohan | 34 | 6000 |
| 404 | scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query for using UCASE is,

SELECT UCASE(name) FROM emp;

Result is,

|  |
| --- |
| **UCASE(name)** |
| ANU |
| SHANE |
| ROHAN |
| SCOTT |
| TIGER |

### LCASE() Function

LCASE function is used to convert value of string columns to Lowecase characters.

**Syntax** for LCASE is,

SELECT LCASE(column\_name) FROM table-name;

#### Using LCASE() function

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | SCOTT | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query for converting string value to Lower case is,

SELECT LCASE(name) FROM emp;

Result will be,

|  |
| --- |
| **LCASE(name)** |
| anu |
| shane |
| rohan |
| scott |
| tiger |

### MID() Function

MID function is used to extract substrings from column values of string type in a table.

**Syntax** for MID function is,

SELECT MID(column\_name, start, length) from table-name;

#### Using MID() function

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | anu | 22 | 9000 |
| 402 | shane | 29 | 8000 |
| 403 | rohan | 34 | 6000 |
| 404 | scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query will be,

SELECT MID(name,2,2) FROM emp;

Result will come out to be,

|  |
| --- |
| **MID(name,2,2)** |
| nu |
| ha |
| oh |
| co |
| ig |

### ROUND() Function

ROUND function is used to round a numeric field to number of nearest integer. It is used on Decimal point values.

**Syntax** of Round function is,

SELECT ROUND(column\_name, decimals) from table-name;

#### Using ROUND() function

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | anu | 22 | 9000.67 |
| 402 | shane | 29 | 8000.98 |
| 403 | rohan | 34 | 6000.45 |
| 404 | scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000.01 |

SQL query is,

SELECT ROUND(salary) from emp;

Result will be,

|  |
| --- |
| **ROUND(salary)** |
| 9001 |
| 8001 |
| 6000 |
| 10000 |
| 8000 |

# SQL Sub Query

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

**Important Rule:**

* A subquery can be placed in a number of SQL clauses like WHERE clause, FROM clause, HAVING clause.
* You can use Subquery with SELECT, UPDATE, INSERT, DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
* A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery.
* Subqueries are on the right side of the comparison operator.
* A subquery is enclosed in parentheses.
* In the Subquery, ORDER BY command cannot be used. But GROUP BY command can be used to perform the same function as ORDER BY command.

## 1. Subqueries with the Select Statement

SQL subqueries are most frequently used with the Select statement.

**Syntax**

1. SELECT column\_name
2. FROM table\_name
3. WHERE column\_name expression operator
4. ( SELECT column\_name  from table\_name WHERE ... );

**Example**

Consider the EMPLOYEE table have the following records:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | John | 20 | US | 2000.00 |
| 2 | Stephan | 26 | Dubai | 1500.00 |
| 3 | David | 27 | Bangkok | 2000.00 |
| 4 | Alina | 29 | UK | 6500.00 |
| 5 | Kathrin | 34 | Bangalore | 8500.00 |
| 6 | Harry | 42 | China | 4500.00 |
| 7 | Jackson | 25 | Mizoram | 10000.00 |

The subquery with a SELECT statement will be:

1. SELECT \*
2. FROM EMPLOYEE
3. WHERE ID IN (SELECT ID
4. FROM EMPLOYEE
5. WHERE SALARY > 4500);

This would produce the following result:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 4 | Alina | 29 | UK | 6500.00 |
| 5 | Kathrin | 34 | Bangalore | 8500.00 |
| 7 | Jackson | 25 | Mizoram | 10000.00 |

## 2. Subqueries with the INSERT Statement

* SQL subquery can also be used with the Insert statement. In the insert statement, data returned from the subquery is used to insert into another table.
* In the subquery, the selected data can be modified with any of the character, date functions.

**Syntax:**

1. INSERT INTO table\_name (column1, column2, column3....)
2. SELECT \*
3. FROM table\_name
4. WHERE VALUE OPERATOR

**Example**

Consider a table EMPLOYEE\_BKP with similar as EMPLOYEE.

Now use the following syntax to copy the complete EMPLOYEE table into the EMPLOYEE\_BKP table.

1. INSERT INTO EMPLOYEE\_BKP
2. SELECT \* FROM EMPLOYEE
3. WHERE ID IN (SELECT ID
4. FROM EMPLOYEE);

## 3. Subqueries with the UPDATE Statement

The subquery of SQL can be used in conjunction with the Update statement. When a subquery is used with the Update statement, then either single or multiple columns in a table can be updated.

**Syntax**

1. UPDATE table
2. SET column\_name = new\_value
3. WHERE VALUE OPERATOR
4. (SELECT COLUMN\_NAME
5. FROM TABLE\_NAME
6. WHERE condition);

**Example**

Let's assume we have an EMPLOYEE\_BKP table available which is backup of EMPLOYEE table. The given example updates the SALARY by .25 times in the EMPLOYEE table for all employee whose AGE is greater than or equal to 29.

1. UPDATE EMPLOYEE
2. SET SALARY = SALARY \* 0.25
3. WHERE AGE IN (SELECT AGE FROM CUSTOMERS\_BKP
4. WHERE AGE >= 29);

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | John | 20 | US | 2000.00 |
| 2 | Stephan | 26 | Dubai | 1500.00 |
| 3 | David | 27 | Bangkok | 2000.00 |
| 4 | Alina | 29 | UK | 1625.00 |
| 5 | Kathrin | 34 | Bangalore | 2125.00 |
| 6 | Harry | 42 | China | 1125.00 |
| 7 | Jackson | 25 | Mizoram | 10000.00 |

## 4. Subqueries with the DELETE Statement

The subquery of SQL can be used in conjunction with the Delete statement just like any other statements mentioned above.

**Syntax**

1. DELETE FROM TABLE\_NAME
2. WHERE VALUE OPERATOR
3. (SELECT COLUMN\_NAME
4. FROM TABLE\_NAME
5. WHERE condition);

**Example**

Let's assume we have an EMPLOYEE\_BKP table available which is backup of EMPLOYEE table. The given example deletes the records from the EMPLOYEE table for all EMPLOYEE whose AGE is greater than or equal to 29.

1. DELETE FROM EMPLOYEE
2. WHERE AGE IN (SELECT AGE FROM EMPLOYEE\_BKP
3. WHERE AGE >= 29 );

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | John | 20 | US | 2000.00 |
| 2 | Stephan | 26 | Dubai | 1500.00 |
| 3 | David | 27 | Bangkok | 2000.00 |
| 7 | Jackson | 25 | Mizoram | 10000.00 |

**Auto Increment in SQL**

In Oracle, you will have to create an auto-increment field with the sequence object (this object generates a number sequence)

Use the following CREATE SEQUENCE syntax :

CREATE SEQUENCE seq\_student

MINVALUE 1

START WITH 1

INCREMENT BY 1

CACHE 10

Here, The code above creates a sequence object called seq\_student, that starts with 1 and will increment by 1.

It will also cache up to 10 values for the performance. The cache option specifies that, how many sequence values will be stored in memory for faster access

Now to insert a new record into the table "Students", we will have to use the function nextval (this function retrieves the next value from seq\_student sequence)

Let's look at the following SQL Statement:

INSERT INTO Students (Student\_ID,FirstName,LastName)

VALUES (seq\_student.nextval,'Richard','Jones')

The above SQL statement would insert a new record into the table "Students". The column "Student\_ID" will be assigned the next number from the seq\_student sequence. The column "FirstName" will be set to "Richard" and the column "LastName" will be set to "Jones"

## SQL CREATE INDEX Statement

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

**Note:** Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

### CREATE INDEX Syntax

Creates an index on a table. Duplicate values are allowed:

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

### CREATE UNIQUE INDEX Syntax

Creates a unique index on a table. Duplicate values are not allowed:

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

**Note:** The syntax for creating indexes varies among different databases. Therefore: Check the syntax for creating indexes in your database.

