# DATABASE

### Overview of Database

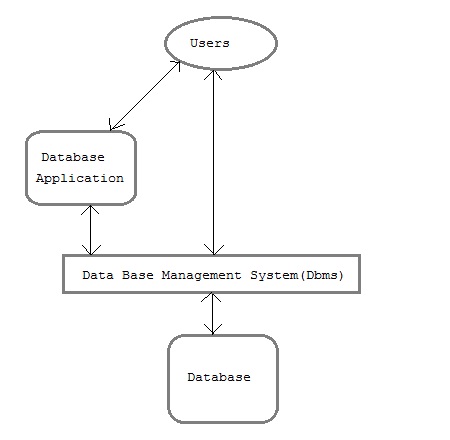
A **Database** is a collection of related data organised in a way that data can be easily accessed, managed and updated. Any piece of information can be a data, for example name of your school. Database is actualy a place where related piece of information is stored and various operations can be performed on it.

#### DBMS

A **DBMS** is a software that allows creation, definition and manipulation of database. Dbms is actualy a tool used to perform any kind of operation on data in database. Dbms also provides protection and security to database. It maintains data consistency in case of multiple users. Here are some examples of popular dbms, MySql, Oracle, Sybase, Microsoft Access and IBM DB2 etc.

#### Components of Database System

The database system can be divided into four components.



* **Users :** Users may be of various type such as DB administrator, System developer and End users.
* **Database application :** Database application may be Personal, Departmental, Enterprise and Internal
* **DBMS :** Software that allow users to define, create and manages database access, Ex: MySql, Oracle etc.
* **Database :** Collection of logical data.

#### Functions of DBMS

* Provides data Independence
* Concurrency Control
* Provides Recovery services
* Provides Utility services
* Provides a clear and logical view of the process that manipulates data.

#### Advantages of DBMS

* Segregation of applicaion program.
* Minimal data duplicacy.
* Easy retrieval of data.
* Reduced development time and maintainance need.

#### Disadvantages of DBMS

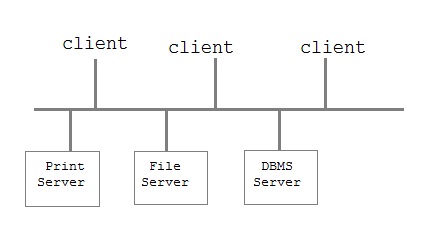
* Complexity
* Costly
* Large in size

### Database Architecture

Database architecture is logically divided into two types.

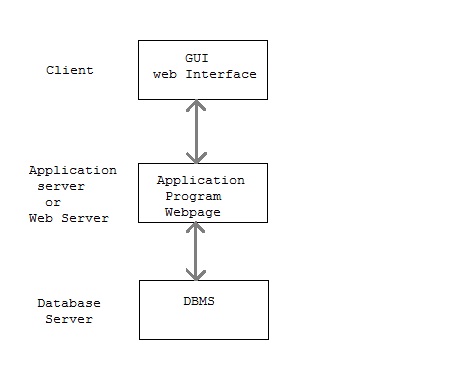
1. Logical two-tier Client / Server architecture
2. Logical three-tier Client / Server architecture

#### Two-tier Client / Server Architecture



Two-tier Client / Server architecture is used for User Interface program and Application Programs that runs on client side. An interface called ODBC(Open Database Connectivity) provides an API that allow client side program to call the dbms. Most DBMS vendors provide ODBC drivers. A client program may connect to several DBMS's. In this architecture some variation of client is also possible for example in some DBMS's more functionality is transferred to the client including data dictionary, optimization etc. Such clients are called **Data server**.

#### Three-tier Client / Server Architecture



Three-tier Client / Server database architecture is commonly used architecture for web applications. Intermediate layer called **Application server** or Web Server stores the web connectivty software and the business logic(constraints) part of application used to access the right amount of data from the database server. This layer acts like medium for sending partially processed data between the database server and the client.

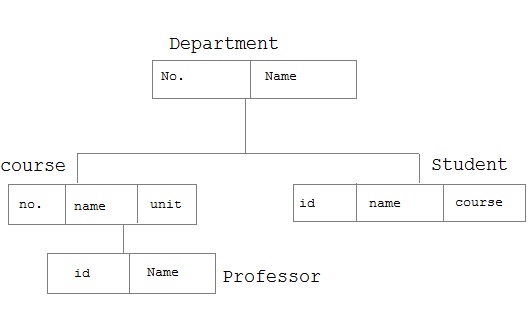
### Database Model

A Database model defines the logical design of data. The model describes the relationships between different parts of the data. Historically, in database design, three models are commonly used. They are,

* Hierarchical Model
* Network Model
* Relational Model

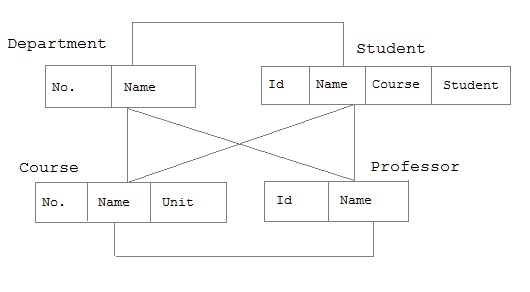
#### Hierarchical Model

In this model each entity has only one parent but can have several children . At the top of hierarchy there is only one entity which is called **Root**.



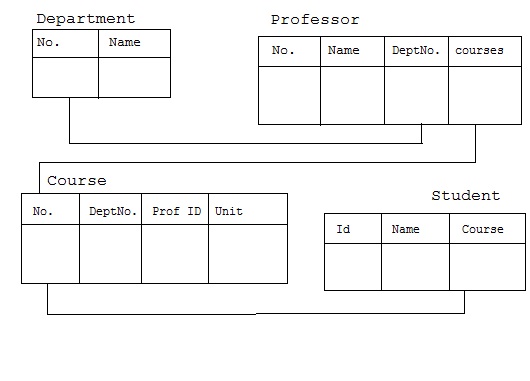
#### Network Model

In the network model, entities are organised in a graph,in which some entities can be accessed through several path



#### Relational Model

In this model, data is organised in two-dimesional tables called **relations**. The tables or relation are related to each other.



### Codd's Rule

E.F Codd was a Computer Scientist who invented **Relational model** for Database management. Based on relational model, **Relation database** was created. Codd proposed 13 rules popularly known as **Codd's 12 rules** to test DBMS's concept against his relational model. Codd's rule actualy define what quality a DBMS requires in order to become a Relational Database Management System(RDBMS). Till now, there is hardly any commercial product that follows all the 13 Codd's rules. Even **Oracle** follows only eight and half out(8.5) of 13. The Codd's 12 rules are as follows.

#### Rule zero

This rule states that for a system to qualify as an **RDBMS**, it must be able to manage database entirely through the relational capabilities.

#### Rule 1 : Information rule

All information(including metadata) is to be represented as stored data in cells of tables. The rows and columns have to be strictly unordered.

#### Rule 2 : Guaranted Access

Each unique piece of data(atomic value) should be accesible by : **Table Name + primary key(Row) + Attribute(column)**.

**NOTE :** Ability to directly access via POINTER is a violation of this rule.

#### Rule 3 : Systemetic treatment of NULL

**Null** has several meanings, it can mean missing data, not applicable or no value. It should be handled consistently. Primary key must not be null. Expression on **NULL** must give null.

#### Rule 4 : Active Online Catalog

Database dictionary(catalog) must have description of **Database**. Catalog to be governed by same rule as rest of the database. The same query language to be used on catalog as on application database.

#### Rule 5 : Powerful language

One well defined language must be there to provide all manners of access to data. Example: **SQL**. If a file supporting table can be accessed by any manner except SQL interface, then its a violation to this rule.

#### Rule 6 : View Updation rule

All view that are theoretically updatable should be updatable by the system.

#### Rule 7 : Relational Level Operation

There must be Insert, Delete, Update operations at each level of relations. Set operation like Union, Intersection and minus should also be supported.

#### Rule 8 : Physical Data Independence

The physical storage of data should not matter to the system. If say, some file supporting table were renamed or moved from one disk to another, it should not effect the application.

#### Rule 9 : Logical Data Independence

If there is change in the logical structure(table structures) of the database the user view of data should not change. Say, if a table is split into two tables, a new view should give result as the join of the two tables. This rule is most difficult to satisfy.

#### Rule 10 : Integrity Independence

The database should be able to conforce its own integrity rather than using other programs. Key and Check constraints, trigger etc should be stored in Data Dictionary. This also make **RDBMS** independent of front-end.

