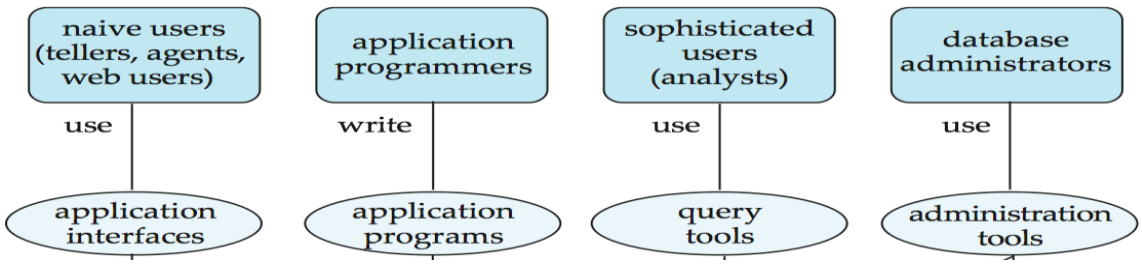
Database Users :-

A primary goal of a database system is to retrieve information from and store new information into the database. People who work with a database can be categorized as database users or database administrators.

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

- **Naive users** are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example - a clerk in the university.
- **Application programmers** are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces.
- **Sophisticated users** interact with the system without writing programs. Instead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submit queries to explore data in the database fall in this category.
- **Specialized users** are sophisticated users who write specialized database applications.



Database Architecture :-

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the storage manager and the query processor components.

Storage Manager

The *storage manager* is the component of a database system that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:

- **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.

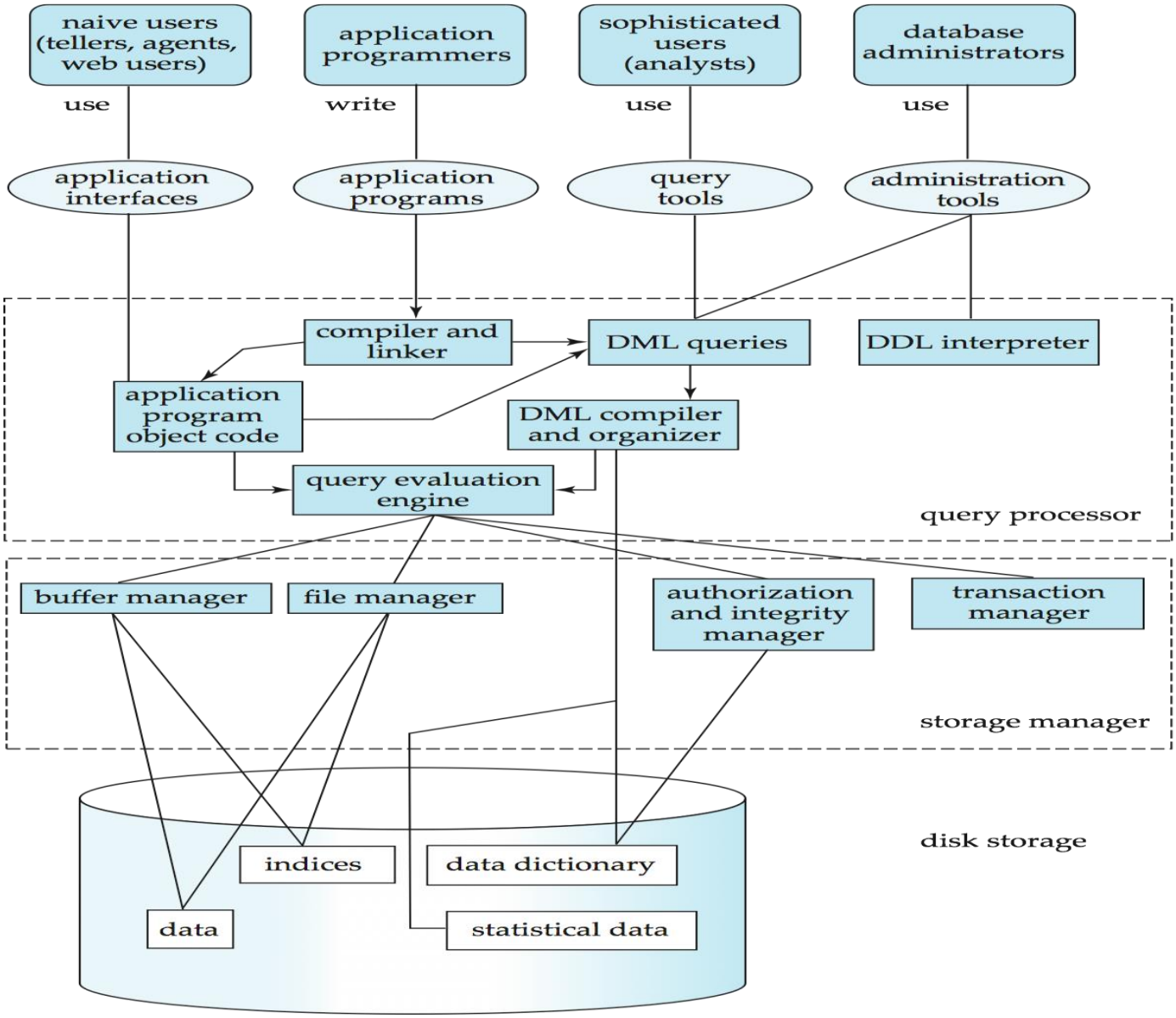
- **Transaction manager**, which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.
- **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
- **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.
- **Data files**, which store the database itself.
- **Data dictionary**, which stores metadata about the structure of the database, in particular the schema of the database.
- **Indices**, which can provide fast access to data items. Like index of a textbook, a database index provides pointers to those data items that hold a particular value.

