Syllabus: Entity Relationship Model: Introduction, Representation of entities, attributes, entity set, relationship, relationship set, constraints, sub classes, super class, inheritance, specialization, generalization using ER Diagrams. SQL: Creating tables with relationship, implementation of key and integrity constraints, nested queries, sub queries, grouping, aggregation, ordering, implementation of different types of joins, view (updatable and non-updatable), relational set operations.

Entity Relation Model

The Entity Relationship (ER) data model allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial data base design. Within the larger context of the overall design process, the ER model is used in a phase called. **"Conceptual database design"**.

ER Modeling helps you to analyze data requirements systematically to produce a well-designed database. So, it is considered a best practice to complete ER modeling before implementing your database.

ER diagrams are a visual tool which is helpful to represent the ER model. It was proposed by **Peter Chen** in 1971 to create a uniform convention which can be used for relational database and network.

Database design and ER Diagrams:

The database design process can be divided into six steps. The ER model is most relevant to the first three steps.

1. Requirement Analysis:

The very first step in designing a database application is to understand what data is to be stored in the database, what application must be built in top of it, and what operations are most frequent and subject to performance requirements. In other words, we must find out what the users want from the database.

2. Conceptual database Design:

The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints known to hold over this data. The ER model is one of several high level or semantic, data models used in database design.

3. Logical Database Design:

We must choose a database to convert the conceptual database design into a database schema in the data model of the chosen DBMS. Normally we will consider the Relational DBMS and therefore, the task in the logical design step is to convert an ER schema into a relational database schema.

Beyond ER Design

4. Schema Refinement:

This step is to analyze the collection of relations in our relational database schema to identify potential problems, and refine it.

5. Physical Database Design:

This step may simply involve building indexes on some table and clustering some tables or it may involve substantial redesign of parts of database schema obtained from the earlier steps.

6. Application and security Design:

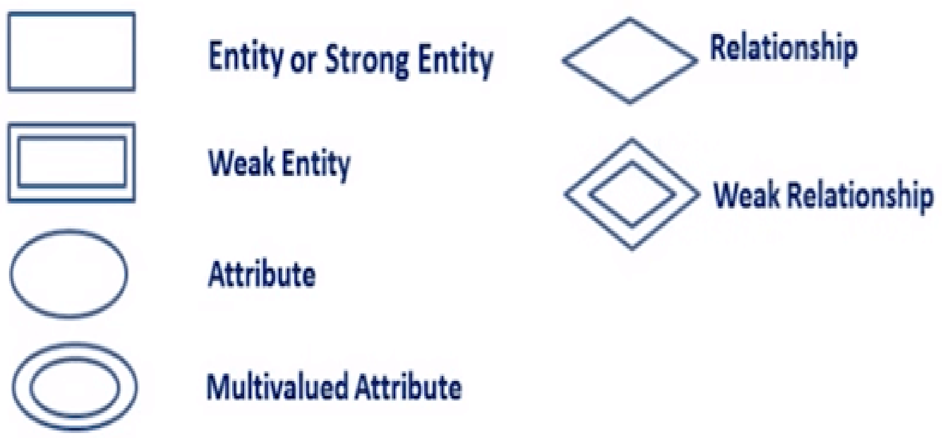
Any software project that involves a DBMS must consider aspects of the application that go beyond the database itself. We must describe the role of each entity (users, user group, departments) in every process that is reflected in some application tasks, as part of a complete workflow for the task. A DBMS Provides several mechanisms to assist in this step.

ER Diagrams Symbols & Notations

Entity Relationship Diagram Symbols & Notations mainly contains three basic symbols which are **rectangle**, **oval** and **diamond** to represent relationships between elements, entities, and attributes. There are some sub-elements which are based on main elements in ERD Diagram. ER Diagram is a visual representation of data that describes how data is related to each other using different ERD Symbols and Notations.

Following are the main components and its symbols in ER Diagrams:

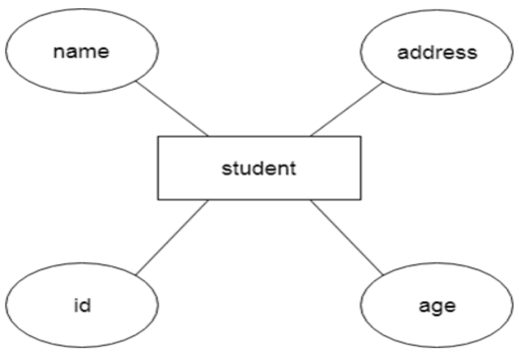
- **Rectangles:** This Entity Relationship Diagram symbol represents entity types
- **Ellipses:** Symbol represent attributes
- **Diamonds:** This symbol represents relationship types
- **Lines:** It links attributes to entity types and entity types with other relationship types
- **Primary key:** attributes are underlined.
- **Double Ellipses:** Represent multi-valued attributes.



ER Diagram Symbols

ER Diagram Examples

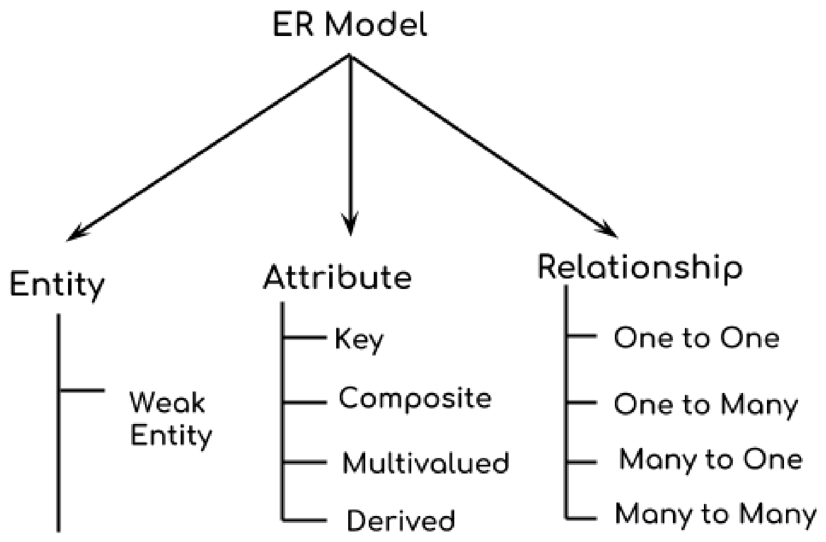
Suppose we design a school database. In this database, the student will be an entity with attributes like address, name, id, age, etc. The address can be another entity with attributes like city, street name, pin code, etc. and there will be a relationship between them.



Components of the ER Diagram

This model is based on three basic concepts:

- **Entities**
- **Attributes**
- **Relationships**



Representation of entities

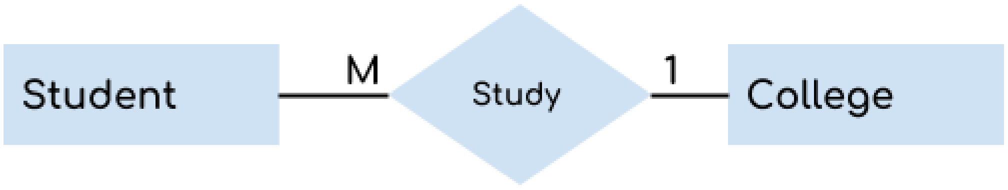
An entity can be place, person, object, event, or a concept, which stores data in the database. The characteristics of entities must have an attribute, and a unique key. Every entity is made up of some 'attributes' which represent that entity.

An entity is an object or component of data. An entity is represented as **rectangle** in an ER diagram.

Examples of entities:

- **Person:** Employee, Student, Patient.
- **Place:** Store, Building.
- **Object:** Machine, product, and Car.
- **Event:** Sale, Registration, Renewal.
- **Concept:** Account, Course.

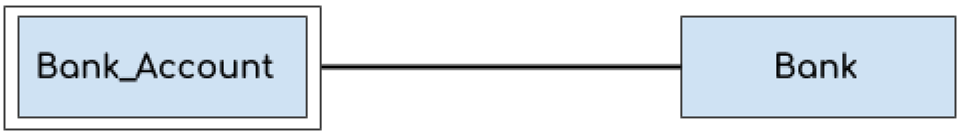
For example: In the following ER diagram we have two entities Student and College, and these two entities have many to one relationship as many student's study in a single college.



Weak Entity

An entity that cannot be uniquely identified by its own attributes and relies on the relationship with other entity is called weak entity. The weak entity is represented by a **double rectangle**.

For example – a bank account cannot be uniquely identified without knowing the bank to which the account belongs, so bank account is a weak entity.



Attributes

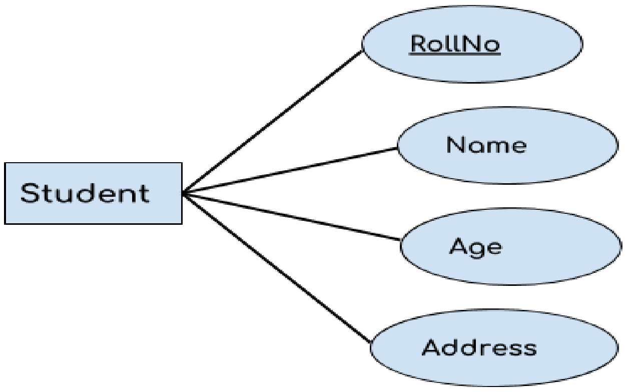
An attribute describes the property of an entity. For example, Roll_No, Name, DOB, Age, Address, Mobile_No are the attributes which defines entity type Student. In ER diagram, attribute is represented by an **oval**.