#### Rule 11 : Distribution Independence

A database should work properly regardless of its distribution across a network. This lays foundation of distributed database.

#### Rule 12 : Nonsubversion rule

If low level access is allowed to a system it should not be able to subvert or bypass integrity rule to change data. This can be achieved by some sort of looking or encryption.

### RDBMS Concepts

A **Relational Database management System**(RDBMS) is a database management system based on relational model introduced by E.F Codd. In relational model, data is represented in terms of tuples(rows).

**RDBMS** is used to manage Relational database. **Relational database** is a collection of organized set of tables from which data can be accessed easily. Relational Database is most commonly used database. It consists of number of tables and each table has its own primary key.

#### What is Table ?

In Relational database, a **table** is a collection of data elements organised in terms of rows and columns. A table is also considered as convenient representation of **relations**. But a table can have duplicate tuples while a true **relation** cannot have duplicate tuples. Table is the most simplest form of data storage. Below is an example of Employee table.

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Name** | **Age** | **Salary** |
| 1 | Adam | 34 | 13000 |
| 2 | Alex | 28 | 15000 |
| 3 | Stuart | 20 | 18000 |
| 4 | Ross | 42 | 19020 |

#### What is a Record ?

A single entry in a table is called a **Record** or **Row**. A **Record** in a table represents set of related data. For example, the above **Employee** table has 4 records. Following is an example of single record.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | Adam | 34 | 13000 |

#### What is Field ?

A table consists of several records(row), each record can be broken into several smaller entities known as **Fields**. The above **Employee** table consist of four fields, **ID**, **Name**, **Age** and **Salary**.

#### What is a Column ?

In **Relational** table, a column is a set of value of a particular type. The term **Attribute** is also used to represent a column. For example, in Employee table, Name is a column that represent names of employee.

|  |
| --- |
| **Name** |
| Adam |
| Alex |
| Stuart |
| Ross |

### Database Keys

Keys are very important part of Relational database. They are used to establish and identify relation between tables. They also ensure that each record within a table can be uniquely identified by combination of one or more fields within a table.

#### Super Key

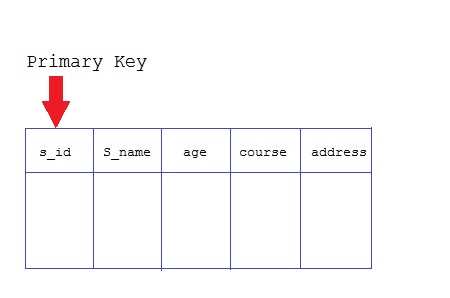
**Super Key** is defined as a set of attributes within a table that uniquely identifies each record within a table. Super Key is a superset of Candidate key.

#### Candidate Key

Candidate keys are defined as the set of fields from which primary key can be selected. It is an attribute or set of attribute that can act as a primary key for a table to uniquely identify each record in that table.

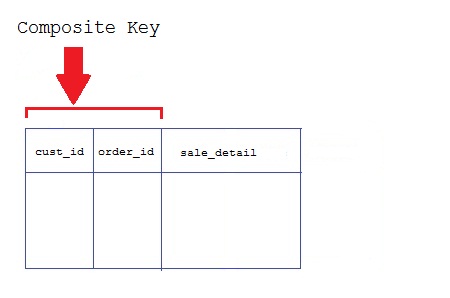
#### Primary Key

Primary key is a candidate key that is most appropriate to become main key of the table. It is a key that uniquely identify each record in a table.



#### Composite Key

Key that consist of two or more attributes that uniquely identify an entity occurance is called **Composite key**. But any attribute that makes up the **Composite key** is not a simple key in its own.



#### Secondary or Alternative key

The candidate key which are not selected for primary key are known as secondary keys or alternative keys

#### Non-key Attribute

**Non-key** attributes are attributes other than **candidate key** attributes in a table.

#### Non-prime Attribute

**Non-prime** Attributes are attributes other than **Primary attribute**.

### Normalization of Database

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

Normalization is used for mainly two purpose,

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

#### Problem Without Normalization

Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anamolies are very frequent if Database is not Normalized. To understand these anomalies let us take an example of **Student** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_id** | **S\_Name** | **S\_Address** | **Subject\_opted** |
| 401 | Adam | Noida | Bio |
| 402 | Alex | Panipat | Maths |
| 403 | Stuart | Jammu | Maths |
| 404 | Adam | Noida | Physics |

* **Updation Anamoly :** To update address of a student who occurs twice or more than twice in a table, we will have to update **S\_Address** column in all the rows, else data will become inconsistent.
* **Insertion Anamoly :** Suppose for a new admission, we have a Student id(S\_id), name and address of a student but if student has not opted for any subjects yet then we have to insert **NULL** there, leading to Insertion Anamoly.
* **Deletion Anamoly :** If (S\_id) 401 has only one subject and temporarily he drops it, when we delete that row, entire student record will be deleted along with it.

#### Normalization Rule

Normalization rule are divided into following normal form.

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

#### First Normal Form (1NF)

As per First Normal Form, no two Rows of data must contain repeating group of information i.e each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row. Each table should be organized into rows, and each row should have a primary key that distinguishes it as unique.

The **Primary key** is usually a single column, but sometimes more than one column can be combined to create a single primary key. For example consider a table which is not in First normal form

**Student Table :**

|  |  |  |
| --- | --- | --- |
| **Student** | **Age** | **Subject** |
| Adam | 15 | Biology, Maths |
| Alex | 14 | Maths |
| Stuart | 17 | Maths |

In First Normal Form, any row must not have a column in which more than one value is saved, like separated with commas. Rather than that, we must separate such data into multiple rows.

**Student Table following 1NF will be :**

|  |  |  |
| --- | --- | --- |
| **Student** | **Age** | **Subject** |
| Adam | 15 | Biology |
| Adam | 15 | Maths |
| Alex | 14 | Maths |
| Stuart | 17 | Maths |

Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

#### Second Normal Form (2NF)

As per the Second Normal Form there must not be any partial dependency of any column on primary key. It means that for a table that has concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence. If any column depends only on one part of the concatenated key, then the table fails **Second normal form**.

In example of First Normal Form there are two rows for Adam, to include multiple subjects that he has opted for. While this is searchable, and follows First normal form, it is an inefficient use of space. Also in the above Table in First Normal Form, while the candidate key is {**Student**, **Subject**}, **Age** of Student only depends on Student column, which is incorrect as per Second Normal Form. To achieve second normal form, it would be helpful to split out the subjects into an independent table, and match them up using the student names as foreign keys.

**New Student Table following 2NF will be :**

|  |  |
| --- | --- |
| **Student** | **Age** |
| Adam | 15 |
| Alex | 14 |
| Stuart | 17 |

In Student Table the candidate key will be **Student** column, because all other column i.e **Age** is dependent on it.

**New Subject Table introduced for 2NF will be :**

|  |  |
| --- | --- |
| **Student** | **Subject** |
| Adam | Biology |
| Adam | Maths |
| Alex | Maths |
| Stuart | Maths |

In Subject Table the candidate key will be {**Student**, **Subject**} column. Now, both the above tables qualifies for Second Normal Form and will never suffer from Update Anomalies. Although there are a few complex cases in which table in Second Normal Form suffers Update Anomalies, and to handle those scenarios Third Normal Form is there.

#### Third Normal Form (3NF)

**Third Normal form** applies that every non-prime attribute of table must be dependent on primary key, or we can say that, there should not be the case that a non-prime attribute is determined by another non-prime attribute. So this *transitive functional dependency* should be removed from the table and also the table must be in **Second Normal form**. For example, consider a table with following fields.

**Student\_Detail Table :**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Student\_id** | **Student\_name** | **DOB** | **Street** | **city** | **State** | **Zip** |

In this table Student\_id is Primary key, but street, city and state depends upon Zip. The dependency between zip and other fields is called **transitive dependency**. Hence to apply **3NF**, we need to move the street, city and state to new table, with **Zip** as primary key.