Query Processor

The query processor components include:

- **DDL interpreter**, which interprets DDL statements and records the definitions in the data dictionary.
- **DML compiler**, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

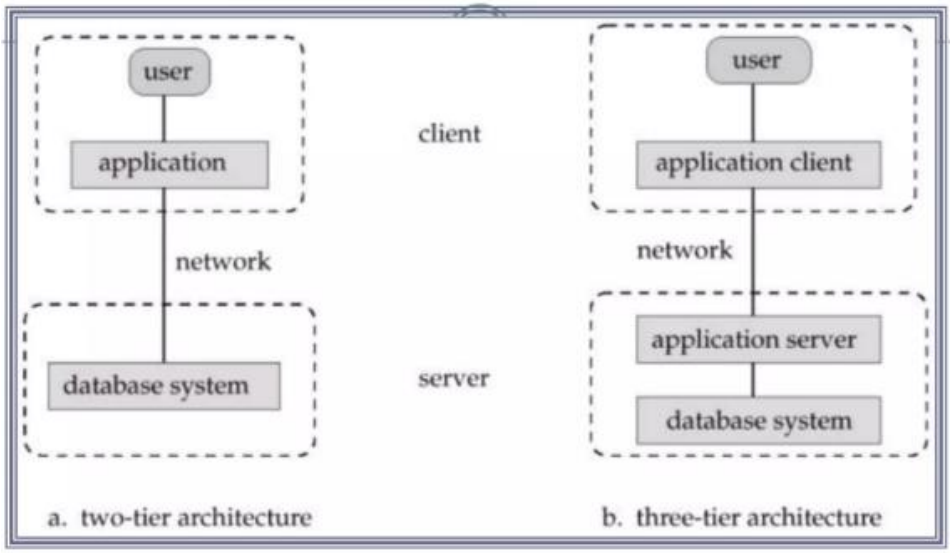
  A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs query optimization; that is, it picks the lowest cost evaluation plan from among the alternatives.
- **Query evaluation engine**, which executes low-level instructions generated by the DML compiler.



Database System Architecture

Two-tier architecture: E.g. client programs using ODBC/JDBC to communicate with a database

Three-tier architecture: E.g. web-based applications, and applications built using "middleware"



Complex Attribute

The complex attribute in DBMS involves both multivalued and composite attributes.