## CREATE INDEX Example

The SQL statement below creates an index named "idx\_lastname" on the "LastName" column in the "Persons" table:

CREATE INDEX idx\_lastname  
ON Persons (LastName);

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

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

## DROP INDEX Statement

The DROP INDEX statement is used to delete an index in a table.

DROP INDEX index\_name;

## SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

### CREATE VIEW Syntax

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

**Note:** A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

## SQL CREATE VIEW Examples

The following SQL creates a view that shows all customers from Brazil:

### Example

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

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_view1)

We can query the view above as follows:

### Example

SELECT \* FROM [Brazil Customers];

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_view2)

The following SQL creates a view that selects every product in the "Products" table with a price higher than the average price:

### Example

CREATE VIEW [Products Above Average Price] AS  
SELECT ProductName, Price  
FROM Products  
WHERE Price > (SELECT AVG(Price) FROM Products);

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_view3)

We can query the view above as follows:

### Example

SELECT \* FROM [Products Above Average Price];

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_view4)

## SQL Updating a View

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

### SQL CREATE OR REPLACE VIEW Syntax

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

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

### Example

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

[Try it Yourself »](https://www.w3schools.com/sql/trysql.asp?filename=trysql_view5)

## SQL Dropping a View

A view is deleted with the DROP VIEW command.

### SQL DROP VIEW Syntax

DROP VIEW view\_name;

The following SQL drops the "Brazil Customers" view:

### Example

DROP VIEW [Brazil Customers];

# SQL - Transactions

A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

A transaction is the propagation of one or more changes to the database. For example, if you are creating a record or updating a record or deleting a record from the table, then you are performing a transaction on that table. It is important to control these transactions to ensure the data integrity and to handle database errors.

Practically, you will club many SQL queries into a group and you will execute all of them together as a part of a transaction.

## Properties of Transactions

Transactions have the following four standard properties, usually referred to by the acronym **ACID**.

* **Atomicity** − ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state.
* **Consistency** − ensures that the database properly changes states upon a successfully committed transaction.
* **Isolation** − enables transactions to operate independently of and transparent to each other.
* **Durability** − ensures that the result or effect of a committed transaction persists in case of a system failure.

### Transaction Control

The following commands are used to control transactions.

* **COMMIT** − to save the changes.
* **ROLLBACK** − to roll back the changes.
* **SAVEPOINT** − creates points within the groups of transactions in which to ROLLBACK.
* **SET TRANSACTION** − Places a name on a transaction.

## Transactional Control Commands

Transactional control commands are only used with the **DML Commands** such as - INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

### The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for the COMMIT command is as follows.

COMMIT;

**Example**

Consider the CUSTOMERS table having the following records −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

Following is an example which would delete those records from the table which have age = 25 and then COMMIT the changes in the database.

SQL> DELETE FROM CUSTOMERS

WHERE AGE = 25;

SQL> COMMIT;

Thus, two rows from the table would be deleted and the SELECT statement would produce the following result.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

### The ROLLBACK Command

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for a ROLLBACK command is as follows −

ROLLBACK;

**Example**

Consider the CUSTOMERS table having the following records −

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

Following is an example, which would delete those records from the table which have the age = 25 and then ROLLBACK the changes in the database.

SQL> DELETE FROM CUSTOMERS

WHERE AGE = 25;

SQL> ROLLBACK;

Thus, the delete operation would not impact the table and the SELECT statement would produce the following result.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

### The SAVEPOINT Command

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for a SAVEPOINT command is as shown below.

SAVEPOINT SAVEPOINT\_NAME;

This command serves only in the creation of a SAVEPOINT among all the transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as shown below.

ROLLBACK TO SAVEPOINT\_NAME;

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.

**Example**

Consider the CUSTOMERS table having the following records.

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

The following code block contains the series of operations.

SQL> SAVEPOINT SP1;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=1;

1 row deleted.

SQL> SAVEPOINT SP2;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=2;

1 row deleted.

SQL> SAVEPOINT SP3;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=3;

1 row deleted.

Now that the three deletions have taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone −

SQL> ROLLBACK TO SP2;

Rollback complete.

Notice that only the first deletion took place since you rolled back to SP2.

SQL> SELECT \* FROM CUSTOMERS;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 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 |

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

6 rows selected.

### The RELEASE SAVEPOINT Command

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The syntax for a RELEASE SAVEPOINT command is as follows.

RELEASE SAVEPOINT SAVEPOINT\_NAME;

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT.

# PL/SQL - Overview

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are certain notable facts about PL/SQL −

* PL/SQL is a completely portable, high-performance transaction-processing language.
* PL/SQL provides a built-in, interpreted and OS independent programming environment.
* PL/SQL can also directly be called from the command-line **SQL\*Plus interface**.
* Direct call can also be made from external programming language calls to database.
* PL/SQL's general syntax is based on that of ADA and Pascal programming language.
* Apart from Oracle, PL/SQL is available in **TimesTen in-memory database** and **IBM DB2**.

## Features of PL/SQL

PL/SQL has the following features −

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

## Advantages of PL/SQL

PL/SQL has the following advantages −

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

Text Editor

Running large programs from the command prompt may land you in inadvertently losing some of the work. It is always recommended to use the command files. To use the command files −

* Type your code in a text editor, like **Notepad, Notepad+,** or **EditPlus**, etc.
* Save the file with the **.sql** extension in the home directory.
* Launch the **SQL\*Plus command prompt** from the directory where you created your PL/SQL file.
* Type **@file\_name** at the SQL\*Plus command prompt to execute your program.

If you are not using a file to execute the PL/SQL scripts, then simply copy your PL/SQL code and right-click on the black window that displays the SQL prompt; use the **paste** option to paste the complete code at the command prompt. Finally, just press **Enter** to execute the code, if it is not already executed.

# PL/SQL - Basic Syntax

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

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

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

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling>

END;

## The 'Hello World' Example

DECLARE

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

BEGIN

dbms\_output.put\_line(message);

END;

/

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

Hello World

PL/SQL procedure successfully completed.

## The PL/SQL Identifiers

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

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

## The PL/SQL Delimiters

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

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

## The PL/SQL Comments

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

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

DECLARE

-- variable declaration

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

BEGIN

/\*

\* PL/SQL executable statement(s)

\*/

dbms\_output.put\_line(message);

END;

/

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

Hello World

PL/SQL procedure successfully completed.

## PL/SQL Program Units

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

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

# PL/SQL - Data Types

In this chapter, we will discuss the Data Types in PL/SQL. The PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, constraints, and a valid range of values. We will focus on the **SCALAR** and the **LOB** data types in this chapter. The other two data types will be covered in other chapters.

|  |  |
| --- | --- |
| **S.No** | **Category & Description** |
| 1 | **Scalar**  Single values with no internal components, such as a **NUMBER, DATE,** or **BOOLEAN**. |
| 2 | **Large Object (LOB)**  Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms. |
| 3 | **Composite**  Data items that have internal components that can be accessed individually. For example, collections and records. |
| 4 | **Reference**  Pointers to other data items. |

## PL/SQL Scalar Data Types and Subtypes

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

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

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

## PL/SQL Numeric Data Types and Subtypes

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

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

Following is a valid declaration −

DECLARE

num1 INTEGER;

num2 REAL;

num3 DOUBLE PRECISION;

BEGIN

null;

END;

/

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

PL/SQL procedure successfully completed

## PL/SQL Character Data Types and Subtypes

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

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

## PL/SQL Boolean Data Types

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

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

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

## PL/SQL Datetime and Interval Types

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

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

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

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

## PL/SQL Large Object (LOB) Data Types

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

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

## PL/SQL User-Defined Subtypes

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

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

SUBTYPE CHARACTER IS CHAR;

SUBTYPE INTEGER IS NUMBER(38,0);

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

DECLARE

SUBTYPE name IS char(20);

SUBTYPE message IS varchar2(100);

salutation name;

greetings message;

BEGIN

salutation := 'Reader ';

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

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

END;

/

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

Hello Reader Welcome to the World of PL/SQL

PL/SQL procedure successfully completed.