**New Student\_Detail Table :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_id** | **Student\_name** | **DOB** | **Zip** |

**Address Table :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Zip** | **Street** | **city** | **state** |

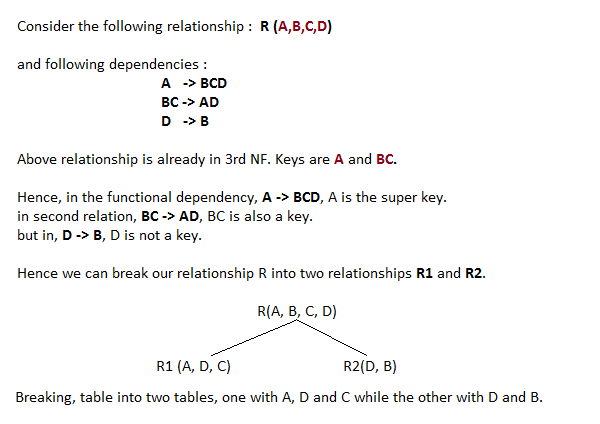
The advantage of removing transtive dependency is,

* Amount of data duplication is reduced.
* Data integrity achieved.

#### Boyce and Codd Normal Form (BCNF)

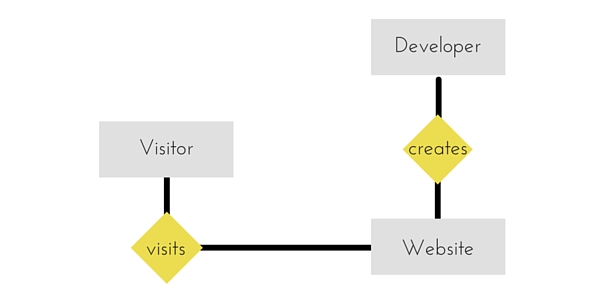
**Boyce and Codd Normal Form** is a higher version of the Third Normal form. This form deals with certain type of anamoly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF. For a table to be in BCNF, following conditions must be satisfied:

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

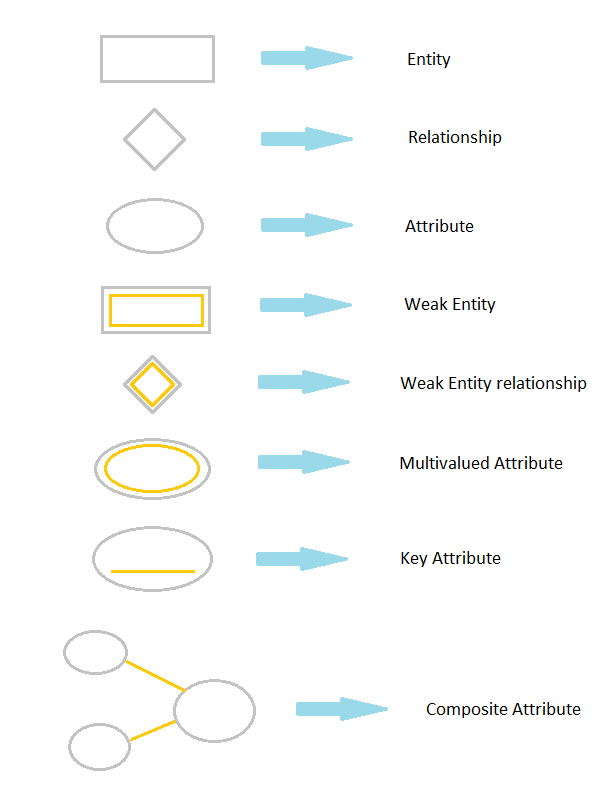


### E-R Diagram

ER-Diagram is a visual representation of data that describes how data is related to each other.



#### Symbols and Notations

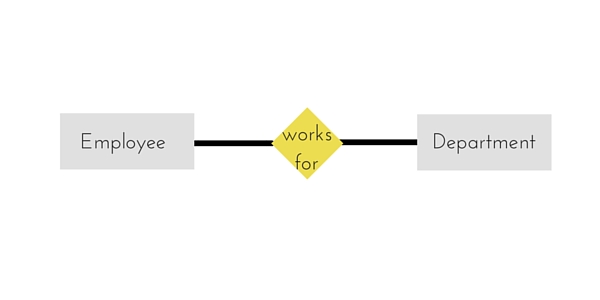


### Components of E-R Diagram

The E-R diagram has three main components.

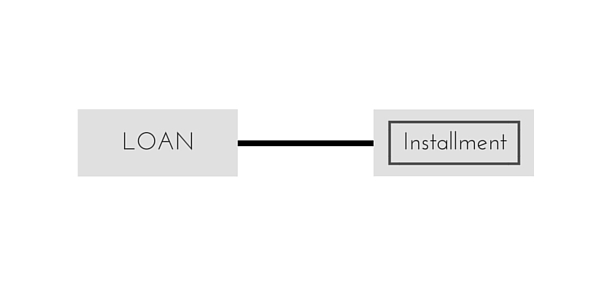
#### 1) Entity

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



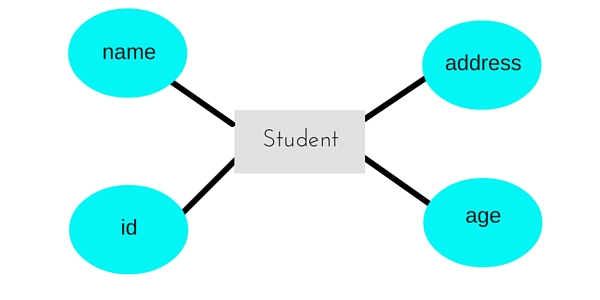
#### Weak Entity

Weak entity is an entity that depends on another entity. Weak entity doen't have key attribute of their own. Double rectangle represents weak entity.



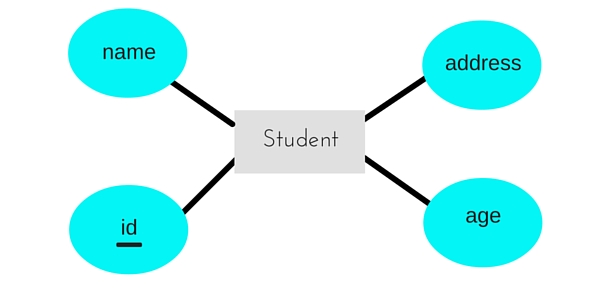
#### 2) Attribute

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



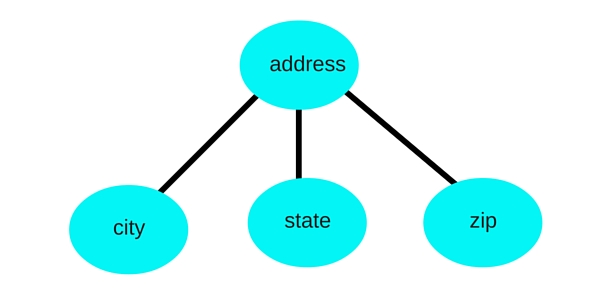
#### Key Attribute

Key attribute represents the main characterstic of an Entity. It is used to represent Primary key. Ellipse with underlying lines represent Key Attribute.



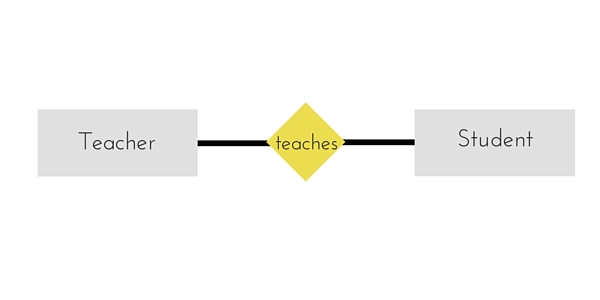
#### Composite Attribute

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



#### 3) Relationship

A Relationship describes relations between **entities**. Relationship is represented using diamonds.



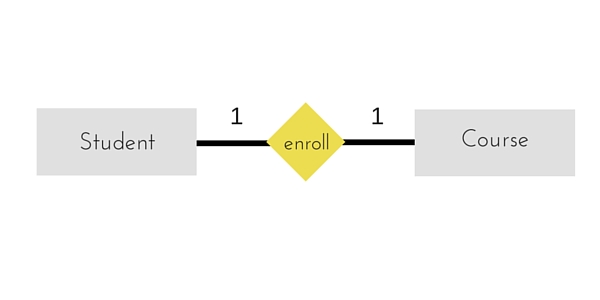
There are three types of relationship that exist between Entities.