Example: someone might have more than one house, and each house might have more than one phone. The phone is then considered a complex attribute.
The phone number is a composite attribute of the area code, exchange, and line number. Complex attributes are often used in database design to represent relationships between entities.

Key Attribute

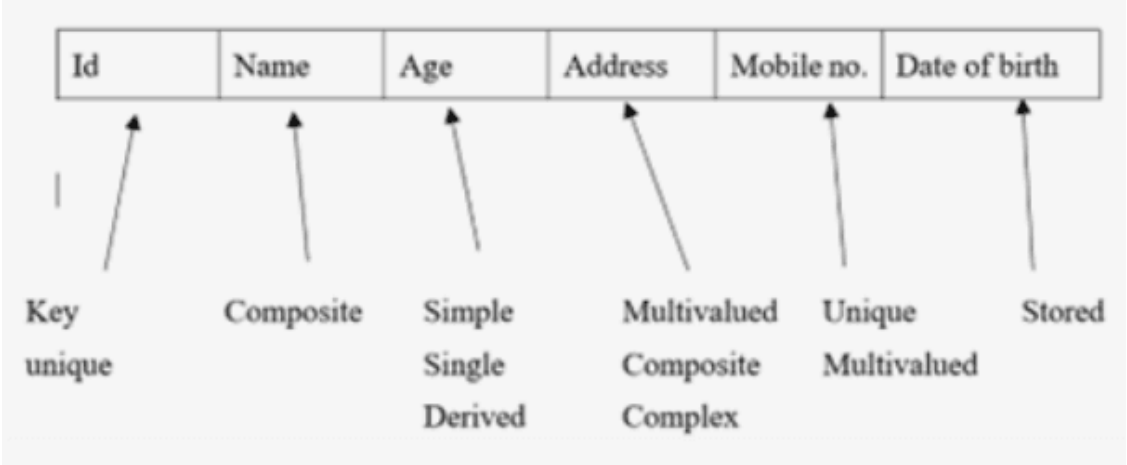
DBMS's key attributes are used to uniquely identify each row in a table. Usually, there is more than one key attribute in a table (primary key and foreign key).

Example: In a table of employees, the employee ID would be the primary key, while the manager ID would be the foreign key.

Stored Attribute

Stored attributes are the data that remain constant and fixed for an entity instance. These values help in deriving the derived attributes.

Example: consider a customer entity in a bank. The customer's name, age, and address would be stored attributes. The customer's account balance (a derived attribute) could be calculated based on the transactions (another stored attribute) associated with the customer.



Relational Query Language

Relational Query Language is used by the user to communicate with the database user requests for the information from the database. Relational algebra breaks the user requests and instructs the DBMS to execute the requests. It is the language by which the user communicates with the database. They are generally on a higher level than any other programming language. These relational query languages can be Procedural and Non-Procedural.
There are two types of relational query language:

- Procedural Query Language
- Non-Procedural Language

Relational Algebra - The relational algebra is a ***procedural* query language**. It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are *select, project*, *union*, *set difference*, *Cartesian product,* and *rename*. set intersection, natural join, division, and assignment.

Fundamental Operations - The select, project, and rename operations are called *unary* operations, because they operate on one relation. The other three operations operate on pairs of relations and are, therefore, called *binary* operations.

Select Operation - The **select** operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ. The argument relation is in parentheses after the σ.

Example -

1) To select those tuples of the *loan* relation where the branch is “dehradun”.

$$\sigma_{branch\text{-}name} = \text{“dehradun”} (loan)$$

2) Find all tuples in which the amount is more than 5400.

$$\sigma_{amount} > 5400 (loan)$$

We allow comparisons using =, _=, <, ≤, >, ≥ in the selection predicate. We can combine several predicates into a larger predicate by using the connectives *and* (∧), *or* (∨), and *not* (¬).

3) To find those tuples pertaining to loans of more than 25000 made by the dehradun branch.

$$\sigma_{branch\text{-}name} = \text{“dehradun”} \wedge_{amount} > 25000 (loan)$$

Project Operation - The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the uppercase Greek letter pi (Π). We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows in parentheses.