## NULLs in PL/SQL

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

# PL/SQL - Variables

In this chapter, we will discuss Variables in Pl/SQL. A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in PL/SQL has a specific data type, which determines the size and the layout of the variable's memory; the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The name of a PL/SQL variable consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. By default, variable names are not case-sensitive. You cannot use a reserved PL/SQL keyword as a variable name.

PL/SQL programming language allows to define various types of variables, such as date time data types, records, collections, etc. which we will cover in subsequent chapters. For this chapter, let us study only basic variable types.

## Variable Declaration in PL/SQL

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

The syntax for declaring a variable is −

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

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

sales number(10, 2);

pi CONSTANT double precision := 3.1415;

name varchar2(25);

address varchar2(100);

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

sales number(10, 2);

name varchar2(25);

address varchar2(100);

## Initializing Variables in PL/SQL

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

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

For example −

counter binary\_integer := 0;

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

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

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

DECLARE

a integer := 10;

b integer := 20;

c integer;

f real;

BEGIN

c := a + b;

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

f := 70.0/3.0;

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

END;

/

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

Value of c: 30

Value of f: 23.333333333333333333

PL/SQL procedure successfully completed.

## Variable Scope in PL/SQL

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

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

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

DECLARE

-- Global variables

num1 number := 95;

num2 number := 85;

BEGIN

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

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

DECLARE

-- Local variables

num1 number := 195;

num2 number := 185;

BEGIN

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

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

END;

END;

/

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

Outer Variable num1: 95

Outer Variable num2: 85

Inner Variable num1: 195

Inner Variable num2: 185

PL/SQL procedure successfully completed.

## Assigning SQL Query Results to PL/SQL Variables

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

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

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25),

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

Table Created

Let us now insert some values in the table −

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

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

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

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

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

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

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

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

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

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

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

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

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

DECLARE

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

c\_name customers.name%type;

c\_addr customers.address%type;

c\_sal customers.salary%type;

BEGIN

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

FROM customers

WHERE id = c\_id;

dbms\_output.put\_line

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

END;

/

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

Customer Ramesh from Ahmedabad earns 2000

PL/SQL procedure completed successfully

# PL/SQL - Constants and Literals

In this chapter, we will discuss **constants** and **literals** in PL/SQL. A constant holds a value that once declared, does not change in the program. A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the **NOT NULL constraint**.

## Declaring a Constant

A constant is declared using the **CONSTANT** keyword. It requires an initial value and does not allow that value to be changed. For example −

PI CONSTANT NUMBER := 3.141592654;

DECLARE

-- constant declaration

pi constant number := 3.141592654;

-- other declarations

radius number(5,2);

dia number(5,2);

circumference number(7, 2);

area number (10, 2);

BEGIN

-- processing

radius := 9.5;

dia := radius \* 2;

circumference := 2.0 \* pi \* radius;

area := pi \* radius \* radius;

-- output

dbms\_output.put\_line('Radius: ' || radius);

dbms\_output.put\_line('Diameter: ' || dia);

dbms\_output.put\_line('Circumference: ' || circumference);

dbms\_output.put\_line('Area: ' || area);

END;

/

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

Radius: 9.5

Diameter: 19

Circumference: 59.69

Area: 283.53

Pl/SQL procedure successfully completed.

## The PL/SQL Literals

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. For example, TRUE, 786, NULL, 'tutorialspoint' are all literals of type Boolean, number, or string. PL/SQL, literals are case-sensitive. PL/SQL supports the following kinds of literals −

* Numeric Literals
* Character Literals
* String Literals
* BOOLEAN Literals
* Date and Time Literals

The following table provides examples from all these categories of literal values.

|  |  |
| --- | --- |
| **S.No** | **Literal Type & Example** |
| 1 | **Numeric Literals**  050 78 -14 0 +32767  6.6667 0.0 -12.0 3.14159 +7800.00  6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3 |
| 2 | **Character Literals**  'A' '%' '9' ' ' 'z' '(' |
| 3 | **String Literals**  'Hello, world!'  'Tutorials Point'  '19-NOV-12' |
| 4 | **BOOLEAN Literals**  TRUE, FALSE, and NULL. |
| 5 | **Date and Time Literals**  DATE '1978-12-25';  TIMESTAMP '2012-10-29 12:01:01'; |

To embed single quotes within a string literal, place two single quotes next to each other as shown in the following program −

DECLARE

message varchar2(30):= 'That''s tutorialspoint.com!';

BEGIN

dbms\_output.put\_line(message);

END;

/

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

That's tutorialspoint.com!

PL/SQL procedure successfully completed.

# PL/SQL - Conditions

In this chapter, we will discuss conditions in PL/SQL. Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

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



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

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

# PL/SQL - IF-THEN Statement

It is the simplest form of the **IF** control statement, frequently used in decision-making and changing the control flow of the program execution.

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

## Syntax

Syntax for **IF-THEN** statement is −

IF condition THEN

S;

END IF;

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

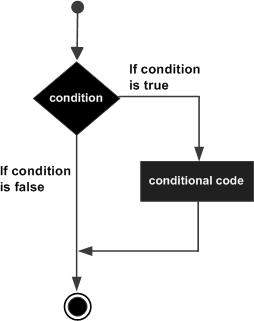
IF (a <= 20) THEN

c:= c+1;

END IF;

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

## Flow Diagram



## Example 1

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

DECLARE

a number(2) := 10;

BEGIN

a:= 10;

-- check the boolean condition using if statement

IF( a < 20 ) THEN

-- if condition is true then print the following

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

END IF;

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

END;

/

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

a is less than 20

value of a is : 10

PL/SQL procedure successfully completed.

## Example 2

Consider we have a table and few records in the table as we had created in [PL/SQL Variable Types](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm)

DECLARE

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

c\_sal customers.salary%type;

BEGIN

SELECT salary

INTO c\_sal

FROM customers

WHERE id = c\_id;

IF (c\_sal <= 2000) THEN

UPDATE customers

SET salary = salary + 1000

WHERE id = c\_id;

dbms\_output.put\_line ('Salary updated');

END IF;

END;

/

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

Salary updated

PL/SQL procedure successfully completed.

# PL/SQL - IF-THEN-ELSE Statement

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

## Syntax

Syntax for the IF-THEN-ELSE statement is −

IF condition THEN

S1;

ELSE

S2;

END IF;

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

IF color = red THEN

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

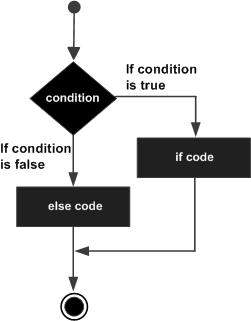
ELSE

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

END IF;

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

## Flow Diagram



## Example

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

DECLARE

a number(3) := 100;

BEGIN

-- check the boolean condition using if statement

IF( a < 20 ) THEN

-- if condition is true then print the following

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

ELSE

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

END IF;

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

END;

/

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

a is not less than 20

value of a is : 100

PL/SQL procedure successfully completed.