* Binary Relationship
* Recursive Relationship
* Ternary Relationship

#### Binary Relationship

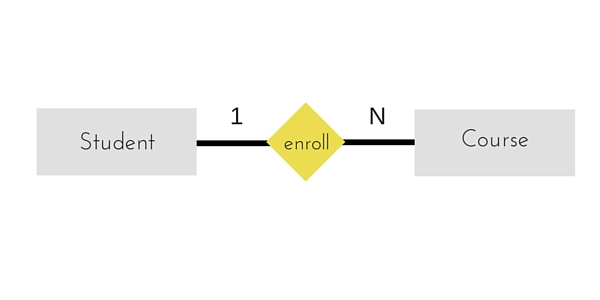
Binary Relationship means relation between two Entities. This is further divided into three types.

1. **One to One :** This type of relationship is rarely seen in real world.



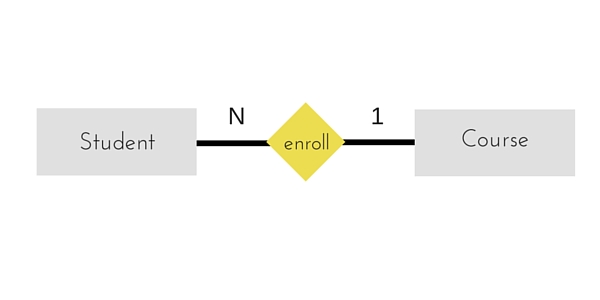
The above example describes that one student can enroll only for one course and a course will also have only one Student. This is not what you will usually see in relationship.

1. **One to Many :** It reflects business rule that one entity is associated with many number of same entity. The example for this relation might sound a little weird, but this menas that one student can enroll to many courses, but one course will have one Student.

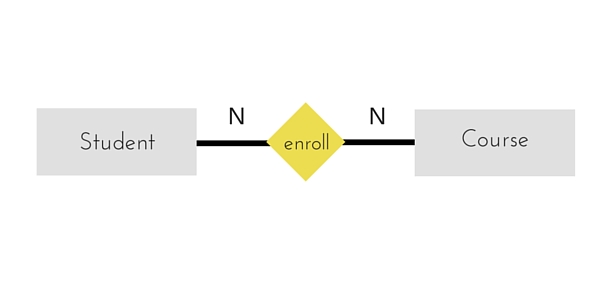


The arrows in the diagram describes that one student can enroll for only one course.

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



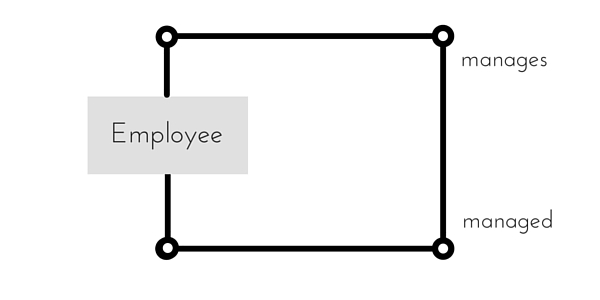
1. **Many to Many :**



The above diagram represents that many students can enroll for more than one courses.

#### Recursive Relationship

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

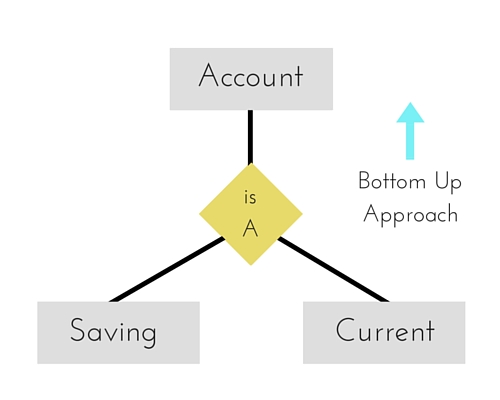


#### Ternary Relationship

Relationship of degree three is called Ternary relationship.

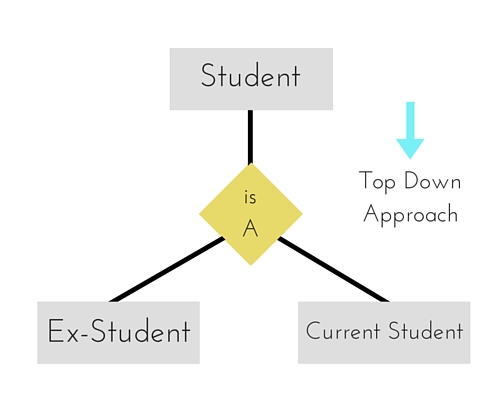
#### Generalization

**Generalization** is a bottom-up approach in which two lower level entities combine to form a higher level entity. In generalization, the higher level entity can also combine with other lower level entity to make further higher level entity.



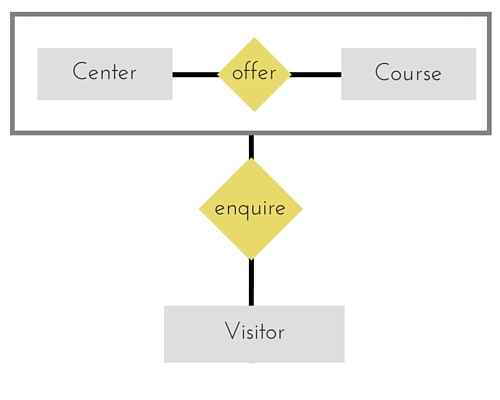
#### Specialization

**Specialization** is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into two lower level entity. In specialization, some higher level entities may not have lower-level entity sets at all.



#### Aggregration

Aggregration is a process when relation between two entity is treated as a single entity. Here the relation between Center and Course, is acting as an Entity in relation with Visitor.



# SQL Concept

### Introduction to SQL

Structure Query Language(SQL) is a programming language used for storing and managing data in RDBMS. SQL was the first commercial language introduced for E.F Codd's **Relational** model. Today almost all RDBMS(MySql, Oracle, Infomix, Sybase, MS Access) uses **SQL** as the standard database language. SQL is used to perform all type of data operations in RDBMS.

#### SQL Command

SQL defines following data languages to manipulate data of RDBMS.

#### DDL : Data Definition Language

All DDL commands are auto-committed. That means it saves all the changes permanently in the database.

|  |  |
| --- | --- |
| **Command** | **Description** |
| create | to create new table or database |
| alter | for alteration |
| truncate | delete data from table |
| drop | to drop a table |
| rename | to rename a table |

#### DML : Data Manipulation Language

DML commands are not auto-committed. It means changes are not permanent to database, they can be rolled back.

|  |  |
| --- | --- |
| **Command** | **Description** |
| insert | to insert a new row |
| update | to update existing row |
| delete | to delete a row |
| merge | merging two rows or two tables |

#### TCL : Transaction Control Language

These commands are to keep a check on other commands and their affect on the database. These commands can annul changes made by other commands by rolling back to original state. It can also make changes permanent.

|  |  |
| --- | --- |
| **Command** | **Description** |
| commit | to permanently save |
| rollback | to undo change |
| savepoint | to save temporarily |

#### DCL : Data Control Language

Data control language provides command to grant and take back authority.

|  |  |
| --- | --- |
| **Command** | **Description** |
| grant | grant permission of right |
| revoke | take back permission. |

#### DQL : Data Query Language

|  |  |
| --- | --- |
| **Command** | **Description** |
| select | retrieve records from one or more table |

## DDL Command

### create command

**create** is a DDL command used to create a table or a database.

#### Creating a Database

To create a database in RDBMS, *create* command is uses. Following is the Syntax,

**create** database *database-name*;

#### Example for Creating Database

create database Test;

The above command will create a database named **Test**.

#### Creating a Table

*create* command is also used to create a table. We can specify names and datatypes of various columns along.Following is the Syntax,

**create** table *table-name*

{

*column-name1* datatype1,

*column-name2* datatype2,

*column-name3* datatype3,

*column-name4* datatype4

};

create table command will tell the database system to create a new table with given table name and column information.

#### Example for creating Table

create table Student(id int, name varchar, age int);

The above command will create a new table **Student** in database system with 3 columns, namely id, name and age.

### alter command

*alter* command is used for alteration of table structures. There are various uses of *alter* command, such as,

* to add a column to existing table
* to rename any existing column
* to change datatype of any column or to modify its size.
* *alter* is also used to drop a column.