Example - To list all loan numbers and the amount of the loan relation.

$$\Pi_{loan\text{-}number,\ amount}(loan)$$

Output -

loan-number	*amount*
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

Composition of Relational Operations –

Example - Find those customers who live in rishikesh.

$$\Pi_{customer\text{-}name}\ (\sigma_{customer\text{-}city} = \text{“rishikesh”}\ (customer))$$

Union Operation – Outputs the union of tuples from both the relations. Duplicate tuples are eliminated automatically. It is a binary operator means it require two operands. Union operation denoted by *U*.

Example - To find the names of all bank customers who have either an account or a loan or both. To answer this query, we need the information from the *depositor* and *borrower* relation.

$$\boldsymbol{\Pi_{customer\text{-}name}\ (borrower)\ U\ \Pi_{customer\text{-}name}\ (depositor)}$$

Since relations are sets, duplicate values are eliminated.

Set Difference Operation - The **set-difference** operation, denoted by −, allows us to find tuples that are in one relation but are not in another. The expression *r* − *s* produces a relation containing those tuples in *r* but not in *s*.

Example - Find all customers of the bank who have an account but not a loan.

$$\Pi_{customer\text{-}name}\ (depositor) - \Pi_{customer\text{-}name}\ (borrower)$$

Cartesian-Product Operation - The **Cartesian-product** operation, denoted by a cross (*X*), allows us to combine information from any two relations. We write the Cartesian product of relations *r*1 and *r*2 as *r*1 *X r*2.

Example - To find the names of all customers who have a loan at the dehradun branch.

$$\Pi_{customer\text{-}name}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\sigma_{branch\text{-}name = \text{“dehradun”}}(borrower \times loan)))$$

Rename Operation - Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful to be able to give them names; the **rename** operator, denoted by the lowercase Greek letter rho (ρ), lets us do this. Given a relational-algebra expression *E*, the expression

$$\rho x\,(E)$$

returns the result of expression *E* under the name *x*.
A relation *r* by itself is considered a (trivial) relational-algebra expression. Thus, we can also apply the rename operation to a relation *r* to get the same relation under a new name.
A second form of the rename operation is as follows. Assume that a relational algebra expression *E* has arity *n*. Then, the expression

$$\rho x(A1,A2,\ldots,An)\,(E)$$

returns the result of expression *E* under the name *x*, and with the attributes renamed to *A*1,*A*2, . . .,*An*.

Set-Intersection Operation - The first additional-relational algebra operation that we shall define is **set intersection** (∩).
Example - To find all customers who have both a loan and an account.

$$\Pi_{customer\text{-}name}(borrower) \cap \Pi_{customer\text{-}name}(depositor)$$

Tuple Relational Calculus - The tuple relational calculus is a **nonprocedural** query language. It describes the desired information without giving a specific procedure for obtaining the information. A query in the tuple relational calculus is expressed as

$$\{ t \mid P(t) \}$$

that is, it is the set of all tuples t such that predicate P is true for t.
Following our earlier notation, we use $t[A]$ to denote the value of tuple t on attribute A, and we use $t \in r$ to denote that tuple t is in relation r.
Example - To find the *branch-name*, *loan-number*, and *amount* for loans of over 5420.

$$\{ t \mid t \in loan \wedge t[amount] > 5420 \}$$

Suppose that we want only the *loan-number* attribute, rather than all attributes of the *loan* relation. To write this query in the tuple relational calculus, we need to write an expression for a relation on the schema (*loan-number*). We need those tuples on (*loan-number*) such that there is a tuple in *loan* with the *amount* attribute > 1200. To express this request, we need the construct “there exists” from mathematical logic. The notation

$$\exists t \in r \ (Q(t))$$

means “there exists a tuple t in relation r such that predicate $Q(t)$ is true.”
Using this notation, we can write the query “Find the loan number for each loan of an amount greater than $1200” as

$$\{ t \mid \exists s \in loan \ (t[loan\text{-}number] = s[loan\text{-}number] \wedge s[amount] > 1200) \}$$

Domain Relational Calculus - A second form of relational calculus, called domain relational calculus, uses *domain* variables that take on values from an attributes domain, rather than values for an entire tuple. The domain relational calculus, however, is closely related to the tuple relational calculus.
Domain relational calculus serves as the theoretical basis of the widely used QBE language, just as relational algebra serves as the basis for the SQL language.