# PL/SQL - IF-THEN-ELSIF Statement

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

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

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

## Syntax

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

IF(boolean\_expression 1)THEN

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

ELSIF( boolean\_expression 2) THEN

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

ELSIF( boolean\_expression 3) THEN

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

ELSE

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

END IF;

## Example

DECLARE

a number(3) := 100;

BEGIN

IF ( a = 10 ) THEN

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

ELSIF ( a = 20 ) THEN

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

ELSIF ( a = 30 ) THEN

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

ELSE

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

END IF;

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

END;

/

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

None of the values is matching

Exact value of a is: 100

PL/SQL procedure successfully completed.

# PL/SQL - CASE Statement

Like the **IF** statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the **CASE** statement uses a selector rather than multiple Boolean expressions. A selector is an expression, the value of which is used to select one of several alternatives.

## Syntax

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

CASE selector

WHEN 'value1' THEN S1;

WHEN 'value2' THEN S2;

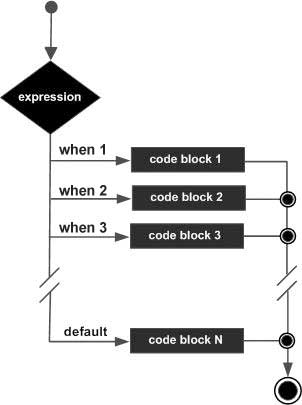
WHEN 'value3' THEN S3;

...

ELSE Sn; -- default case

END CASE;

## Flow Diagram



## Example

DECLARE

grade char(1) := 'A';

BEGIN

CASE grade

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

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

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

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

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

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

END CASE;

END;

/

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

Excellent

PL/SQL procedure successfully completed.

# PL/SQL - Searched CASE Statement

Advertisements

[Previous Page](https://www.tutorialspoint.com/plsql/plsql_conditional_control.htm)

[Next Page](https://www.tutorialspoint.com/plsql/plsql_conditional_control.htm)

The searched **CASE** statement has no selector and the **WHEN** clauses of the statement contain search conditions that give Boolean values.

## Syntax

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

CASE

WHEN selector = 'value1' THEN S1;

WHEN selector = 'value2' THEN S2;

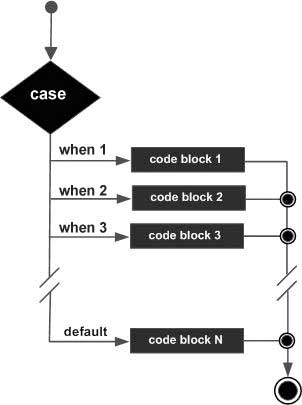
WHEN selector = 'value3' THEN S3;

...

ELSE Sn; -- default case

END CASE;

## Flow Diagram



## Example

DECLARE

grade char(1) := 'B';

BEGIN

case

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

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

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

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

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

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

end case;

END;

/

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

Very good

PL/SQL procedure successfully completed.

# PL/SQL - Nested IF-THEN-ELSE Statements

It is always legal in PL/SQL programming to nest the **IF-ELSE** statements, which means you can use one **IF** or **ELSE IF** statement inside another **IF** or **ELSE IF** statement(s).

## Syntax

IF( boolean\_expression 1)THEN

-- executes when the boolean expression 1 is true

IF(boolean\_expression 2) THEN

-- executes when the boolean expression 2 is true

sequence-of-statements;

END IF;

ELSE

-- executes when the boolean expression 1 is not true

else-statements;

END IF;

## Example

DECLARE

a number(3) := 100;

b number(3) := 200;

BEGIN

-- check the boolean condition

IF( a = 100 ) THEN

-- if condition is true then check the following

IF( b = 200 ) THEN

-- if condition is true then print the following

dbms\_output.put\_line('Value of a is 100 and b is 200' );

END IF;

END IF;

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

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

END;

/

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

Value of a is 100 and b is 200

Exact value of a is : 100

Exact value of b is : 200

PL/SQL procedure successfully completed.

# PL/SQL - Loops

In this chapter, we will discuss Loops in PL/SQL. There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

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



# PL/SQL - Basic Loop Statement

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

## Syntax

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

LOOP

Sequence of statements;

END LOOP;

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

## Example

DECLARE

x number := 10;

BEGIN

LOOP

dbms\_output.put\_line(x);

x := x + 10;

IF x > 50 THEN

exit;

END IF;

END LOOP;

-- after exit, control resumes here

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

END;

/

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

10

20

30

40

50

After Exit x is: 60

PL/SQL procedure successfully completed.

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

DECLARE

x number := 10;

BEGIN

LOOP

dbms\_output.put\_line(x);

x := x + 10;

exit WHEN x > 50;

END LOOP;

-- after exit, control resumes here

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

END;

/

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

10

20

30

40

50

After Exit x is: 60

PL/SQL procedure successfully completed.

# PL/SQL - WHILE LOOP Statement

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

## Syntax

WHILE condition LOOP

sequence\_of\_statements

END LOOP;

## Example

DECLARE

a number(2) := 10;

BEGIN

WHILE a < 20 LOOP

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

a := a + 1;

END LOOP;

END;

/

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

PL/SQL procedure successfully completed.

# PL/SQL - FOR LOOP Statement

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

## Syntax

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

sequence\_of\_statements;

END LOOP;

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

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

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

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

## Example

DECLARE

a number(2);

BEGIN

FOR a in 10 .. 20 LOOP

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

END LOOP;

END;

/

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

value of a: 20

PL/SQL procedure successfully completed.

## Reverse FOR LOOP Statement

By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the **REVERSE** keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented.

However, you must write the range bounds in ascending (not descending) order. The following program illustrates this −

DECLARE

a number(2) ;

BEGIN

FOR a IN REVERSE 10 .. 20 LOOP

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

END LOOP;

END;

/

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

value of a: 20

value of a: 19

value of a: 18

value of a: 17

value of a: 16

value of a: 15

value of a: 14

value of a: 13

value of a: 12

value of a: 11

value of a: 10

PL/SQL procedure successfully completed.

# PL/SQL - Nested Loops

Advertisements

[Previous Page](https://www.tutorialspoint.com/plsql/plsql_loops.htm)

[Next Page](https://www.tutorialspoint.com/plsql/plsql_loops.htm)

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

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

LOOP

Sequence of statements1

LOOP

Sequence of statements2

END LOOP;

END LOOP;

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

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

sequence\_of\_statements1

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

sequence\_of\_statements2

END LOOP;

END LOOP;

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

WHILE condition1 LOOP

sequence\_of\_statements1

WHILE condition2 LOOP

sequence\_of\_statements2

END LOOP;

END LOOP;

## Example

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

DECLARE

i number(3);

j number(3);

BEGIN

i := 2;

LOOP

j:= 2;

LOOP

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

j := j +1;

END LOOP;

IF (j = i ) THEN

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

END IF;

i := i + 1;

exit WHEN i = 50;

END LOOP;

END;

/

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

2 is prime

3 is prime

5 is prime

7 is prime

11 is prime

13 is prime

17 is prime

19 is prime

23 is prime

29 is prime

31 is prime

37 is prime

41 is prime

43 is prime

47 is prime

PL/SQL procedure successfully completed.

The Loop Control Statements

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

# PL/SQL - EXIT Statement

The **EXIT** statement in PL/SQL programming language has the following two usages −

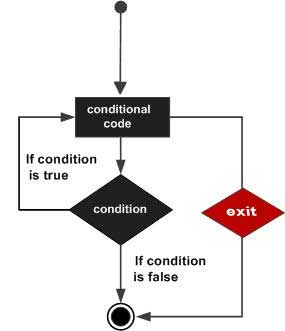
* When the EXIT statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
* If you are using nested loops (i.e., one loop inside another loop), the EXIT statement will stop the execution of the innermost loop and start executing the next line of code after the block.