There are four types of attributes:

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

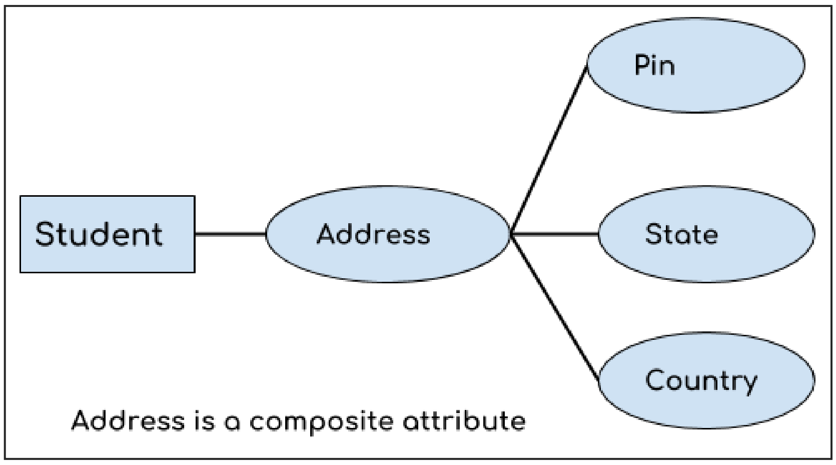
1. Key attribute

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



2. Composite attribute

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

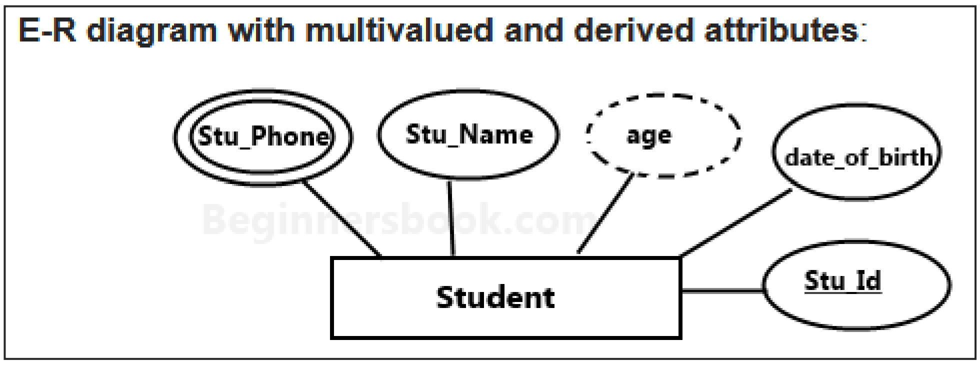


3. Multivalued attribute

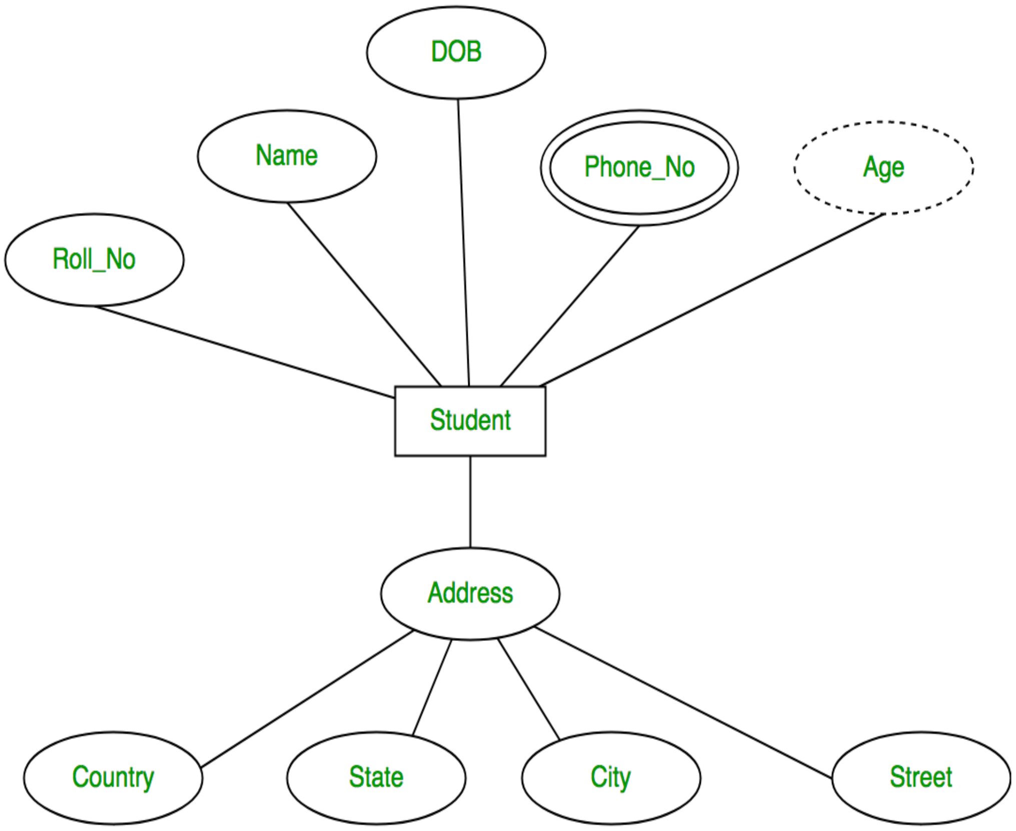
An attribute that can hold multiple values is known as multivalued attribute. It is represented with **double ovals** in an ER Diagram. For example – A person can have more than one phone numbers so the phone number attribute is multivalued.

4. Derived attribute:

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



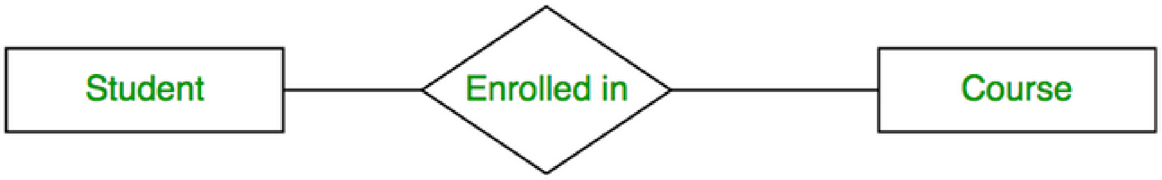
The complete entity type **Student** with its attributes can be represented as:



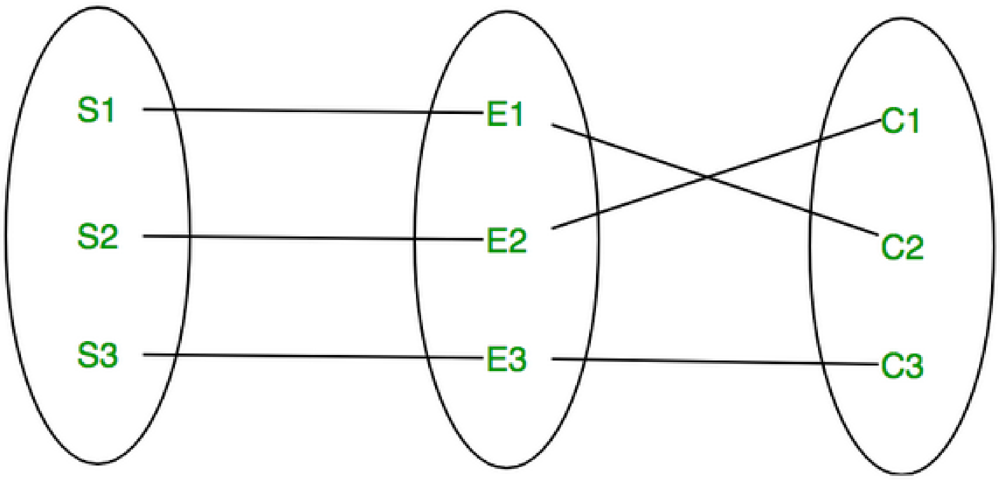
Relationship Type and Relationship Set

A relationship type represents the **association between entity types**. For example, 'Enrolled in' is a relationship type that exists between entity type Student and Course.

A relationship is represented by diamond shape and connecting the entities with lines in ER diagram, Name of the relationship is written inside the diamond-box, it shows the relationship among entities.



A set of relationships of same type is known as **relationship set**. The following relationship set depicts S1 is enrolled in C2, S2 is enrolled in C1 and S3 is enrolled in C3.