#### To Add Column to existing Table

Using alter command we can add a column to an existing table. Following is the Syntax,

**alter** table *table-name* add(**column-name** *datatype*);

Here is an Example for this,

alter table Student add(address char);

The above command will add a new column *address* to the **Student** table

#### To Add Multiple Column to existing Table

Using alter command we can even add multiple columns to an existing table. Following is the Syntax,

**alter** table *table-name* add(**column-name1** *datatype1*, **column-name2** *datatype2*, **column-name3** *datatype3*);

Here is an Example for this,

alter table Student add(father-name varchar(60), mother-name varchar(60), dob date);

The above command will add three new columns to the **Student** table

#### To Add column with Default Value

alter command can add a new column to an existing table with default values. Following is the Syntax,

**alter** table *table-name* add(**column-name1** *datatype1* **default** *data*);

Here is an Example for this,

alter table Student add(dob date default '1-Jan-99');

The above command will add a new column with default value to the **Student** table

#### To Modify an existing Column

alter command is used to modify data type of an existing column . Following is the Syntax,

**alter** table *table-name* modify(**column-name** *datatype*);

Here is an Example for this,

alter table Student modify(address varchar(30));

The above command will modify *address* column of the **Student table**

#### To Rename a column

Using alter command you can rename an existing column. Following is the Syntax,

**alter** table *table-name* **rename** old-column-name to column-name;

Here is an Example for this,

alter table Student rename address to Location;

The above command will rename *address* column to *Location*.

#### To Drop a Column

alter command is also used to drop columns also. Following is the Syntax,

**alter** table *table-name* drop(column-name);

Here is an Example for this,

alter table Student drop(address);

The above command will drop *address* column from the **Student table**

### SQL queries to Truncate, Drop or Rename a Table

#### truncate command

*truncate* command removes all records from a table. But this command will not destroy the table's structure. When we apply truncate command on a table its Primary key is initialized. Following is its Syntax,

**truncate** table *table-name*

Here is an Example explaining it.

truncate table Student;

The above query will delete all the records of **Student** table.

**truncate** command is different from **delete** command. delete command will delete all the rows from a table whereas truncate command re-initializes a table(like a newly created table).

**For eg.** If you have a table with 10 rows and an auto\_increment primary key, if you use *delete* command to delete all the rows, it will delete all the rows, but will not initialize the primary key, hence if you will insert any row after using delete command, the auto\_increment primary key will start from 11. But in case of *truncate* command, primary key is re-initialized.

#### drop command

*drop* query completely removes a table from database. This command will also destroy the table structure. Following is its Syntax,

**drop** table *table-name*

Here is an Example explaining it.

drop table Student;

The above query will delete the **Student** table completely. It can also be used on Databases. For Example, to drop a database,

drop database Test;

The above query will drop a database named **Test** from the system.

#### rename query

*rename* command is used to rename a table. Following is its Syntax,

**rename** table *old-table-name* to *new-table-name*

Here is an Example explaining it.

rename table Student to Student-record;

The above query will rename **Student** table to **Student-record**.

# DML Command

### DML command

Data Manipulation Language (DML) statements are used for managing data in database. DML commands are not auto-committed. It means changes made by DML command are not permanent to database, it can be rolled back.

#### 1) INSERT command

Insert command is used to insert data into a table. Following is its general syntax,

**INSERT** into *table-name* values(data1,data2,..)

Lets see an example,

Consider a table **Student** with following fields.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |

INSERT into Student values(101,'Adam',15);

The above command will insert a record into **Student** table.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |

#### Example to Insert NULL value to a column

Both the statements below will insert NULL value into **age** column of the Student table.

INSERT into Student(id,name) values(102,'Alex');

Or,

INSERT into Student values(102,'Alex',null);

The above command will insert only two column value other column is set to null.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex |  |

#### Example to Insert Default value to a column

INSERT into Student values(103,'Chris',default)

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex |  |
| 103 | chris | 14 |

Suppose the **age** column of student table has default value of 14.

Also, if you run the below query, it will insert default value into the age column, whatever the default value may be.

INSERT into Student values(103,'Chris')

#### 2) UPDATE command

Update command is used to update a row of a table. Following is its general syntax,

**UPDATE** *table-name* set column-name = value *where* **condition**;

Lets see an example,

update Student set age=18 where s\_id=102;

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | chris | 14 |

#### Example to Update multiple columns

UPDATE Student set s\_name='Abhi',age=17 where s\_id=103;

The above command will update two columns of a record.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

#### 3) Delete command

Delete command is used to delete data from a table. Delete command can also be used with condition to delete a particular row. Following is its general syntax,

**DELETE** from *table-name*;

#### Example to Delete all Records from a Table

DELETE from Student;

The above command will delete all the records from **Student** table.

#### Example to Delete a particular Record from a Table

Consider the following **Student** table

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

DELETE from Student where s\_id=103;

The above command will delete the record where s\_id is 103 from **Student** table.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |

# TCL Command

### TCL command

Transaction Control Language(TCL) commands are used to manage transactions in database.These are used to manage the changes made by DML statements. It also allows statements to be grouped together into logical transactions.

#### Commit command

Commit command is used to permanently save any transaaction into database.

Following is Commit command's syntax,

***commit***;

#### Rollback command

This command restores the database to last commited state. It is also use with savepoint command to jump to a savepoint in a transaction.

Following is Rollback command's syntax,

**rollback** to *savepoint-name*;

#### Savepoint command

**savepoint** command is used to temporarily save a transaction so that you can rollback to that point whenever necessary.

Following is savepoint command's syntax,

**savepoint** *savepoint-name*;

#### Example of Savepoint and Rollback

Following is the **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |

Lets use some SQL queries on the above table and see the results.

INSERT into class values(5,'Rahul');

commit;

UPDATE class set name='abhijit' where id='5';

savepoint **A**;

INSERT into class values(6,'Chris');

savepoint **B**;

INSERT into class values(7,'Bravo');

savepoint **C**;

SELECT \* from class;

The resultant table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |
| 5 | abhijit |
| 6 | chris |
| 7 | bravo |

Now **rollback** to **savepoint B**

rollback to B;

SELECT \* from class;

The resultant table will look like

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |
| 5 | abhijit |
| 6 | chris |

Now **rollback** to **savepoint A**

rollback to A;

SELECT \* from class;

The result table will look like

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |
| 5 | abhijit |

# DCL Command

#### DCL command

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

* **System :** creating session, table etc are all types of system privilege.
* **Object :** any command or query to work on tables comes under object privilege.

DCL defines two commands,

* **Grant :** Gives user access privileges to database.
* **Revoke :** Take back permissions from user.

#### To Allow a User to create Session

**grant** create session to *username*;

#### To Allow a User to create Table

**grant** create table to *username*;

#### To provide User with some Space on Tablespace to store Table

**alter** user *username* quota unlimited on system;

#### To Grant all privilege to a User

**grant** sysdba to *username*

#### To Grant permission to Create any Table

**grant** *create* any table to *username*

#### To Grant permission to Drop any Table

**grant** *drop* any table to *username*

#### To take back Permissions

**revoke** create table from *username*

### WHERE clause

Where clause is used to specify condition while retriving data from table. *Where* clause is used mostly with *Select*, *Update* and *Delete* query. If condititon specified by *where* clause is true then only the result from table is returned.

#### Syntax for WHERE clause

*SELECT* column-name1,

column-name2,

column-name3,

column-nameN

from table-name **WHERE [condition]**;

#### Example using WHERE clause

Consider a **Student** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **s\_Name** | **age** | **address** |
| 101 | Adam | 15 | Noida |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Rohtak |
| 104 | Ankit | 22 | Panipat |

Now we will use a SELECT statement to display data of the table, based on a condition, which we will add to the SELECT query using WHERE clause.

SELECT s\_id,

s\_name,

age,

address

from Student **WHERE** s\_id=101;

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **s\_Name** | **age** | **address** |
| 101 | Adam | 15 | Noida |

### SELECT Query

Select query is used to retrieve data from a tables. It is the most used SQL query. We can retrieve complete tables, or partial by mentioning conditions using WHERE clause.

#### Syntax of SELECT Query

**SELECT** column-name1, column-name2, column-name3, column-nameN from *table-name*;

#### Example for SELECT Query

Conside the following **Student** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_id** | **S\_Name** | **age** | **address** |
| 101 | Adam | 15 | Noida |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Rohtak |
| 104 | Ankit | 22 | Panipat |

SELECT s\_id, s\_name, age from Student.

The above query will fetch information of s\_id, s\_name and age column from Student table

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |
| 104 | Ankit | 22 |

#### Example to Select all Records from Table

A special character **asterisk** \* is used to address all the data(belonging to all columns) in a query. *SELECT* statement uses \* character to retrieve all records from a table.

SELECT \* from student;

The above query will show all the records of Student table, that means it will show complete Student table as result.