### Syntax

The syntax for an EXIT statement in PL/SQL is as follows −

EXIT;

### Flow Diagram



### Example

DECLARE

a number(2) := 10;

BEGIN

-- while loop execution

WHILE a < 20 LOOP

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

a := a + 1;

IF a > 15 THEN

-- terminate the loop using the exit statement

EXIT;

END IF;

END LOOP;

END;

/

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

PL/SQL procedure successfully completed.

## The EXIT WHEN Statement

The **EXIT-WHEN** statement allows the condition in the WHEN clause to be evaluated. If the condition is true, the loop completes and control passes to the statement immediately after the END LOOP.

Following are the two important aspects for the EXIT WHEN statement −

* Until the condition is true, the EXIT-WHEN statement acts like a NULL statement, except for evaluating the condition, and does not terminate the loop.
* A statement inside the loop must change the value of the condition.

### Syntax

The syntax for an EXIT WHEN statement in PL/SQL is as follows −

EXIT WHEN condition;

The EXIT WHEN statement **replaces a conditional statement like if-then** used with the EXIT statement.

### Example

DECLARE

a number(2) := 10;

BEGIN

-- while loop execution

WHILE a < 20 LOOP

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

a := a + 1;

-- terminate the loop using the exit when statement

EXIT WHEN a > 15;

END LOOP;

END;

/

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

PL/SQL procedure successfully completed.

# PL/SQL - CONTINUE Statement

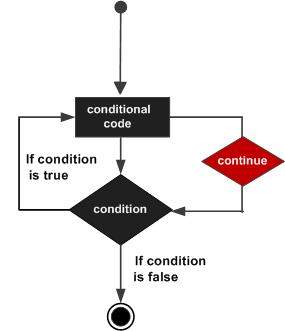
The **CONTINUE** statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. In other words, it forces the next iteration of the loop to take place, skipping any code in between.

## Syntax

The syntax for a CONTINUE statement is as follows −

CONTINUE;

## Flow Diagram



## Example

DECLARE

a number(2) := 10;

BEGIN

-- while loop execution

WHILE a < 20 LOOP

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

a := a + 1;

IF a = 15 THEN

-- skip the loop using the CONTINUE statement

a := a + 1;

CONTINUE;

END IF;

END LOOP;

END;

/

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 16

value of a: 17

value of a: 18

value of a: 19

PL/SQL procedure successfully completed.

# PL/SQL - GOTO Statement

A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

**NOTE** − The use of GOTO statement is not recommended in any programming language because it makes it difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

### Syntax

The syntax for a GOTO statement in PL/SQL is as follows −

GOTO label;

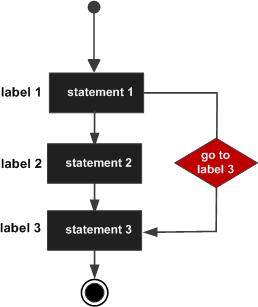
..

..

<< label >>

statement;

### Flow Diagram



### Example

DECLARE

a number(2) := 10;

BEGIN

<<loopstart>>

-- while loop execution

WHILE a < 20 LOOP

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

a := a + 1;

IF a = 15 THEN

a := a + 1;

GOTO loopstart;

END IF;

END LOOP;

END;

/

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 16

value of a: 17

value of a: 18

value of a: 19

PL/SQL procedure successfully completed.

## Restrictions with GOTO Statement

GOTO Statement in PL/SQL imposes the following restrictions −

* A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement or sub-block.
* A GOTO statement cannot branch from one IF statement clause to another or from one CASE statement WHEN clause to another.
* A GOTO statement cannot branch from an outer block into a sub-block (i.e., an inner BEGIN-END block).
* A GOTO statement cannot branch out of a subprogram. To end a subprogram early, either use the RETURN statement or have GOTO branch to a place right before the end of the subprogram.
* A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block

# PL/SQL - Operators

In this chapter, we will discuss operators in PL/SQL. An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following types of operators −

* Arithmetic operators
* Relational operators
* Comparison operators
* Logical operators
* String operators

Here, we will understand the arithmetic, relational, comparison and logical operators one by one. The String operators will be discussed in a later chapter − **PL/SQL - Strings**.

## Arithmetic Operators

Following table shows all the arithmetic operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 5, then −

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_arithmetic_operators.htm" \o "Arithmetic Operators in PL/SQL)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | A + B will give 15 |
| - | Subtracts second operand from the first | A - B will give 5 |
| \* | Multiplies both operands | A \* B will give 50 |
| / | Divides numerator by de-numerator | A / B will give 2 |
| \*\* | Exponentiation operator, raises one operand to the power of other | A \*\* B will give 100000 |

## Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 20, then −

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_relational_operators.htm" \o "Relational Operators in PL/SQL)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A = B) is not true. |
| !=  <>  ~= | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true |

## Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either **TRUE, FALSE** or **NULL**.

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_comparison_operators.htm" \o "Comparison Operators in PL/SQL)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| LIKE | The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not. | If 'Zara Ali' like 'Z% A\_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A\_i' returns a Boolean false. |
| BETWEEN | The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a and x <= b. | If x = 10 then, x between 5 and 20 returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false. |
| IN | The IN operator tests set membership. x IN (set) means that x is equal to any member of set. | If x = 'm' then, x in ('a', 'b', 'c') returns Boolean false but x in ('m', 'n', 'o') returns Boolean true. |
| IS NULL | The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL. | If x = 'm', then 'x is null' returns Boolean false. |

## Logical Operators

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produce Boolean results. Let us assume **variable A** holds true and **variable B** holds false, then −

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_logical_operators.htm" \o "Logical Operators in PL/SQL)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Examples** |
| and | Called the logical AND operator. If both the operands are true then condition becomes true. | (A and B) is false. |
| or | Called the logical OR Operator. If any of the two operands is true then condition becomes true. | (A or B) is true. |
| not | Called the logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false. | not (A and B) is true. |

## PL/SQL Operator Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, **x = 7 + 3 \* 2**; here, **x** is assigned **13**, not 20 because operator \* has higher precedence than +, so it first gets multiplied with **3\*2** and then adds into **7**.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

The precedence of operators goes as follows: =, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN.

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_operators_precedence.htm" \o "Operators Precedence in PL/SQL)

|  |  |
| --- | --- |
| **Operator** | **Operation** |
| \*\* | exponentiation |
| +, - | identity, negation |
| \*, / | multiplication, division |
| +, -, || | addition, subtraction, concatenation |
| comparison |  |
| NOT | logical negation |
| AND | conjunction |
| OR | inclusion |

# PL/SQL - Strings

The string in PL/SQL is actually a sequence of characters with an optional size specification. The characters could be numeric, letters, blank, special characters or a combination of all. PL/SQL offers three kinds of strings −

* **Fixed-length strings** − In such strings, programmers specify the length while declaring the string. The string is right-padded with spaces to the length so specified.
* **Variable-length strings** − In such strings, a maximum length up to 32,767, for the string is specified and no padding takes place.
* **Character large objects (CLOBs)** − These are variable-length strings that can be up to 128 terabytes.

PL/SQL strings could be either variables or literals. A string literal is enclosed within quotation marks. For example,

'This is a string literal.' Or 'hello world'

To include a single quote inside a string literal, you need to type two single quotes next to one another. For example,

'this isn''t what it looks like'

## Declaring String Variables

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

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

DECLARE

name varchar2(20);

company varchar2(30);

introduction clob;

choice char(1);

BEGIN

name := 'John Smith';

company := 'Infotech';

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

choice := 'y';

IF choice = 'y' THEN

dbms\_output.put\_line(name);

dbms\_output.put\_line(company);

dbms\_output.put\_line(introduction);

END IF;

END;

/

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

John Smith

Infotech

Hello! I'm John Smith from Infotech.