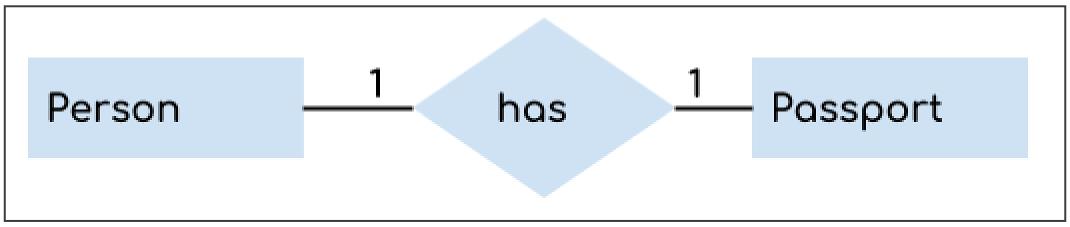


There are four types of relationships:

1. **One to One**
2. **One to Many**
3. **Many to One**
4. **Many to Many**

1. One to One Relationship

When a single instance of an entity is associated with a single instance of another entity then it is called one to one relationship. ***For example***, if there are two entities 'Person' (Id, Name, Age, Address) and 'Passport' (Passport_id, Passport_no). So, each person can have only one passport and each passport belongs to only one person.

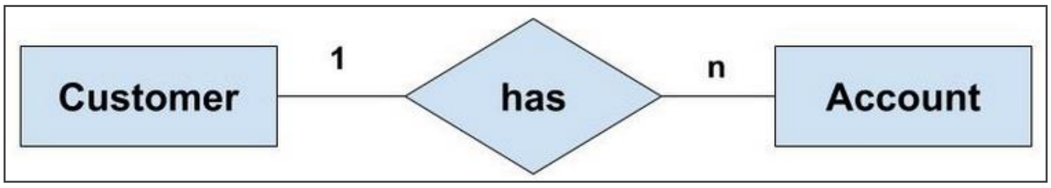


In the above example, we can easily store the passport id in the 'Person' table only. But we make another table for the 'Passport' because Passport number may be sensitive data and it should be hidden from certain users. So, by making a separate table we provide extra security that only certain database users can see it.

2. One to Many Relationship

When a single instance of an entity is associated with more than one instances of another entity then it is called one to many relationship.

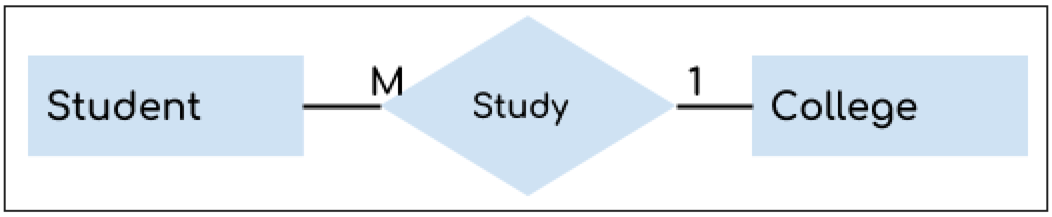
For example, if there are two entity type 'Customer' and 'Account' then each 'Customer' can have more than one 'Account' but each 'Account' is held by only one 'Customer'.



In this example, we can say that each Customer is associated with many Account. So, it is a one-to-many relationship. But if we see it the other way i.e., many Account is associated with one Customer then we can say that it is a many-to-one relationship.

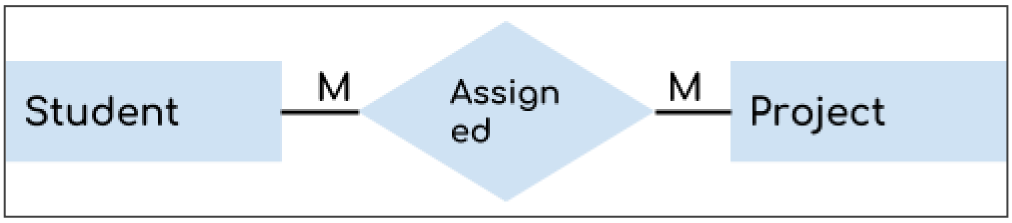
3. Many to One Relationship: When more than one instances of an entity is associated with a single instance of another entity then it is called many to one

relationship. For example – many students can study in a single college, but a student cannot study in many colleges at the same time.



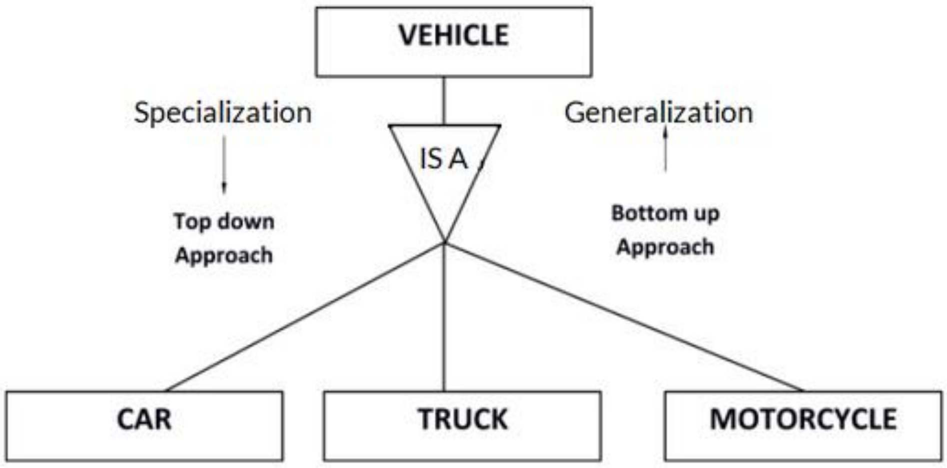
4. Many to Many Relationship

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



Sub classes, Super classes, Inheritance

A subclass is a class derived from the superclass. It inherits the properties of the superclass and also contains attributes of its own.



In the above Example Car, Truck and Motorcycle are all subclasses of the superclass Vehicle. They all inherit common attributes from vehicle such as speed, color etc. while they have different attributes also i.e., Number of wheels in Car is 4 while in Motorcycle is 2.

Super classes

A superclass is the class from which many subclasses can be created. The subclasses inherit the characteristics of a superclass. The superclass is also known as the parent class or base class.

In the above example, Vehicle is the Superclass and its subclasses are Car, Truck and Motorcycle.

Inheritance

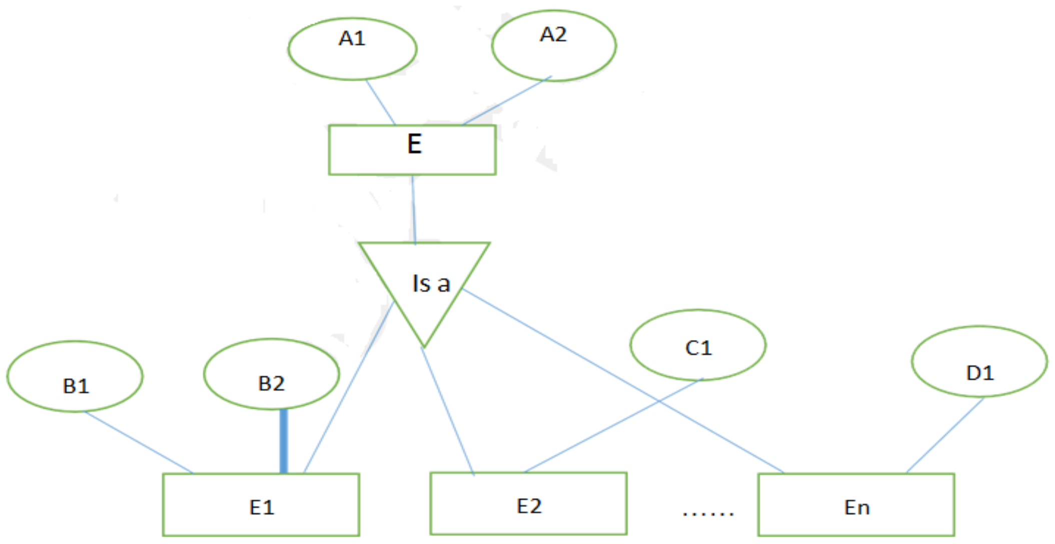
Inheritance is basically the process of basing a class on another class i.e., to build a class on an existing class. The new class contains all the features and functionalities of the old class in addition to its own.

The class which is newly created is known as the subclass or child class and the original class is the parent class or the superclass.

Specialization

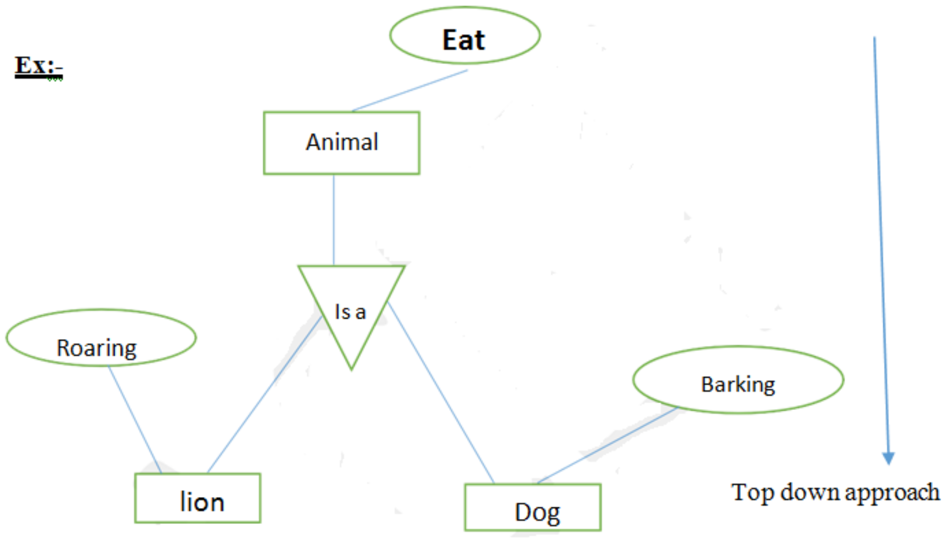
Specialization is the process of designating the sub-entities of an entity set depending upon the commonality of attributes is called specialization. An entity set **E** may include some sub-groups of entities say (**E1, E2,………. , En**) such that each of these may have some distinct or specific attributes that the other sub-groups or sub-entities. There will be some attributes common to all sub groups.