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_id** | **S\_Name** | **age** | **address** |
| 101 | Adam | 15 | Noida |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Rohtak |
| 104 | Ankit | 22 | Panipat |

#### Example to Select particular Record based on Condition

SELECT \* from Student **WHERE** s\_name = 'Abhi';

|  |  |  |  |
| --- | --- | --- | --- |
| 103 | Abhi | 17 | Rohtak |

#### Example to Perform Simple Calculations using Select Query

Conside the following **Employee** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **Name** | **age** | **salary** |
| 101 | Adam | 26 | 5000 |
| 102 | Ricky | 42 | 8000 |
| 103 | Abhi | 22 | 10000 |
| 104 | Rohan | 35 | 5000 |

SELECT eid, name, salary+3000 from Employee;

The above command will display a new column in the result, showing 3000 added into existing salaries of the employees.

|  |  |  |
| --- | --- | --- |
| **eid** | **Name** | **salary+3000** |
| 101 | Adam | 8000 |
| 102 | Ricky | 11000 |
| 103 | Abhi | 13000 |
| 104 | Rohan | 8000 |

### Like clause

**Like** clause is used as condition in SQL query. **Like** clause compares data with an expression using wildcard operators. It is used to find similar data from the table.

#### Wildcard operators

There are two wildcard operators that are used in like clause.

* **Percent sign %** : represents zero, one or more than one character.
* **Underscore sign \_** : represents only one character.

#### Example of LIKE clause

Consider the following **Student** table.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

SELECT \* from Student where s\_name like 'A%';

The above query will return all records where **s\_name** starts with character 'A'.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

#### Example

SELECT \* from Student where s\_name like '\_d%';

The above query will return all records from **Student** table where **s\_name** contain 'd' as second character.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 101 | Adam | 15 |

#### Example

SELECT \* from Student where s\_name like '%x';

The above query will return all records from **Student** table where **s\_name** contain 'x' as last character.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 102 | Alex | 18 |

### Order By Clause

Order by clause is used with **Select** statement for arranging retrieved data in sorted order. The **Order by** clause by default sort data in ascending order. To sort data in descending order **DESC** keyword is used with **Order by** clause.

#### Syntax of Order By

*SELECT* column-list|\* from table-name **order by** *asc*|*desc*;

#### Example using Order by

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 |

SELECT \* from Emp **order by** salary;

The above query will return result in ascending order of the **salary**.

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

#### Example of Order by DESC

Consider the **Emp** table described above,

SELECT \* from Emp order by salary DESC;

The above query will return result in descending order of the **salary**.

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

### Group By Clause

Group by clause is used to group the results of a SELECT query based on one or more columns. It is also used with SQL functions to group the result from one or more tables.

Syntax for using Group by in a statement.

SELECT column\_name, function(column\_name)

FROM table\_name

WHERE condition

GROUP BY column\_name

#### Example of Group by in a Statement

Consider the following **Emp** table.

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

Here we want to find name and age of employees grouped by their salaries

SQL query for the above requirement will be,

SELECT name, age

from Emp **group by** salary

Result will be,

|  |  |
| --- | --- |
| **name** | **age** |
| Rohan | 34 |
| shane | 29 |
| anu | 22 |

#### Example of Group by in a Statement with WHERE clause

Consider the following **Emp** table

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

SQL query will be,

select name, salary

from Emp

where age > 25

**group by** salary

Result will be.

|  |  |
| --- | --- |
| **name** | **salary** |
| Rohan | 6000 |
| Shane | 8000 |
| Scott | 9000 |

You must remember that Group By clause will always come at the end, just like the Order by clause.

### HAVING Clause

having clause is used with SQL Queries to give more precise condition for a statement. It is used to mention condition in Group based SQL functions, just like WHERE clause.

Syntax for having will be,

select column\_name, function(column\_name)

FROM table\_name

WHERE column\_name condition

GROUP BY column\_name

**HAVING** function(column\_name) condition

#### Example of HAVING Statement

Consider the following **Sale** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **oid** | **order\_name** | **previous\_balance** | **customer** |
| 11 | ord1 | 2000 | Alex |
| 12 | ord2 | 1000 | Adam |
| 13 | ord3 | 2000 | Abhi |
| 14 | ord4 | 1000 | Adam |
| 15 | ord5 | 2000 | Alex |

Suppose we want to find the customer whose previous\_balance sum is more than 3000.

We will use the below SQL query,

SELECT \*

from sale group customer

having sum(previous\_balance) > 3000

Result will be,

|  |  |  |  |
| --- | --- | --- | --- |
| **oid** | **order\_name** | **previous\_balance** | **customer** |
| 11 | ord1 | 2000 | Alex |

### Distinct keyword

The **distinct** keyword is used with **Select** statement to retrieve unique values from the table. **Distinct** removes all the duplicate records while retrieving from database.

#### Syntax for DISTINCT Keyword

**SELECT** *distinct* column-name from *table-name*;

#### Example

Consider the following **Emp** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 10000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

select distinct salary from Emp;

The above query will return only the unique salary from **Emp** table

|  |
| --- |
| **salary** |
| 5000 |
| 8000 |
| 10000 |

### AND & OR operator

**AND** and **OR** operators are used with **Where** clause to make more precise conditions for fetching data from database by combining more than one condition together.

#### AND operator

AND operator is used to set multiple conditions with *Where* clause.

#### Example of AND

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

SELECT \* from Emp WHERE salary < 10000 **AND** age > 25

The above query will return records where salary is less than 10000 and age greater than 25.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 402 | Shane | 29 | 8000 |
| 405 | Tiger | 35 | 9000 |

#### OR operator

OR operator is also used to combine multiple conditions with *Where* clause. The only difference between AND and OR is their behaviour. When we use AND to combine two or more than two conditions, records satisfying all the condition will be in the result. But in case of OR, atleast one condition from the conditions specified must be satisfied by any record to be in the result.

#### Example of OR

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

SELECT \* from Emp WHERE salary > 10000 **OR** age > 25

The above query will return records where either salary is greater than 10000 or age greater than 25.

|  |  |  |  |
| --- | --- | --- | --- |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

# Advance SQL

### SQL Constraints

SQL Constraints are rules used to limit the type of data that can go into a table, to maintain the accuracy and integrity of the data inside table.

Constraints can be divided into following two types,

* **Column level constraints :** limits only column data
* **Table level constraints :** limits whole table data

Constraints are used to make sure that the integrity of data is maintained in the database. Following are the most used constraints that can be applied to a table.

* NOT NULL
* UNIQUE
* PRIMARY KEY
* FOREIGN KEY
* CHECK
* DEFAULT

#### NOT NULL Constraint

NOT NULL constraint restricts a column from having a NULL value. Once **NOT NULL** constraint is applied to a column, you cannot pass a null value to that column. It enforces a column to contain a proper value. One important point to note about NOT NULL constraint is that it cannot be defined at table level.

#### Example using NOT NULL constraint

CREATE table Student(s\_id int NOT NULL, Name varchar(60), Age int);

The above query will declare that the **s\_id** field of **Student** table will not take NULL value.

#### UNIQUE Constraint

UNIQUE constraint ensures that a field or column will only have unique values. A UNIQUE constraint field will not have duplicate data. UNIQUE constraint can be applied at column level or table level.

#### Example using UNIQUE constraint when creating a Table (Table Level)

CREATE table Student(s\_id int NOT NULL UNIQUE, Name varchar(60), Age int);

The above query will declare that the **s\_id** field of **Student** table will only have unique values and wont take NULL value.

#### Example using UNIQUE constraint after Table is created (Column Level)

ALTER table Student add UNIQUE(s\_id);

The above query specifies that **s\_id** field of **Student** table will only have unique value.

#### Primary Key Constraint

Primary key constraint uniquely identifies each record in a database. A Primary Key must contain unique value and it must not contain null value. Usually Primary Key is used to index the data inside the table.

#### Example using PRIMARY KEY constraint at Table Level

CREATE table Student (s\_id int **PRIMARY KEY**, Name varchar(60) NOT NULL, Age int);

The above command will creates a PRIMARY KEY on the s\_id.

#### Example using PRIMARY KEY constraint at Column Level

ALTER table Student add PRIMARY KEY (s\_id);

The above command will creates a PRIMARY KEY on the s\_id.

#### Foreign Key Constraint

FOREIGN KEY is used to relate two tables. FOREIGN KEY constraint is also used to restrict actions that would destroy links between tables. To understand FOREIGN KEY, let's see it using two table.