PL/SQL procedure successfully completed

To declare a fixed-length string, use the CHAR datatype. Here you do not have to specify a maximum length for a fixed-length variable. If you leave off the length constraint, Oracle Database automatically uses a maximum length required. The following two declarations are identical −

red\_flag CHAR(1) := 'Y';

red\_flag CHAR := 'Y';

## PL/SQL String Functions and Operators

PL/SQL offers the concatenation operator **(||)** for joining two strings. The following table provides the string functions provided by PL/SQL −

|  |  |
| --- | --- |
| **S.No** | **Function & Purpose** |
| 1 | **ASCII(x);**  Returns the ASCII value of the character x. |
| 2 | **CHR(x);**  Returns the character with the ASCII value of x. |
| 3 | **CONCAT(x, y);**  Concatenates the strings x and y and returns the appended string. |
| 4 | **INITCAP(x);**  Converts the initial letter of each word in x to uppercase and returns that string. |
| 5 | **INSTR(x, find\_string [, start] [, occurrence]);**  Searches for **find\_string** in x and returns the position at which it occurs. |
| 6 | **INSTRB(x);**  Returns the location of a string within another string, but returns the value in bytes. |
| 7 | **LENGTH(x);**  Returns the number of characters in x. |
| 8 | **LENGTHB(x);**  Returns the length of a character string in bytes for single byte character set. |
| 9 | **LOWER(x);**  Converts the letters in x to lowercase and returns that string. |
| 10 | **LPAD(x, width [, pad\_string]) ;**  Pads **x** with spaces to the left, to bring the total length of the string up to width characters. |
| 11 | **LTRIM(x [, trim\_string]);**  Trims characters from the left of **x**. |
| 12 | **NANVL(x, value);**  Returns value if x matches the NaN special value (not a number), otherwise **x** is returned. |
| 13 | **NLS\_INITCAP(x);**  Same as the INITCAP function except that it can use a different sort method as specified by NLSSORT. |
| 14 | **NLS\_LOWER(x) ;**  Same as the LOWER function except that it can use a different sort method as specified by NLSSORT. |
| 15 | **NLS\_UPPER(x);**  Same as the UPPER function except that it can use a different sort method as specified by NLSSORT. |
| 16 | **NLSSORT(x);**  Changes the method of sorting the characters. Must be specified before any NLS function; otherwise, the default sort will be used. |
| 17 | **NVL(x, value);**  Returns value if **x** is null; otherwise, x is returned. |
| 18 | **NVL2(x, value1, value2);**  Returns value1 if x is not null; if x is null, value2 is returned. |
| 19 | **REPLACE(x, search\_string, replace\_string);**  Searches **x** for search\_string and replaces it with replace\_string. |
| 20 | **RPAD(x, width [, pad\_string]);**  Pads **x** to the right. |
| 21 | **RTRIM(x [, trim\_string]);**  Trims **x** from the right. |
| 22 | **SOUNDEX(x) ;**  Returns a string containing the phonetic representation of **x**. |
| 23 | **SUBSTR(x, start [, length]);**  Returns a substring of **x** that begins at the position specified by start. An optional length for the substring may be supplied. |
| 24 | **SUBSTRB(x);**  Same as SUBSTR except that the parameters are expressed in bytes instead of characters for the single-byte character systems. |
| 25 | **TRIM([trim\_char FROM) x);**  Trims characters from the left and right of **x**. |
| 26 | **UPPER(x);**  Converts the letters in x to uppercase and returns that string. |

Let us now work out on a few examples to understand the concept −

### Example 1

DECLARE

greetings varchar2(11) := 'hello world';

BEGIN

dbms\_output.put\_line(UPPER(greetings));

dbms\_output.put\_line(LOWER(greetings));

dbms\_output.put\_line(INITCAP(greetings));

/\* retrieve the first character in the string \*/

dbms\_output.put\_line ( SUBSTR (greetings, 1, 1));

/\* retrieve the last character in the string \*/

dbms\_output.put\_line ( SUBSTR (greetings, -1, 1));

/\* retrieve five characters,

starting from the seventh position. \*/

dbms\_output.put\_line ( SUBSTR (greetings, 7, 5));

/\* retrieve the remainder of the string,

starting from the second position. \*/

dbms\_output.put\_line ( SUBSTR (greetings, 2));

/\* find the location of the first "e" \*/

dbms\_output.put\_line ( INSTR (greetings, 'e'));

END;

/

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

HELLO WORLD

hello world

Hello World

h

d

World

ello World

2

PL/SQL procedure successfully completed.

### Example 2

DECLARE

greetings varchar2(30) := '......Hello World.....';

BEGIN

dbms\_output.put\_line(RTRIM(greetings,'.'));

dbms\_output.put\_line(LTRIM(greetings, '.'));

dbms\_output.put\_line(TRIM( '.' from greetings));

END;

/

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

......Hello World

Hello World.....

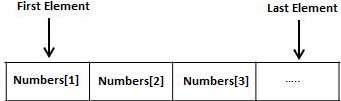
Hello World

PL/SQL procedure successfully completed.

# PL/SQL - Arrays

In this chapter, we will discuss arrays in PL/SQL. The PL/SQL programming language provides a data structure called the **VARRAY**, which can store a fixed-size sequential collection of elements of the same type. A varray is used to store an ordered collection of data, however it is often better to think of an array as a collection of variables of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



An array is a part of collection type data and it stands for variable-size arrays. We will study other collection types in a later chapter **'PL/SQL Collections'**.

Each element in a **varray** has an index associated with it. It also has a maximum size that can be changed dynamically.

## Creating a Varray Type

A varray type is created with the **CREATE TYPE** statement. You must specify the maximum size and the type of elements stored in the varray.

The basic syntax for creating a VARRAY type at the schema level is −

CREATE OR REPLACE TYPE varray\_type\_name IS VARRAY(n) of <element\_type>

Where,

* *varray\_type\_name* is a valid attribute name,
* *n* is the number of elements (maximum) in the varray,
* *element\_type* is the data type of the elements of the array.

Maximum size of a varray can be changed using the **ALTER TYPE** statement.

For example,

CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);

/

Type created.

The basic syntax for creating a VARRAY type within a PL/SQL block is −

TYPE varray\_type\_name IS VARRAY(n) of <element\_type>

For example −

TYPE namearray IS VARRAY(5) OF VARCHAR2(10);

Type grades IS VARRAY(5) OF INTEGER;

Let us now work out on a few examples to understand the concept −

### Example 1

The following program illustrates the use of varrays −

DECLARE

type namesarray IS VARRAY(5) OF VARCHAR2(10);

type grades IS VARRAY(5) OF INTEGER;

names namesarray;

marks grades;

total integer;

BEGIN

names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');

marks:= grades(98, 97, 78, 87, 92);

total := names.count;

dbms\_output.put\_line('Total '|| total || ' Students');

FOR i in 1 .. total LOOP

dbms\_output.put\_line('Student: ' || names(i) || '

Marks: ' || marks(i));

END LOOP;

END;

/

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

Total 5 Students

Student: Kavita Marks: 98

Student: Pritam Marks: 97

Student: Ayan Marks: 78

Student: Rishav Marks: 87

Student: Aziz Marks: 92

PL/SQL procedure successfully completed.

**Please note** −

* In Oracle environment, the starting index for varrays is always 1.
* You can initialize the varray elements using the constructor method of the varray type, which has the same name as the varray.
* Varrays are one-dimensional arrays.
* A varray is automatically NULL when it is declared and must be initialized before its elements can be referenced.

### Example 2

Elements of a varray could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept.

We will use the CUSTOMERS table stored in our database as −

Select \* from customers;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

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

Following example makes the use of **cursor**, which you will study in detail in a separate chapter.