E-R model:-



- In the above example entity set **E** has been specialized into sub entity sets designated as **E1, E2,…….., En**.
- **E** is called **"super class"** or **"higher level entity set"** and the entity sets **E1, E2, --- En** are called as **"sub class"** or "low level entity sets" of **E**. the common attribute of all sub entity sets are represented with super entity sets.
- The specific attributes of each sub entity set are represented with the sub entity set.
- The relationship of higher level entity set with the lower level entity set is called **'is a'** relationship. i.e. **E1** is a sub group or sub entity of **E**. etc……

Ex:-

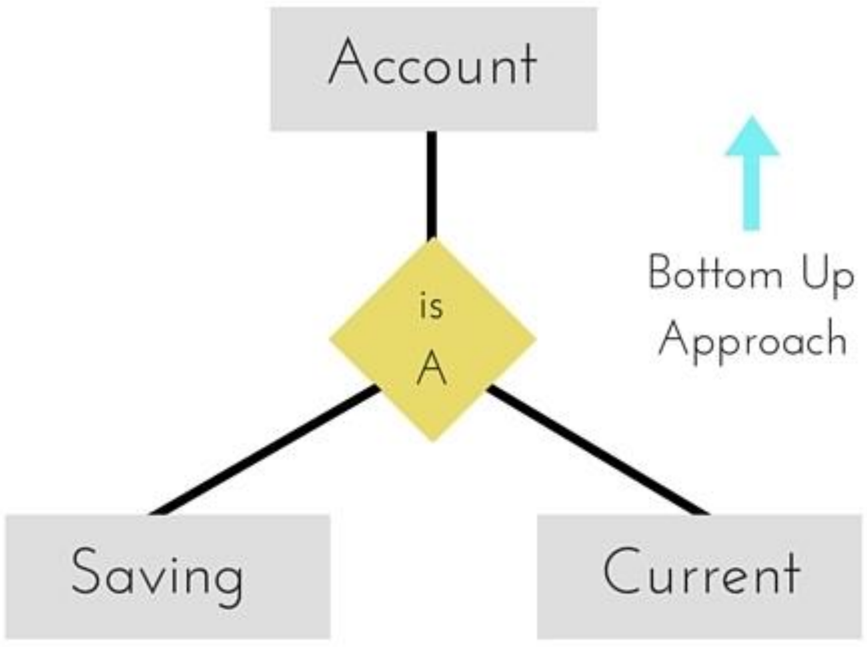


'Eat' is common to all animals but **'roaring'** is a specific to lion and **'barking'** is specific to dog.

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 entities to make further higher level entity.

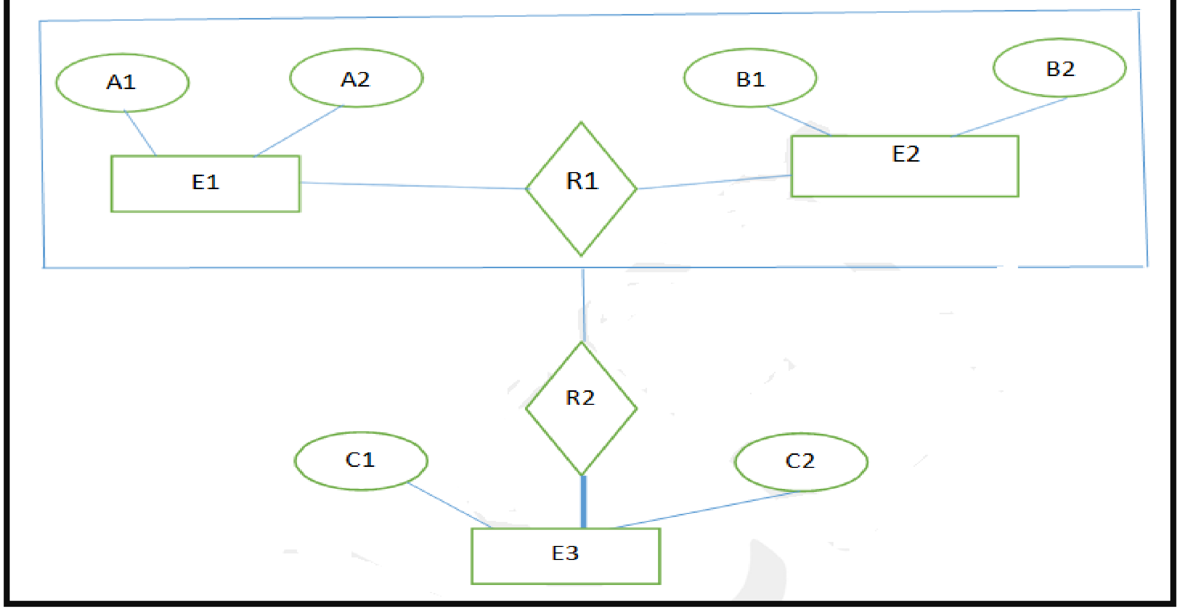
It's more like Superclass and Subclass system, but the only difference is the approach, which is bottom-up. Hence, entities are combined to form a more generalised entity, in other words, sub-classes are combined to form a super-class.



Aggregation

The limitation of **E-R** model is that, it fails to express relationship among the relationships sets aggregation provides solution for this. In aggregation relationship sets are treated as higher level entity sets which can participate in relationship sets with other entity sets.

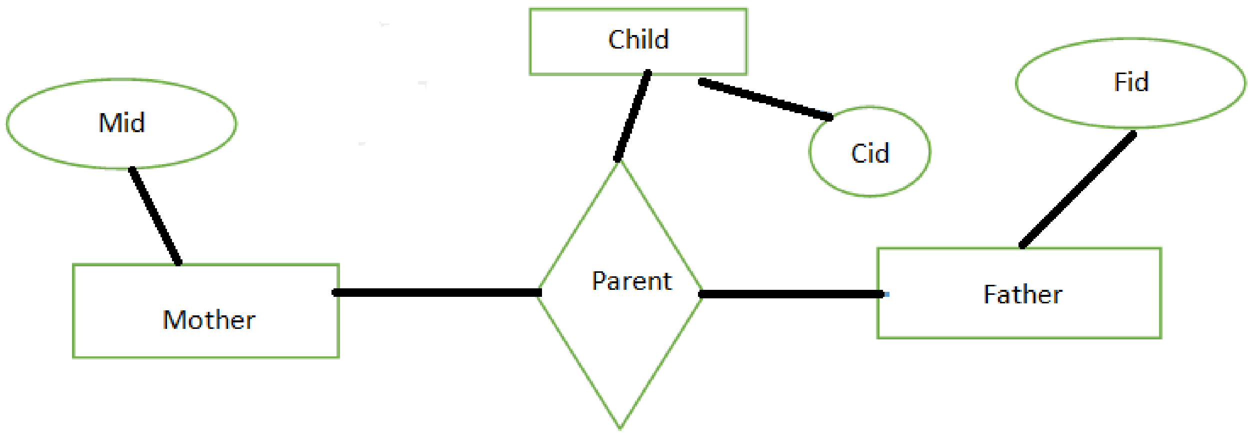
Ex:-



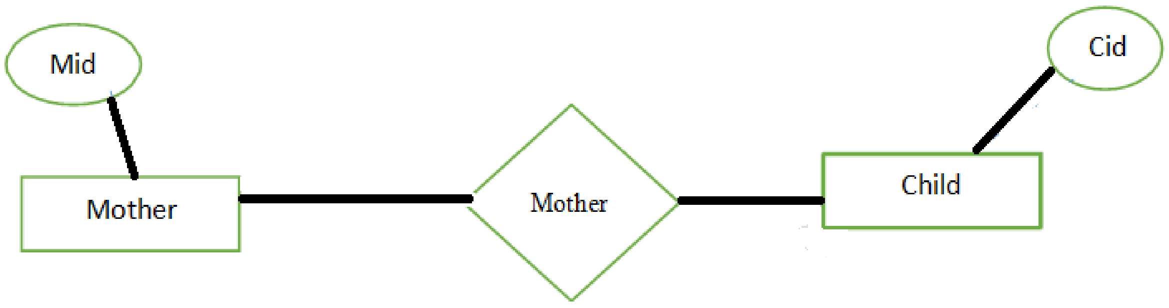
- The relationship **R1** between entity sets **E1** and **E2** has been aggregated as higher-level entity asset "**E**". This higher-level entity set will participating in a relationship set **R2** with an entity set **E3**.
- So, through aggregation we are able to form a relationship between two relationship sets.