**Customer\_Detail Table :**

|  |  |  |
| --- | --- | --- |
| **c\_id** | **Customer\_Name** | **address** |
| 101 | Adam | Noida |
| 102 | Alex | Delhi |
| 103 | Stuart | Rohtak |

**Order\_Detail Table :**

|  |  |  |
| --- | --- | --- |
| **Order\_id** | **Order\_Name** | **c\_id** |
| 10 | Order1 | 101 |
| 11 | Order2 | 103 |
| 12 | Order3 | 102 |

In **Customer\_Detail** table, c\_id is the primary key which is set as foreign key in **Order\_Detail** table. The value that is entered in c\_id which is set as foreign key in **Order\_Detail** table must be present in **Customer\_Detail** table where it is set as primary key. This prevents invalid data to be inserted into c\_id column of **Order\_Detail** table.

#### Example using FOREIGN KEY constraint at Table Level

CREATE table Order\_Detail(order\_id int PRIMARY KEY,

order\_name varchar(60) NOT NULL,

*c\_id int* **FOREIGN KEY** REFERENCES **Customer\_Detail**(*c\_id*));

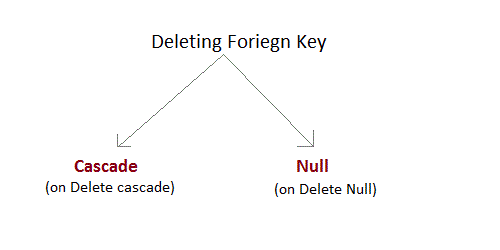
In this query, c\_id in table Order\_Detail is made as foriegn key, which is a reference of c\_id column of Customer\_Detail.

#### Example using FOREIGN KEY constraint at Column Level

ALTER table Order\_Detail add **FOREIGN KEY** (c\_id) REFERENCES Customer\_Detail(c\_id);

#### Behaviour of Foriegn Key Column on Delete

There are two ways to maintin the integrity of data in Child table, when a particular record is deleted in main table. When two tables are connected with Foriegn key, and certain data in the main table is deleted, for which record exit in child table too, then we must have some mechanism to save the integrity of data in child table.



* **On Delete Cascade :** This will remove the record from child table, if that value of foriegn key is deleted from the main table.
* **On Delete Null :** This will set all the values in that record of child table as NULL, for which the value of foriegn key is deleted from the main table.
* If we don't use any of the above, then we cannot delete data from the main table for which data in child table exists. We will get an error if we try to do so.

ERROR : Record in child table exist

#### CHECK Constraint

CHECK constraint is used to restrict the value of a column between a range. It performs check on the values, before storing them into the database. Its like condition checking before saving data into a column.

#### Example using CHECK constraint at Table Level

create table Student(s\_id int NOT NULL **CHECK(s\_id > 0)**,

Name varchar(60) NOT NULL,

Age int);

The above query will restrict the s\_id value to be greater than zero.

#### Example using CHECK constraint at Column Level

ALTER table Student add CHECK(s\_id > 0);

### 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 catagories,

* Aggregrate Functions
* Scalar Functions

#### Aggregrate Functions

These functions return a single value after calculating from a group of values.Following are some frequently used Aggregrate functions.

#### 1) AVG()

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

Its general Syntax is,

SELECT **AVG**(column\_name) from *table\_name*

#### Example using AVG()

Consider 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 of salary will be,

SELECT **avg(salary)** from Emp;

Result of the above query will be,

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

#### 2) COUNT()

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*

#### Example using COUNT()

Consider 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 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 |

#### 3) FIRST()

First function returns first value of a selected column

Syntax for FIRST function is,

SELECT **FIRST**(column\_name) from *table-name*

#### Example of FIRST()

Consider 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

SELECT FIRST(salary) from Emp;

Result will be,

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

#### 4) LAST()

LAST return the return last value from selected column

Syntax of LAST function is,

SELECT **LAST**(column\_name) from *table-name*

#### Example of LAST()

Consider 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 |

#### 5) MAX()

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

Syntax of MAX function is,

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

#### Example of MAX()

Consider 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 Maximum salary is,

SELECT MAX(salary) from emp;

Result of the above query will be,

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

#### 6) MIN()

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

Syntax for MIN function is,

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

#### Example of MIN()

Consider 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 |

#### 7) SUM()

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

Syntax for SUM is,

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

#### Example of SUM()

Consider 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 soe frequently used Scalar Functions.

#### 1) UCASE()

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

Syntax of UCASE,

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

#### Example of UCASE()

Consider 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 |

#### 2) LCASE()

LCASE function is used to convert value of string column to Lowecase character.

Syntax for LCASE is,

SELECT **LCASE**(column\_name) from *table-name*

#### Example of LCASE()

Consider 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 |

#### 3) MID()

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*

#### Example of MID()

Consider 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 |

#### 4) ROUND()

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*

#### Example of ROUND()

Consider 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 |

### Join in SQL

SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data. SQL Join is used for combining column from two or more tables by using values common to both tables. **Join** Keyword is used in SQL queries for joining two or more tables. Minimum required condition for joining table, is **(n-1)** where **n**, is number of tables. A table can also join to itself known as, **Self Join**.

#### Types of Join

The following are the types of JOIN that we can use in SQL.

* Inner
* Outer
* Left
* Right

#### Cross JOIN or Cartesian Product

This type of JOIN returns the cartesian product of rows from the tables in Join. It will return a table which consists of records which combines each row from the first table with each row of the second table.

Cross JOIN Syntax is,

SELECT column-name-list

from *table-name1*

**CROSS JOIN**

*table-name2*;

#### Example of Cross JOIN

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Cross** JOIN query will be,

SELECT \*

from class,

cross JOIN class\_info;

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 1 | DELHI |
| 4 | alex | 1 | DELHI |
| 1 | abhi | 2 | MUMBAI |
| 2 | adam | 2 | MUMBAI |
| 4 | alex | 2 | MUMBAI |
| 1 | abhi | 3 | CHENNAI |
| 2 | adam | 3 | CHENNAI |
| 4 | alex | 3 | CHENNAI |

#### INNER Join or EQUI Join

This is a simple JOIN in which the result is based on matched data as per the equality condition specified in the query.

Inner Join Syntax is,

SELECT column-name-list

from *table-name1*

**INNER JOIN**

*table-name2*

WHERE table-name1.column-name = table-name2.column-name;

#### Example of Inner JOIN

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Inner** JOIN query will be,

SELECT \* from class, class\_info where class.id = class\_info.id;

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |

#### Natural JOIN

Natural Join is a type of Inner join which is based on column having same name and same datatype present in both the tables to be joined.

Natural Join Syntax is,

SELECT \*

from *table-name1*

**NATURAL JOIN**

*table-name2*;

#### Example of Natural JOIN

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Natural join query will be,**

SELECT \* from class NATURAL JOIN class\_info;

The result table will look like,

|  |  |  |
| --- | --- | --- |
| **ID** | **NAME** | **Address** |
| 1 | abhi | DELHI |
| 2 | adam | MUMBAI |
| 3 | alex | CHENNAI |

In the above example, both the tables being joined have ID column(same name and same datatype), hence the records for which value of ID matches in both the tables will be the result of Natural Join of these two tables.

#### Outer JOIN

Outer Join is based on both matched and unmatched data. Outer Joins subdivide further into,

* Left Outer Join
* Right Outer Join
* Full Outer Join

#### Left Outer Join

The left outer join returns a result table with the **matched data** of two tables then remaining rows of the **left** table and null for the **right** table's column.

Left Outer Join syntax is,

SELECT column-name-list

from *table-name1*

**LEFT OUTER JOIN**

*table-name2*

on table-name1.column-name = table-name2.column-name;

Left outer Join Syntax for **Oracle** is,

select column-name-list

from *table-name1*,

*table-name2*

on table-name1.column-name = table-name2.column-name(**+**);

#### Example of Left Outer Join

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Left Outer Join** query will be,

SELECT \* FROM class LEFT OUTER JOIN class\_info ON (class.id=class\_info.id);

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | anu | null | null |
| 5 | ashish | null | null |

#### Right Outer Join

The right outer join returns a result table with the **matched data** of two tables then remaining rows of the **right table** and null for the **left** table's columns.