DECLARE

CURSOR c\_customers is

SELECT name FROM customers;

type c\_list is varray (6) of customers.name%type;

name\_list c\_list := c\_list();

counter integer :=0;

BEGIN

FOR n IN c\_customers LOOP

counter := counter + 1;

name\_list.extend;

name\_list(counter) := n.name;

dbms\_output.put\_line('Customer('||counter ||'):'||name\_list(counter));

END LOOP;

END;

/

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

Customer(1): Ramesh

Customer(2): Khilan

Customer(3): kaushik

Customer(4): Chaitali

Customer(5): Hardik

Customer(6): Komal

PL/SQL procedure successfully completed.

# 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.

Parameter Modes in PL/SQL Subprograms

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

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

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

## PL/SQL Create Procedure

**Syntax for creating procedure:**

1. **CREATE** [OR REPLACE] **PROCEDURE** procedure\_name
2. [ (parameter [,parameter]) ]
3. **IS**
4. [declaration\_section]
5. **BEGIN**
6. executable\_section
7. [EXCEPTION
8. exception\_section]
9. **END** [procedure\_name];

## Create procedure example

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

**Table creation:**

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

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

**Procedure Code:**

1. **create** or replace **procedure** "INSERTUSER"
2. (id IN NUMBER,
3. **name** IN VARCHAR2)
4. **is**
5. **begin**
6. **insert** **into** user **values**(id,**name**);
7. **end**;
8. /

Output:

Procedure created.

## PL/SQL program to call procedure

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

1. **BEGIN**
2. insertuser(101,'Rahul');
3. dbms\_output.put\_line('record inserted successfully');
4. **END**;
5. /

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

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

## PL/SQL Drop Procedure

**Syntax for drop procedure**

1. **DROP** **PROCEDURE** procedure\_name;

## Example of drop procedure

1. **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 you are going to return from the function.
* Function\_body contains the executable part.
* The AS keyword is used instead of the IS keyword for creating a standalone function.

## 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.

#### Create customers table and have records in it.

|  |  |  |  |
| --- | --- | --- | --- |
| **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:**

1. **CREATE** OR REPLACE **FUNCTION** totalCustomers
2. **RETURN** number **IS**
3. total number(2) := 0;
4. **BEGIN**
5. **SELECT** count(\*) **into** total
6. **FROM** customers;
7. **RETURN** total;
8. **END**;
9. /

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

Function created.

**Calling PL/SQL Function:**

While creating a function, you have to give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. 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 you have to pass the required parameters along with function name and if function returns a value then you can store returned value. Following program calls the function totalCustomers from an anonymous block:

1. **DECLARE**
2. c number(2);
3. **BEGIN**
4. c := totalCustomers();
5. dbms\_output.put\_line('Total no. of Customers: ' || c);
6. **END**;
7. /

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

Total no. of Customers: 4

PL/SQL procedure successfully completed.

## PL/SQL Recursive Function

You already know that a program or a subprogram can call another subprogram. When a subprogram calls itself, it is called recursive call and the process is known as recursion.

## Example to calculate the factorial of a number

Let's take an example to calculate the factorial of a number. This example calculates the factorial of a given number by calling itself recursively.

1. **DECLARE**
2. num number;
3. factorial number;
5. **FUNCTION** fact(x number)
6. **RETURN** number
7. **IS**
8. f number;
9. **BEGIN**
10. IF x=0 **THEN**
11. f := 1;
12. **ELSE**
13. f := x \* fact(x-1);
14. **END** IF;
15. **RETURN** f;
16. **END**;
18. **BEGIN**
19. num:= 6;
20. factorial := fact(num);
21. dbms\_output.put\_line(' Factorial '|| num || ' is ' || factorial);
22. **END**;
23. /

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

Factorial 6 is 720

PL/SQL procedure successfully completed.

## PL/SQL Drop Function

**Syntax for removing your created function:**

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

1. **DROP** **FUNCTION** function\_name;

# 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 you 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 you 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 you 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 you check the records in customer table, you 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:

You 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. You 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.

# Exception Handling in PL/SQL

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

There are two types of exceptions defined in PL/SQL

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

Syntax to write an exception

**WHEN** exception **THEN**

statement;

*DECLARE  
declarations section;*

*BEGIN  
executable command(s);*

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

*END;*

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

1. **System defined exceptions:**  
   These exceptions are predefined in PL/SQL which get raised WHEN certain **database rule is violated.**  
   System-defined exceptions are further divided into two categories:
   1. Named system exceptions.
   2. Unnamed system exceptions.
   3. **Named system exceptions:** They have a predefined name by the system like ACCESS\_INTO\_NULL, DUP\_VAL\_ON\_INDEX, LOGIN\_DENIED etc. the list is quite big.

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

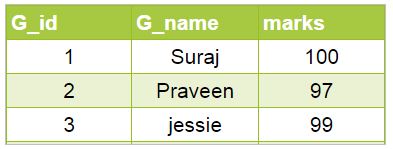
Lets create a table geeks.

create table geeks(g\_id int , g\_name varchar(20), marks int);

insert into geeks values(1, 'Suraj',100);

insert into geeks values(2, 'Praveen',97);

insert into geeks values(3, 'Jessie', 99);



* + 1. **NO\_DATA\_FOUND**: It is raised WHEN a SELECT INTO statement returns no rows. For eg:

filter\_none

brightness\_4

|  |
| --- |
| DECLARE     temp varchar(20);    BEGIN     SELECT g\_id into temp from geeks where g\_name='GeeksforGeeks';    exception     WHEN no\_data\_found THEN        dbms\_output.put\_line('ERROR');        dbms\_output.put\_line('there is no name as');        dbms\_output.put\_line('GeeksforGeeks in geeks table');  end; |

Output:

ERROR

there is no name as GeeksforGeeks in geeks table

* + 1. **TOO\_MANY\_ROWS**:It is raised WHEN a SELECT INTO statement returns more than one row.

filter\_none

brightness\_4

|  |
| --- |
| DECLARE     temp varchar(20);    BEGIN    -- raises an exception as SELECT  -- into trying to return too many rows     SELECT g\_name into temp from geeks;     dbms\_output.put\_line(temp);    EXCEPTION     WHEN too\_many\_rows THEN        dbms\_output.put\_line('error trying to SELECT too many rows');    end; |

Output:

error trying to SELECT too many rows

* + 1. **VALUE\_ERROR**:This error is raised WHEN a statement is executed that resulted in an arithmetic, numeric, string, conversion, or constraint error. This error mainly results from programmer error or invalid data input.

filter\_none

brightness\_4

|  |
| --- |
| DECLARE     temp number;    BEGIN     SELECT g\_name  into temp from geeks where g\_name='Suraj';     dbms\_output.put\_line('the g\_name is '||temp);    EXCEPTION     WHEN value\_error THEN     dbms\_output.put\_line('Error');     dbms\_output.put\_line('Change data type of temp to varchar(20)');    END; |

Output:

Error

Change data type of temp to varchar(20)

* + 1. **ZERO\_DIVIDE** = raises exception WHEN dividing with zero.

filter\_none

brightness\_4

|  |
| --- |
| DECLARE     a int:=10;     b int:=0;     answer int;    BEGIN     answer:=a/b;     dbms\_output.put\_line('the result after division is'||answer);    exception     WHEN zero\_divide THEN        dbms\_output.put\_line('dividing by zero please check the values again');        dbms\_output.put\_line('the value of a is '||a);        dbms\_output.put\_line('the value of b is '||b);  END; |

Output:

dividing by zero please check the values again

the value of a is 10

the value of b is 0

* 1. **Unnamed system exceptions:**Oracle doesn’t provide name for some system exceptions called unnamed system exceptions.These exceptions don’t occur frequently.These exceptions have two parts code and an associated message.  
     The way to handle to these exceptions is to assign name to them using **Pragma EXCEPTION\_INIT**  
     Syntax:
  2. PRAGMA EXCEPTION\_INIT(exception\_name, -error\_number);

error\_number are pre-defined and have negative integer range from -20000 to -20999.