Ternary relationship:

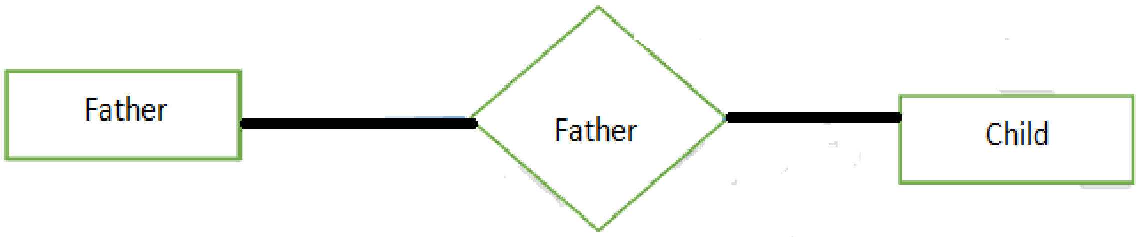
Relationships in database are often binary. But there may be some relationships which are non-binary.



- The above diagram is an example of ternary relationship. The relationship set is a parent is formed by the participation of three entity sets mother, father and child.
- We can divide the above ternary relationship into two binary relationships as shown below.



Mother is the relationship between child and mother entity sets.



Father is the relationship between father and child entity sets.

Using two relationships mother and father provides us a record of a child's mother even if we are not aware of the father's identity, provides us a record of a child's father, even if we are not aware of the mother identity. A null value would be required if the ternary relationship set parent is used in two cases. Using of binary relationship preferable in this case.

Converting of E-R diagrams into relational

An E-R diagram can be reduced to a set of tables while reducing

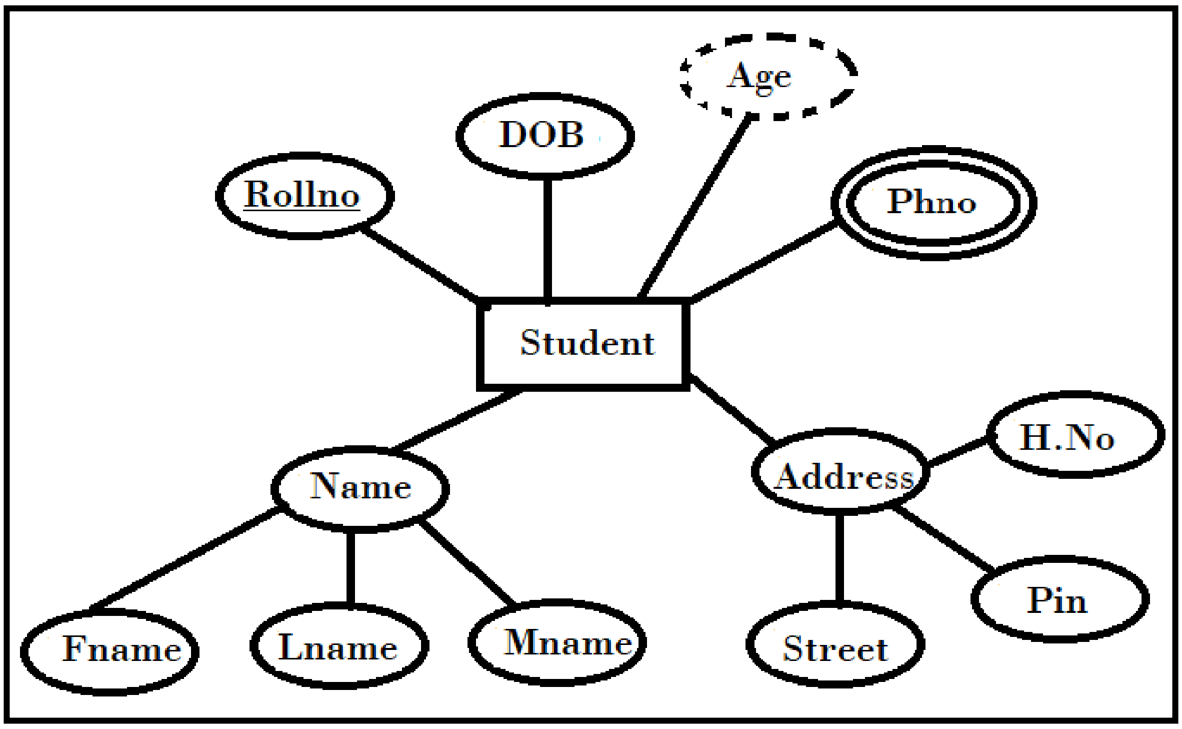
Points to be remember: -

- There will be a separate column for each simple and single valued attributes of given entity set.
- There will be a separate column for each sub-attribute and there is no column for composite attribute.
- There is no separate column for derived attribute
- Each multi valued attribute will be represented in a separate table which will have a column for primary key attribute and multi valued attribute.
- There is a separate table for each entity set and relationship set.

Tabular Representation of Strong Entity Set

Entity in ER Model is changed into tables, or we can say for every Entity in ER model, a table is created in Relational Model.

And the **attributes** of the Entity gets converted to columns of the table. And the primary key specified for the entity in the ER model, will become the primary key for the table in relational model.



A table with name **Student** will be created in relational model, which will have **8** columns Rollno, DOB, Fname, Lname, Mname, Hno, Street, Pin and Rollno will be the primary key for this table.

Rollno	DOB	Fname	Lname	Mname	Hno	Street	Pin

Student

Another Table with a name **Student_Phno** will be created in relational model, which have **2** columns Rollno, Phno.

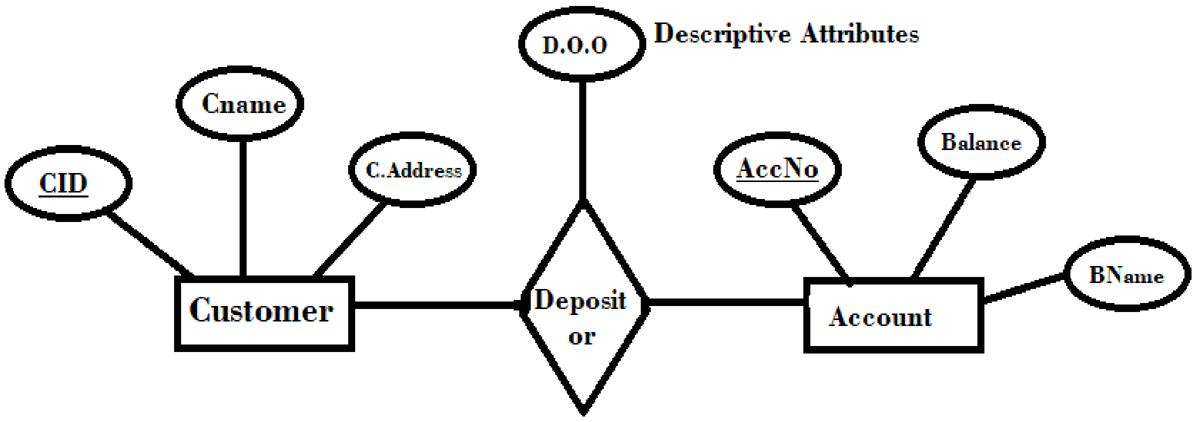
Rollno	Phno

Student_Phno

Tabular Representation of Relationship Set

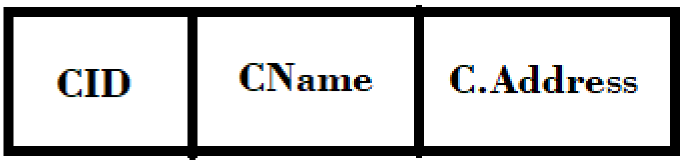
In ER diagram, we use diamond/rhombus to represent a relationship between two entities. In Relational model we create a relationship table for ER Model relationships too.

In the ER diagram below, we have two entities **Customer** and **Account** with a relationship between them.



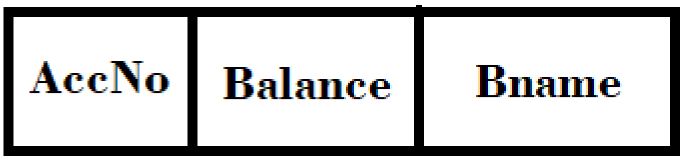
There are three tables for Customer, Account and Depositor.

As discussed above, entity gets mapped to table, hence we will create table for **Customer** and a table for **Account** with all the attributes converted into columns.