Right Outer Join Syntax is,

select column-name-list

from *table-name1*

**RIGHT OUTER JOIN**

*table-name2*

on table-name1.column-name = table-name2.column-name;

Right outer Join Syntax for **Oracle** is,

select column-name-list

from *table-name1*,

*table-name2*

on table-name1.column-name(**+**) = table-name2.column-name;

#### Example of Right Outer Join

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Right Outer Join** query will be,

SELECT \* FROM class RIGHT OUTER JOIN class\_info on (class.id=class\_info.id);

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |

#### Full Outer Join

The full outer join returns a result table with the **matched data** of two table then remaining rows of both **left** table and then the **right** table.

Full Outer Join Syntax is,

select column-name-list

from *table-name1*

**FULL OUTER JOIN**

*table-name2*

on table-name1.column-name = table-name2.column-name;

#### Example of Full outer join is,

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Full Outer Join** query will be like,

SELECT \* FROM class FULL OUTER JOIN class\_info on (class.id=class\_info.id);

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | anu | null | null |
| 5 | ashish | null | null |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |

### SQL Alias

Alias is used to give an alias name to a table or a column. This is quite useful in case of large or complex queries. Alias is mainly used for giving a short alias name for a column or a table with complex names.

Syntax of Alias for table names,

**SELECT** column-name

from *table-name*

as **alias-name**

Following is an Example using Alias,

SELECT \* from Employee\_detail as **ed**;

Alias syntax for columns will be like,

**SELECT**

*column-name* as **alias-name**

from

*table-name*

Example using alias for columns,

SELECT customer\_id as **cid** from Emp;

#### Example of Alias in SQL Query

Consider the following two tables,

The **class** table,

|  |  |
| --- | --- |
| **ID** | **Name** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

Below is the Query to fetch data from both the tables using SQL Alias,

SELECT C.id, C.Name, Ci.Address from Class as C, Class\_info as Ci where C.id=Ci.id;

Result table look like,

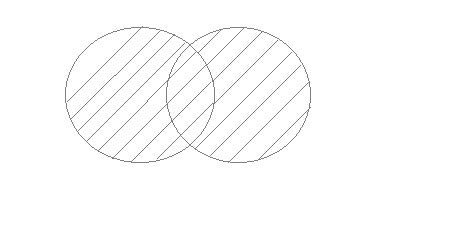
|  |  |  |
| --- | --- | --- |
| **ID** | **Name** | **Address** |
| 1 | abhi | DELHI |
| 2 | adam | MUMBAI |
| 3 | alex | CHENNAI |

### Set Operation in SQL

SQL supports few Set operations to be performed on table data. These are used to get meaningful results from data, under different special conditions.

#### Union

UNION is used to combine the results of two or more Select statements. However it will eliminate duplicate rows from its result set. In case of union, number of columns and datatype must be same in both the tables.



#### Example of UNION

The **First** table,

|  |  |
| --- | --- |
| **ID** | **Name** |
| 1 | abhi |
| 2 | adam |

The **Second** table,

|  |  |
| --- | --- |
| **ID** | **Name** |
| 2 | adam |
| 3 | Chester |

Union SQL query will be,

select \* from First

**UNION**

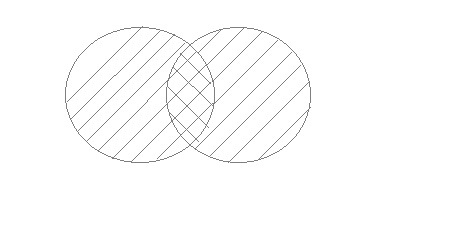
select \* from second

The result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | Chester |

#### Union All

This operation is similar to Union. But it also shows the duplicate rows.



#### Example of Union All

The **First** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |

The **Second** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | adam |
| 3 | Chester |

Union All query will be like,

select \* from First

**UNION ALL**

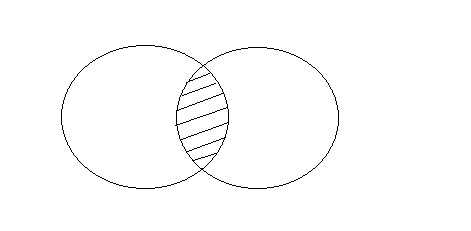
select \* from second

The result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 2 | adam |
| 3 | Chester |

#### Intersect

Intersect operation is used to combine two SELECT statements, but it only retuns the records which are common from both SELECT statements. In case of **Intersect** the number of columns and datatype must be same. MySQL does not support INTERSECT operator.



#### Example of Intersect

The **First** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |

The **Second** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | adam |
| 3 | Chester |

Intersect query will be,

select \* from First

**INTERSECT**

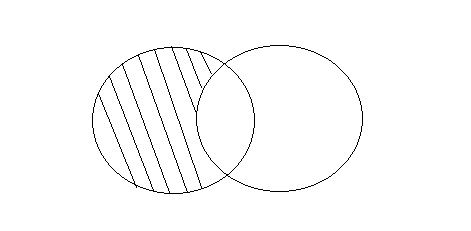
select \* from second

The result table will look like

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | adam |

#### Minus

Minus operation combines result of two Select statements and return only those result which belongs to first set of result. MySQL does not support INTERSECT operator.



#### Example of Minus

The **First** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |

The **Second** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | adam |
| 3 | Chester |

Minus query will be,

select \* from First

**MINUS**

select \* from second

The result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |

### SQL Sequence

Sequence is a feature supported by some database systems to produce unique values on demand. Some DBMS like **MySQL** supports AUTO\_INCREMENT in place of Sequence. AUTO\_INCREMENT is applied on columns, it automatically increments the column value by 1 each time a new record is entered into the table. Sequence is also some what similar to AUTO\_INCREMENT but its has some extra features.

#### Creating Sequence

Syntax to create sequences is,

CREATE **Sequence** *sequence-name*

**start** with *initial-value*

**increment** by *increment-value*

**maxvalue** *maximum-value*

cycle|nocycle

**initial-value** specifies the starting value of the Sequence, **increment-value** is the value by which sequence will be incremented and **maxvalue** specifies the maximum value until which sequence will increment itself. **cycle** specifies that if the maximum value exceeds the set limit, sequence will restart its cycle from the begining. **No cycle** specifies that if sequence exceeds **maxvalue** an error will be thrown.

#### Example to create Sequence

The sequence query is following

CREATE **Sequence** seq\_1

start with 1

increment by 1

maxvalue 999

cycle ;

#### Example to use Sequence

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |

The sql query will be,

INSERT into class value(**seq\_1.nextval**,'anu');

Result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |
| 1 | anu |

Once you use nextval the sequence will increment even if you don't Insert any record into the table.

### SQL View

A view in SQL is a logical subset of data from one or more tables. View is used to restrict data access.

Syntax for creating a View,

CREATE or REPLACE **view** *view\_name* AS

SELECT *column\_name*(s)

FROM *table\_name*

WHERE *condition*

#### Example of Creating a View

Consider following **Sale** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **oid** | **order\_name** | **previous\_balance** | **customer** |
| 11 | ord1 | 2000 | Alex |
| 12 | ord2 | 1000 | Adam |
| 13 | ord3 | 2000 | Abhi |
| 14 | ord4 | 1000 | Adam |
| 15 | ord5 | 2000 | Alex |

SQL Query to Create View

CREATE or REPLACE **view** sale\_view as select \* from Sale where customer = 'Alex';

The data fetched from select statement will be stored in another object called **sale\_view**. We can use create seperately and replace too but using both together works better.

#### Example of Displaying a View

Syntax of displaying a view is similar to fetching data from table using Select statement.

SELECT \* from **sale\_view**;

#### Force View Creation

force keyword is used while creating a view. This keyword force to create View even if the table does not exist. After creating a force View if we create the base table and enter values in it, the view will be automatically updated.

Syntax for forced View is,

CREATE or REPLACE *force* **view** *view\_name* AS

SELECT *column\_name*(s)

FROM *table\_name*

WHERE *condition*

#### Update a View

Update command for view is same as for tables.

Syntax to Update a View is,

UPDATE **view-name**

set value

WHERE condition;

If we update a view it also updates base table data automatically.

#### Read-Only View

We can create a view with read-only option to restrict access to the view.

Syntax to create a view with Read-Only Access

CREATE or REPLACE *force* **view** *view\_name* AS

SELECT *column\_name*(s)

FROM *table\_name*

WHERE *condition* with **read-only**

The above syntax will create view for read-only purpose, we cannot Update or Insert data into read-only view. It will throw an error.

#### Types of View

There are two types of view,

* Simple View
* Complex View

|  |  |
| --- | --- |
| **Simple View** | **Complex View** |
| Created from one table | Created from one or more table |
| Does not contain functions | Contain functions |
| Does not contain groups of data | Contains groups of data |