**Example:**

filter\_none

brightness\_4

|  |
| --- |
| DECLARE     exp exception;     pragma exception\_init (exp, -20015);     n int:=10;    BEGIN     FOR i IN 1..n LOOP        dbms\_output.put\_line(i\*i);           IF i\*i=36 THEN              RAISE exp;           END IF;     END LOOP;    EXCEPTION     WHEN exp THEN        dbms\_output.put\_line('Welcome to GeeksforGeeks');    END; |

Output:

1

4

9

16

25

36

Welcome to GeeksforGeeks

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

Example:

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

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

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

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| DECLARE     x int:=&x; /\*taking value at run time\*/     y int:=&y;     div\_r float;     exp1 EXCEPTION;     exp2 EXCEPTION;    BEGIN     IF y=0 then         raise exp1;       ELSEIF y > x then        raise exp2;       ELSE        div\_r:= x / y;        dbms\_output.put\_line('the result is '||div\_r);       END IF;    EXCEPTION     WHEN exp1 THEN        dbms\_output.put\_line('Error');        dbms\_output.put\_line('division by zero not allowed');       WHEN exp2 THEN        dbms\_output.put\_line('Error');        dbms\_output.put\_line('y is greater than x please check the input');    END; |

Input 1: x = 20

y = 10

Output: the result is 2

Input 2: x = 20

y = 0

Output:

Error

division by zero not allowed

Input 3: x=20

y = 30

Output:<.em>

Error

y is greater than x please check the input

# 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: You 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 you use the id defined in the above table (i.e. 1 to 6), you 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, you 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 - Triggers

In this chapter, we will discuss Triggers in PL/SQL. Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

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

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

### Benefits of Triggers

Triggers can be written for the following purposes −

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

## Creating Triggers

The syntax for creating a trigger is −

CREATE [OR REPLACE ] TRIGGER trigger\_name

{BEFORE | AFTER | INSTEAD OF }

{INSERT [OR] | UPDATE [OR] | DELETE}

[OF col\_name]

ON table\_name

[REFERENCING OLD AS o NEW AS n]

[FOR EACH ROW]

WHEN (condition)

DECLARE

Declaration-statements

BEGIN

Executable-statements

EXCEPTION

Exception-handling-statements

END;

Where,

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

### Example

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

Select \* from customers;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

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

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

CREATE OR REPLACE TRIGGER display\_salary\_changes

BEFORE DELETE OR INSERT OR UPDATE ON customers

FOR EACH ROW

WHEN (NEW.ID > 0)

DECLARE

sal\_diff number;

BEGIN

sal\_diff := :NEW.salary - :OLD.salary;

dbms\_output.put\_line('Old salary: ' || :OLD.salary);

dbms\_output.put\_line('New salary: ' || :NEW.salary);

dbms\_output.put\_line('Salary difference: ' || sal\_diff);

END;

/

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

Trigger created.

The following points need to be considered here −

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

## Triggering a Trigger

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

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

VALUES (7, 'Kriti', 22, 'HP', 7500.00 );

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

Old salary:

New salary: 7500

Salary difference:

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

UPDATE customers

SET salary = salary + 500

WHERE id = 2;

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

Old salary: 1500

New salary: 2000

Salary difference: 500

# PL/SQL - Collections

In this chapter, we will discuss the Collections in PL/SQL. A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types −

* Index-by tables or Associative array
* Nested table
* Variable-size array or Varray

## Index-By Table

An **index-by** table (also called an **associative array**) is a set of **key-value** pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here, we are creating an **index-by** table named **table\_name**, the keys of which will be of the subscript\_type and associated values will be of the *element\_type*

TYPE type\_name IS TABLE OF element\_type [NOT NULL] INDEX BY subscript\_type;

table\_name type\_name;

### Example

Following example shows how to create a table to store integer values along with names and later it prints the same list of names.

DECLARE

TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);

salary\_list salary;

name VARCHAR2(20);

BEGIN

-- adding elements to the table

salary\_list('Rajnish') := 62000;

salary\_list('Minakshi') := 75000;

salary\_list('Martin') := 100000;

salary\_list('James') := 78000;

-- printing the table

name := salary\_list.FIRST;

WHILE name IS NOT null LOOP

dbms\_output.put\_line

('Salary of ' || name || ' is ' || TO\_CHAR(salary\_list(name)));

name := salary\_list.NEXT(name);

END LOOP;

END;

/

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

Salary of James is 78000

Salary of Martin is 100000

Salary of Minakshi is 75000

Salary of Rajnish is 62000

PL/SQL procedure successfully completed.

# PL/SQL Packages

PL/SQL Packages is schema object and collection of related data type (variables, constants), cursors, procedures, functions are defining within a single context. Package are divide into two part,

1. Package Specification
2. Package Body

Package specification block you can define variables, constants, exceptions and package body you can create procedure, function, subprogram.

### PL/SQL Package Advantages

1. You can create package to store all related functions and procedures are grouped together into single unit called packages.
2. Package are reliable to granting a privileges.
3. All function and procedure within a package can share variable among them.
4. Package are support overloading to overload functions and procedures.
5. Package are improve the performance to loading the multiple object into memory at once, therefore, subsequent calls to related program doesn't required to calling physically I/O.
6. Package are reduce the traffic because all block execute all at once.

### PL/SQL Package Example

PL/SQL Package example step by step explain to you, you are create your own package using this reference example. We have emp1 table having employee information,

|  |  |  |  |
| --- | --- | --- | --- |
| EMP\_NO | EMP\_NAME | EMP\_DEPT | EMP\_SALARY |
| 1 | Forbs ross | Web Developer | 45k |
| 2 | marks jems | Program Developer | 38k |
| 3 | Saulin | Program Developer | 34k |
| 4 | Zenia Sroll | Web Developer | 42k |

## Package Specification

Create Package specification code for defining procedure, function IN or OUT parameter and execute package specification program.

CREATE or REPLACE PACKAGE pkg1

IS | AS

PROCEDURE pro1

(no in number, name out varchar2);

FUNCTION fun1

(no in number)

RETURN varchar2;

END;

/

## Package Body

Create Package body code for implementing procedure or function that are defined package specification. Once you implement execute this program.

CREATE or REPLACE PACKAGE BODY pkg1

IS

PROCEDURE pro1(no in number,info our varchar2)

IS

BEGIN

SELECT \* INTO temp FROM emp1 WHERE eno = no;

END;

FUNCTION fun1(no in number) return varchar2

IS

name varchar2(20);

BEGIN

SELECT ename INTO name FROM emp1 WHERE eno = no;

RETURN name;

END;

END;

/

## Pl/SQL Program calling Package

Now we have a one package pkg1, to call package defined function, procedures also pass the parameter and get the return result.

*pkg\_prg.sql*

DECLARE

no number := &no;

name varchar2(20);

BEGIN

pkg1.pro1(no,info);

dbms\_output.put\_line('Procedure Result');

dbms\_output.put\_line(info.eno||' '||

info.ename||' '||

info.edept||' '||

info.esalary||' '||);

dbms\_output.put\_line('Function Result');

name := pkg1.fun1(no);

dbms\_output.put\_line(name);

END;

/

Result

Now execute the above created pkg\_prg.sql program to asking which user information you want to get, you put user id and give information.

**SQL>@pkg\_prg**  
no number &n=2  
Procedure Result  
2    marks jems    Program Developer    38K  
Function Result  
marks jems  
  
PL/SQL procedure successfully completed.