CID	CName	C.Address

Customer



AccNo	Balance	Bname

Account

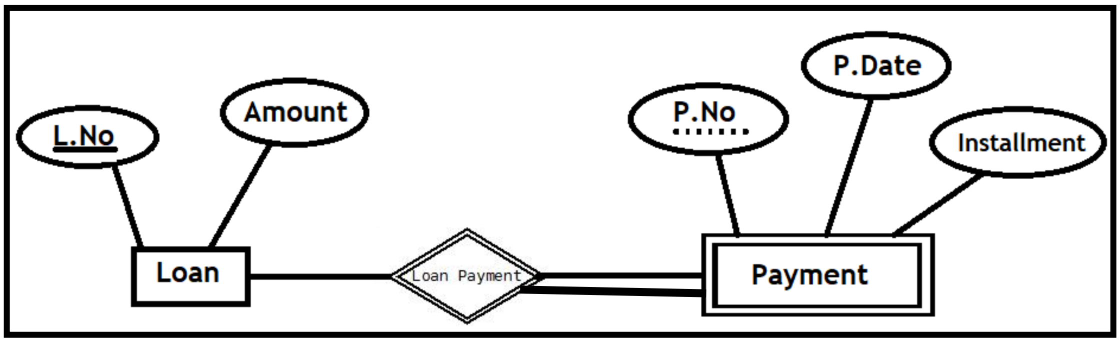
Now, an additional table will be created for the relationship, for example **Depositor**. This table will hold the primary key for both Customer and Account, in a tuple to describe the relationship, which Customer Depositor which Account. If there are additional attributes related to this relationship, then they become the columns for this table, like **D.O.O**.

CID	AccNo	D.O.O

Depositor

Tabular Representation of Weak Entity Set

In the below ER Diagram, **'Payment'** is the weak entity. 'Loan Payment' is the identifying relationship and 'Payment Number' is the partial key. Primary Key of the Loan along with the partial key would be used to identify the records.



There are three tables for Loan, Payment and Loan_Payment.

L.No	Amount

Loan

L.No	P.No	P.Date	Installment

Payment

L.No	P.No

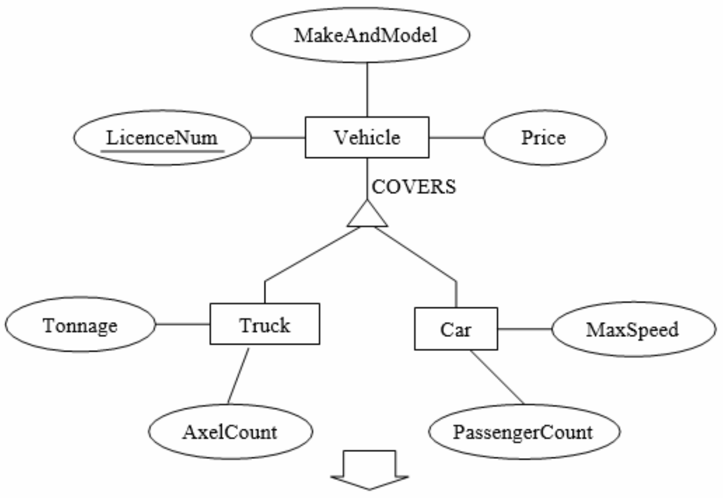
Loan_Payment

Tabular Representation of Generalization

- Create a table for each lower-level entity set only
- Columns of new tables should include
    - Attributes of lower-level entity set
    - Attributes of the superset

- The higher-level entity set can be defined as a view on the tables for the lower-level entity sets.

Example:



Truck

LicenceNum	MakeAndModel	Price	Tonnage	AxelCount

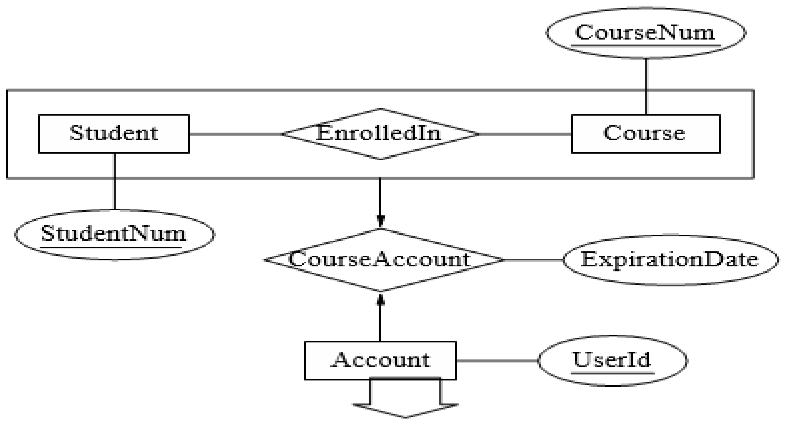
Car

LicenceNum	MakeAndModel	Price	MaxSpeed	PassengerCount

Tabular Representation of Aggregation

To represent relationship set involving aggregation of R, treat the aggregation like an entity set whose primary key is the **primary key** of the table for R.

Example:



Student

StudentNum

Course

CourseNum

Account

UserId

EnrolledIn

StudentNum	CourseNum

CourseAccount

UserId	StudentNum	CourseNum	ExpirationDate

Key constraints

Constraint is a condition or rule on a column that restricts value in the database. Constraints are used to full fill data integrity.

Types of constraints:-

- Not null
- Unique
- Primary key
- Foreign key/ references
- Check

Another classification of constraints

Column level: - If we define a constraint on a column immediately after the column definition, then it is called as column level or inline specification of constraint.

Table level: -If a constraint is defined after all the columns defined or as a part of table definition then that is called table level or out of line specification of constraint.

Not null: - This constraint should be declared in column level or inline only. If we define this constraint with a column, then the column will not allow null values into it.

Ex:-

```
create table student (sid number(6) not null, sname varchar(10) not null);
```

while inserting data into **sid** and **sname** columns of student we should not leave these columns empty (i.e., we need to supply a value)

Ex:-

```
insert into student values(1001,'');
```

if we execute the above command in the sql prompt then it will give the following error. **'cannot insert null into STUDENT.SNAME'** because we tried to place a null value into not null column i.e., sname.

Unique:- This constraint can be defined with column level and table level also. This is used to allow only unique values into a column. If unique key is defined on number of columns then it is called as composite unique key. Composite unique key should be defined in table level only. A unique key column can contain any number of null values. Unique key is also known as candidate key.

Ex:-1

```
create table promotions1(promo_id number(6) unique, promoname varchar(10));
```

Ex:-2

```
Create table promotions2(promo_id number(6) unique,promoname
                                        Varchar(10), unique(promo_id));
```

Ex:-3

```
Create table promotions3(promo_id number(6) unique, promoname varchar2(10),
unique(promo_id,promoname));
```

Primary key:- This constraint can be defined with table level or column level. Primary key= notnull+unique. It is also used to allow only unique values and it will not allow any null values. If this constraint defined on more than one column then it is called composite primary key. Maximum number of columns in a composite key are 32.

Ex:-1

```
Create table location1(location_id number(6) primary key,address varchar(10)
not null);
```

Ex:-2

```
create table location2(location_id number(6) primary key,address varchar(10)
(location_id));
```

consider the following data to **location1** table

location_id	address
1	HYD
2	Pune
3	Nrt

- if we want to insert the following row into **location1**

Ex:-

```
insert into location1 values(2,'delhi');
```

it will give the following error **'unique constraint violated LOCATION1.LOCATION_ID'** , because we are trying to insert **'2'** which is already there in **location_id** column of **location1.**

Ex:-

```
insert into location1 values('','Gnt');
```

it will give the following error **'cannot insert NULL into LOCATION1.LOCATION_ID'** because we are trying to place null into **locations_id.**

foreign key:- This can be defined in column level and table level also. It is also called referential integrity constraint. It is used to establish relationship between two or more tables using primary key and foreign key relationship. If number of columns associated with foreign key, then it is called composite foreign key.

- the table contain the foreign key is called child or detailed table (master, details)
- the table contain the primary key (referenced by) is called parent or master table.
- To maintain this relationship between tables first we should create master table and then the master_details or details table.
- 'References' clause should be used when the foreign key constraint is inline or column level. When the foreign key constraint is out of line or table level then we have to use 'foreign key' clause.
- The column on which the foreign key dependent is called referenced key column.the referenced key column may be in the same table or in the other table.

Ex:-1

```
create table college (cid number(4) primary key, cname varchar(10),cplace
varchar(10));
```

Ex:-2

```
create table student (sid number(4) primary key, sname varchar2(10), cid
number(4) references college(cid));
```

CID	CName	CPlace
101	Cone	Cp1
102	CTwo	Cp2
103	CThree	Cp3

College

Sid	SName	Cid
1001	Anil	101
1002	Pavan	102
1003	Sankar	101
1004	Bhanu	103
1005	Karuna	102
1006	Uma	103

Student

if I want to insert a row into student shown below.

Ex:-

```
Insert into student values(1007,'sseven',105);
```

it will give the following error **'parent key not found'** because in the parent table college **105** for **cid** is not there.

Check:- This can be defined in column level or table level. It is used to define a condition on a column while creating the table. One column can be associated with any number of check constraints.

Ex:-1

```
create table dept (deptno number(3) primary key check(deptno between 10 and
99));
```

Ex:-2

```
create table dept (deptno number(3) primary key, dname varchar(10),
check(deptno between 10 and 99));
```

if we want to insert a value **'120'** into deptno of dept then we get following error **'check constraint voilated'** because deptno column will take the values b/w **10 and 99.**

Data integrity

It is a state in which all the data values are stored in the database. This will increase the quality of data in database.

Entity integrity: - it defines row as a unique entity for a particular table. It enforces through the primary key of a table.

Example:

Sid	SName	Marks
1001	Anil	60
1002	Pavan	60
1003	Sankar	70
1004	Bhanu	80

Student

- Sid in the above table is taken as primary key by using this only we can identify entities or rows of student table uniquely. Each and every row is different from other row.
- If we try to insert a row which takes Sid column value 1002, we cannot because primary key enforces us not to insert duplicate value.

Domain integrity: - it validates the entities for a given column. It enforces through data types, check constraint, foreign key, default, not null.

Example:

Sid	SName	Marks
1001	Anil	60
1002	Pavan	70
1003	Sankar	80
1004	Bhanu	

Student

- The database of sid column is number but if we want to insert into a string value into sid, the system will not accept it because while creating student table we have given the sid column as number data type.

Referential integrity:- it preserves the defined relationship between the tables when the rows are inserted or deleted.

CID	CName
101	Cone
102	CTwo
103	CThree

College

Sid	SName	Cid
1001	Anil	101
1002	Pavan	102
1003	Sankar	101
1004	Bhanu	103
1005	Karuna	102
1006	Uma	103

Student

- Referential integrity enforces inserting rows to a related table if there is no associated row is primary or master table.
- If we want to insert into a row into student table by giving **105** for **cid** column we are unable to insert it because in the parent Table College 105 is not there for cid.

Query language

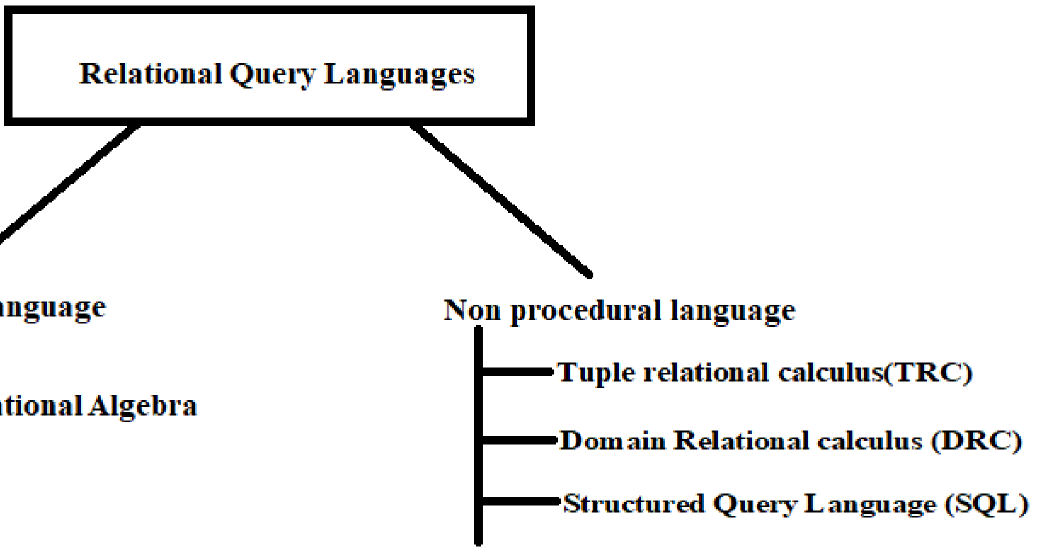
A query language is a language in which users request information from the database. Basically, there are two types of query languages.

- **Procedural language**
- **Non procedural language**

Procedural language: In this the user gives the instructions to the system that **"what information is required from the database"** and **"what are the sequences of steps on the database to get that information"**.

Non procedural language: In this user specifies only **"what information is required from the database"** without giving specific procedure to get that.

General classification:


Relational Query Languages
Procedural language
Relational Algebra
Non procedural language
Tuple relational calculus(TRC)
Domain Relational calculus (DRC)
Structured Query Language (SQL)

Relational Algebra Language: Because it is a procedural language, a query in a relational algebra language has to specify not only **"what information required"** but also **"how the information is getting"**.

To write queries in relational algebra language, we have to know about some operators

Basic operations:

- Select (σ)
- Project (∏)
- Set union (U)
- Set difference (-)
- Cartesian product (×)
- Rename(ρ)

Select : The select operator **σp(r)** selects those tuples from the relation (table) **r** , which satisfies the predicate **P**.

The predicate **P** may contain

1. **Attributes/ columns**
2. **Literals**
3. **Comparison operators like <,>, <=,>=, =**
4. **Logical operators like ^, v.**

Degree:- Degree of any relation can be defined as the no. of columns (total no. of columns) in that relation.

Cardinality: cardinality of a relation can be defined as the total no. of tulles in that relation.

Consider the relation **emp** as follows

Eid	Ename	Salary	Deptno
E1	Bhanu	10000	10
E2	Sankar	20000	20
E3	Pavan Kumar	30000	10
E4	Gopi	40000	30
E5	Srikanth	50000	20

Query: Get the information (details) of those employees who are drawing salary greater than **10000** and working in **20th** department.

SQL: select * from emp where sal >10000 and deptno=20;

Relational Algebra (R.A): $\sigma_{sal>10000 \wedge deptno=20}$ (emp)
Result:

Eid	Ename	Salary	Deptno
E2	Sankar	20000	20
E5	Srikanth	50000	20

- Degree of select (**σ**) is equal to the degree of given relation.
- Cardinality of select (**σ**) is less than or equal (**<=**) to the degree of given relation.

In the above example degree of result (obtained by applying on emp table) is **4** and cardinality is **2** (which is less than the given relation cardinality)

Project(∏): The project operation **∏s(r)** will project attribute list **s** from **r(R)** where

S ⊆R. Any duplicated in the result are automatically removed

- Degree of **π** will be **<=** degree(R)
- Cardinality of **π** will be = cardinality (R)

Where R is the given relation.

Query: Get ename , eid of all employees from the relation Emp

SQL: select ename,eid from emp;

R.A: ∏eid,ename(emp)

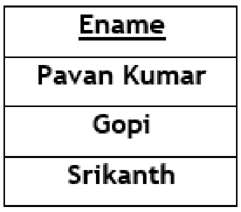
Result:

Eid	Ename
E1	Bhanu
E2	Sankar
E3	Pavan Kumar
E4	Gopi
E5	Srikanth

Query: Get the employee names who are earning salary greater than 20,000

SQL: select ename from emp where sal >20000;

R.A : ∏ ename(σsal > 20000(emp))



Ename
Pavan Kumar
Gopi
Srikanth

Set union(U):- it is the binary operation between the two relations **r** and **s**. denoted by **r U s** . It is the union of set of tuples of the two relations. Duplicate tuples are automatically removed from the result. A tuple will appear in **r U s** if it exists in **r** or in **s** or both.

For **U** to be possible, **r** and **s** must be compatible

- **r** and **s** must be of same degree i.e. they must have same no of attributes.
- For all **i**, the domain of **i**th attribute of **r** must be same as the domain of the **i**th attribute of **s**.

Cardinality of (r U s) = cardinality of **(r)** + cardinality **(s)** – cardinality **(r ∩ s)**

Consider the relations

sname	Account
Aijay	A1
Vijay	A2
Ram	A3

Student 1

sname	loan
Vishal	L1
Ram	L2

Student 2

Query:- Get the names of those students who have either account or loan or both at the bank.

SQL: select sname from student1 union select sname from student2;

R.A: ∏ sname(student1) **U** ∏ sname(student2)

Result:

Sname
Aijay
Vijay
Ram
Vishal

Set Difference (-): The set difference operation (**r - s**) between two relations **r** and **s** produced a relation with tuples which are in **r** but not there in **s**. To possible **r-s**, **r** and **s** must be compatible.

Cardinality of **r- s** = cardinality (**r**) – cardinality (**r ∩ s**)

Query: Get the names of those students who have account in the bank but do not have loan.

SQL: select sname from student1 minus select sname from student2;

RA: ∏sname(student1) - ∏sname(student 2)

Result:

Sname
Aijay
Vijay

Cartesian Product (X): The Cartesian product between two relations **r** and **s** Degree **(r x s) = degree (r) + degree (s)**

Cardinality (r x s) = cardinality (r) * cardinality (s)

Query: Find the Cartesian product between two relations student1 and student2

SQL: select * from student1, student2;

R.A : student1 X student2.

Result:

Student1.sname	Account	Student2.sname	Loan
Aijay	A1	Vishal	L1
Aijay	A1	Ram	L2
Vijay	A2	Vishal	L1
Vijay	A2	Ram	L2
Ram	A3	Vishal	L1
Ram	A3	Ram	L2

Sub queries/Nested queries/ sub select/inner select

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

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

There are a few rules that subqueries must follow

- 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.

Subqueries with the Select Statement

SQL subqueries are most frequently used with the **Select** statement.

Syntax:

```
SELECT column_name FROM table_name WHERE column_name expression

    operator ( 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:

Syntax:

```
SELECT * FROM EMPLOYEE WHERE ID IN (SELECT ID FROM EMPLOYEE
  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

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:

```
INSERT INTO table_name (column1, column2, column3....) SELECT *
FROM table_name
```

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.

Syntax:

```
INSERT INTO EMPLOYEE_BKP SELECT * FROM EMPLOYEE WHERE ID IN
(SELECT ID FROM EMPLOYEE);
```

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:

```
UPDATE table SET column_name = new_value WHERE VALUE OPERATOR
    (SELECT COLUMN_NAME FROM TABLE_NAME 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 0.25 times in the EMPLOYEE table for all employee whose AGE is greater than or equal to 29.

Syntax:

```
UPDATE EMPLOYEE SET SALARY = SALARY * 0.25 WHERE AGE IN
   (SELECT AGE FROM CUSTOMERS_BKP 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

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:

```
DELETE FROM TABLE_NAME WHERE VALUE OPERATOR
    (SELECT COLUMN_NAME FROM TABLE_NAME 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.

Syntax:

```
DELETE FROM EMPLOYEE WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKP
  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

Aggregate functions

Aggregate functions in DBMS take multiple rows from the table and return a value according to the query. All the aggregate functions are used in **Select** statement.

Types of SQL Aggregation Function

1) **Count()**
2) **Sum()**
3) **Avg()**
4) **Min()**
5) **Max()**

Count():

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

Syntax:

SELECT COUNT(column_name) FROM table-name

Consider the following **Emp** table

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

SQL query to count employees, satisfying specified condition is,

Syntax:

```
  SELECT COUNT(name) FROM Emp WHERE salary = 8000;
Result of the above query will be,
count(name)
2
```

Example of **COUNT (distinct)**

Consider the **Emp** table

Syntax:

```
  SELECT COUNT(DISTINCT salary) FROM emp;
Result of the above query will be,
count(distinct salary)
4
```

Sum()

SUM function returns total sum of a selected columns numeric values

Syntax:

```
    SELECT SUM(column_name) from table-name;
```

Using **SUM()** function

Consider the **Emp** table. SQL query to find sum of salaries will be

Syntax:

```
SELECT SUM(salary) FROM emp;
Result of above query is,
SUM(salary)
41000
```

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

Syntax:

SELECT AVG(column_name) FROM table_name;

Using **AVG()** function

Consider the **Emp** table. SQL query to find average salary will be,

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

Result of the above query will be,

```
avg(salary)

8200
```

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

Syntax:

SELECT MIN(column_name) from table-name;

Using **MIN()** function

Consider the **Emp** table, SQL query to find minimum salary is,

Syntax: **SELECT MIN(salary) FROM emp;**

Result will be,

```
MIN(salary)

6000
```

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

Syntax:

SELECT MAX(column_name) from table-name;

Using **MAX()** function

Consider the **Emp** table, SQL query to find the Maximum salary will be,

Syntax: **SELECT MAX(salary) FROM emp;**

Result of the above query will be,

MAX(salary)

10000

ORDER BY

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

Syntax:

SELECT column-list|* FROM table-name ORDER BY ASC | DESC;

Using default **Order by**

Consider the following **Emp** table,

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

Syntax:

SELECT * FROM Emp ORDER BY salary;

The above query will return the resultant data 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	Pavan	44	10000

Using Order by **DESC**

Consider the **Emp** table described above,

Syntax:

```
SELECT * FROM Emp ORDER BY salary DESC;
```

The above query will return the resultant data in descending order of the salary.

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

GROUP BY

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:

```
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** or in other words, we will be grouping employees based on their salaries, hence, as a result, we will get a data set, with unique salaries listed, along side the first employee's name and age to have that salary. **group by** is used to group different row of data together based on any one column.

SQL query for the above requirement will be,

Syntax:

```
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 **Emp** table, SQL query will be,

Syntax:

```
SELECT name, salary FROM Emp WHERE age > 25 GROUP BY salary;
```

Result will be.

name	salary
Rohan	6000
Shane	8000
Scott	9000

Different types of joins

SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data. It 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, which is known as, **Self Join**.

Types of JOIN

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.

Syntax:

SELECT column-name-list FROM table-name1 CROSS JOIN table-name2;

Example of Cross JOIN

Following is the **class** table,

ID	NAME
1	abhi
2	adam
3	alex

and the **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI

Cross JOIN query will be,

Syntax:

SELECT * FROM class CROSS JOIN class_info;

The result set table will look like,

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

As you can see, this join returns the cross product of all the records present in both the tables.

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 SQL query.

Syntax:

```
SELECT column-name-list FROM table-name1 INNER JOIN table-name2
WHERE table-name1.column-name = table-name2.column-name;
```

Example of INNER JOIN

Consider a **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu

and the **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI

Inner JOIN query will be,

Syntax:

```
SELECT * from class INNER JOIN class_info where class.id = class_info.id;
```

The result set 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.

The syntax for Natural Join is,

Syntax:

```
SELECT * FROM table-name1 NATURAL JOIN table-name2;
```

Natural join query will be,

Syntax:

```
SELECT * from class NATURAL JOIN class_info;
```

The result set 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 resultset table with the matched data from the two tables and then the remaining rows of the left table and null from the right table's columns.

Example of Left Outer Join

Here is the **class** table,

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

and the **class_info** table,

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

Left Outer Join query will be,

Syntax:

```
SELECT * FROM class LEFT OUTER JOIN class_info ON (class.id =
class_info.id);
```

The resultset 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 resultset table with the **matched data** from the two tables being joined, then the remaining rows of the **right** table and null for the remaining **left** table's columns.

Example of Right Outer Join

Once again, the **class** table,

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

and the **class_info** table,

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

Right Outer Join query will be,

Syntax:

SELECT * FROM class RIGHT OUTER JOIN class_info ON (class.id = class_info.id);

The resultant 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 resultset table with the **matched data** of two table then remaining rows of both **left** table and then the **right** table.

Example of Full outer join is,

The **class** table,

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

and the **class_info** table,

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

Full Outer Join query will be like,

Syntax:

```
SELECT * FROM class FULL OUTER JOIN class_info ON (class.id =
class_info.id);
```

The resultset 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

Relational Set operations

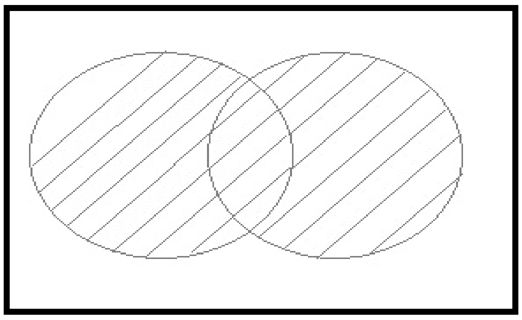
Relational set operators are used to combine or subtract the records from two tables. These operators are used in the **SELECT** query to combine the records or remove the records. In order to set operators to work in database, it should have same number of columns participating in the query and the datatypes of respective columns should be same. This is called **Union Compatibility**. The resulting records will also have same number of columns and same datatypes for the respective column.

There are **4** different types of SET operations.

1. **UNION**
2. **UNION ALL**
3. **INTERSECT**
4. **MINUS**

UNION Operation

UNION is used to combine the results of two or more **SELECT** statements. However, it will eliminate duplicate rows from its resultset. In case of union, number of columns and datatype must be same in both the tables, on which UNION operation is being applied.



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,

Syntax:

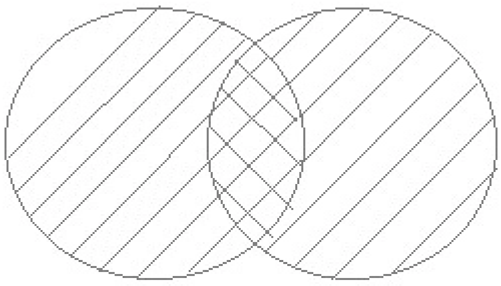
SELECT * FROM First UNION SELECT * FROM Second;

The resultset table will look like,

ID	NAME
1	abhi
2	adam
3	Chester

UNION ALL

This operation is also similar to UNION, but it does not eliminate the duplicate records. It shows all the records from both the tables. All other features are same as UNION. We can have conditions in the SELECT query.



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

Syntax:

SELECT * FROM First UNION ALL SELECT * FROM Second;

The resultset table will look like,

ID	NAME
1	abhi
2	adam
2	adam
3	Chester

INTERSECT

This operator is used to pick the records from both the tables which are common to them. In other words, it picks only the duplicate records from the tables. Even though it selects duplicate records from the table, each duplicate record will be displayed only once in the result set.

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,

Syntax:

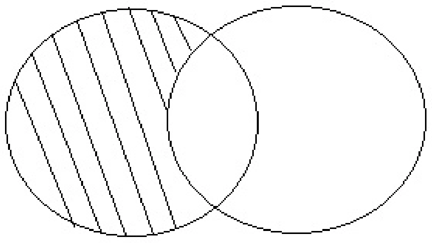
```
SELECT * FROM First INTERSECT SELECT * FROM Second;
```

The resultset table will look like

ID	NAME
2	adam

MINUS

This operator is used to display the records that are present only in the first table or query, and doesn't present in second table / query. It basically subtracts the first query results from the second.



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,

Syntax:

```
SELECT * FROM First MINUS SELECT * FROM Second;
```

The resultset table will look like,

ID	NAME
1	abhi