Formal Definition - An expression in the domain relational calculus is of the form

$$\{< x1, x2, \ldots, xn > \mid P(x1, x2, \ldots, xn)\}$$

where $x1, x2, \ldots, xn$ represent domain variables. P represents a formula composed of atoms, as was the case in the tuple relational calculus. An atom in the domain relational calculus has one of the following forms:

• $< x1, x2, \ldots, xn > \in r$, where r is a relation on n attributes and $x1, x2, \ldots, xn$ are domain variables or domain constants.

• $x \Theta y$, where x and y are domain variables and Θ is a comparison operator ($<$, $\leq$, $=$, $\neq$, $>$, $\geq$). We require that attributes x and y have domains that can be compared by Θ.

• $x \Theta c$, where x is a domain variable, Θ is a comparison operator, and c is a constant in the domain of the attribute for which x is a domain variable.

We build up formulae from atoms by using the following rules:

• An atom is a formula.

• If $P1$ is a formula, then so are $\neg P1$ and $(P1)$.

• If $P1$ and $P2$ are formulae, then so are $P1 \vee P2$, $P1 \wedge P2$, and $P1 \Rightarrow P2$.

• If $P1(x)$ is a formula in x, where x is a domain variable, then

$$\exists x (P1(x)) \text{ and } \forall x (P1(x))$$

we write

$$\exists a, b, c (P(a, b, c))$$

for

$$\exists a (\exists b (\exists c (P(a, b, c))))$$

Example –

1) Find the loan number, branch name, and amount for loans of over 4200:

$$\{< l, b, a > \mid < l, b, a > \in loan \wedge a > 4200\}$$

2) Find all loan numbers for loans with an amount greater than 4200:

$$\{< l > \mid \exists b, a (< l, b, a > \in loan \wedge a > 4200)\}$$

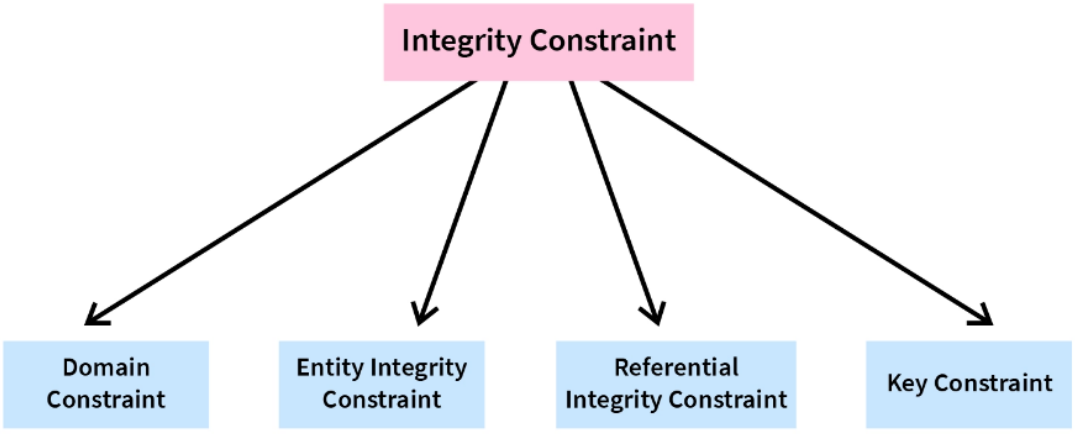
Integrity Constraints

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

Types of Integrity Constraints

There are four types of integrity constraints in DBMS:

1. Domain Constraint
2. Entity Constraint
3. Referential Integrity Constraint
4. Key Constraint



1. Domain constraints

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

Example:

ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1004	Morgan	8th	A

Not allowed. Because AGE is an integer attribute

2. Entity integrity constraints

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

Example:

EMPLOYEE

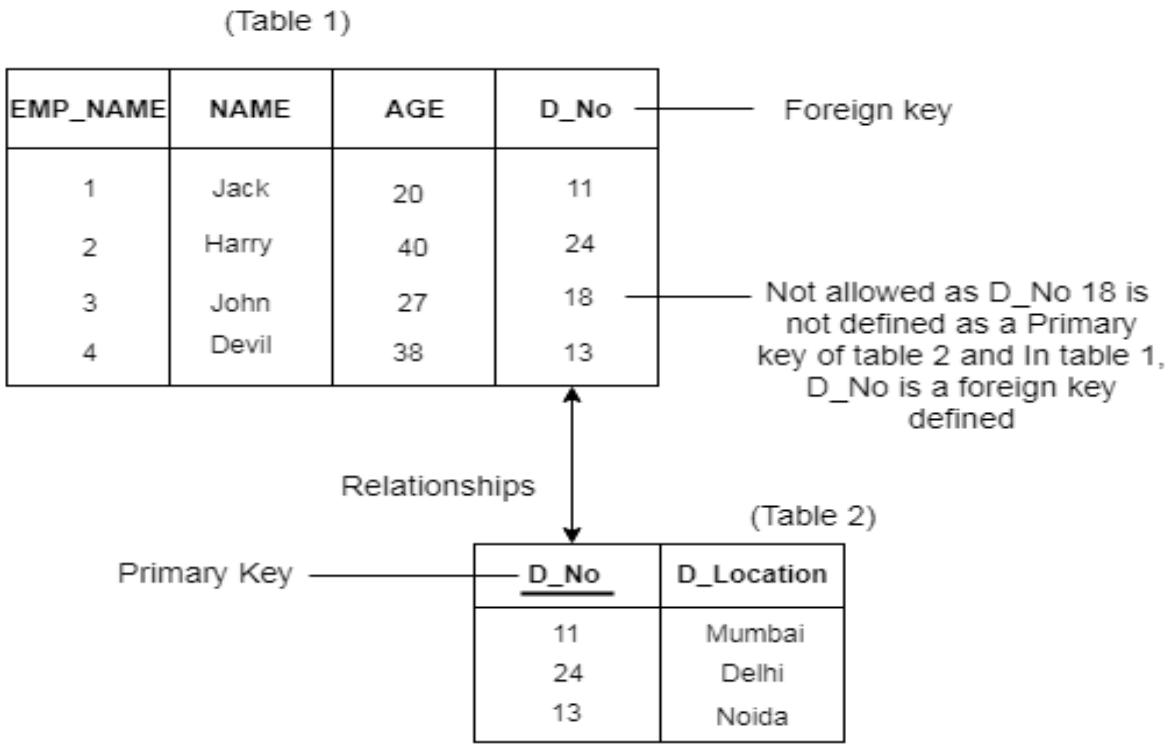
EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

Not allowed as primary key can't contain a NULL value

3. Referential Integrity Constraints

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

Example:



4. Key constraints

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

Example

ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1002	Morgan	8th	22

Not allowed. Because all row must be unique

Specialization

An entity set may include subgroupings of entities that are distinct in some way from other entities in the set. For instance, a subset of entities within an entity set may have attributes that are not shared by all the entities in the entity set.

The E-R model provides a means for representing these distinctive entity groupings. Entity set person, with attributes name, street, and city. A person may be further classified as one of the following:

• customer

• employee

Each of these person types is described by a set of attributes that includes all the attributes of entity set person plus possibly additional attributes. For example, customer entities may be described further by the attribute customer-id, whereas employee entities may be described further by the attributes employee-id and salary. The process of designating subgroupings within an entity set is called specialization. The specialization of person allows us to distinguish among persons according to whether they are employees or customers.
In terms of an E-R diagram, specialization is depicted by a *triangle* component labeled **ISA**. The label ISA stands for “is a” and represents, for example, that a customer “is a” person. The ISA relationship may also be referred to as a **superclass-subclass** relationship.

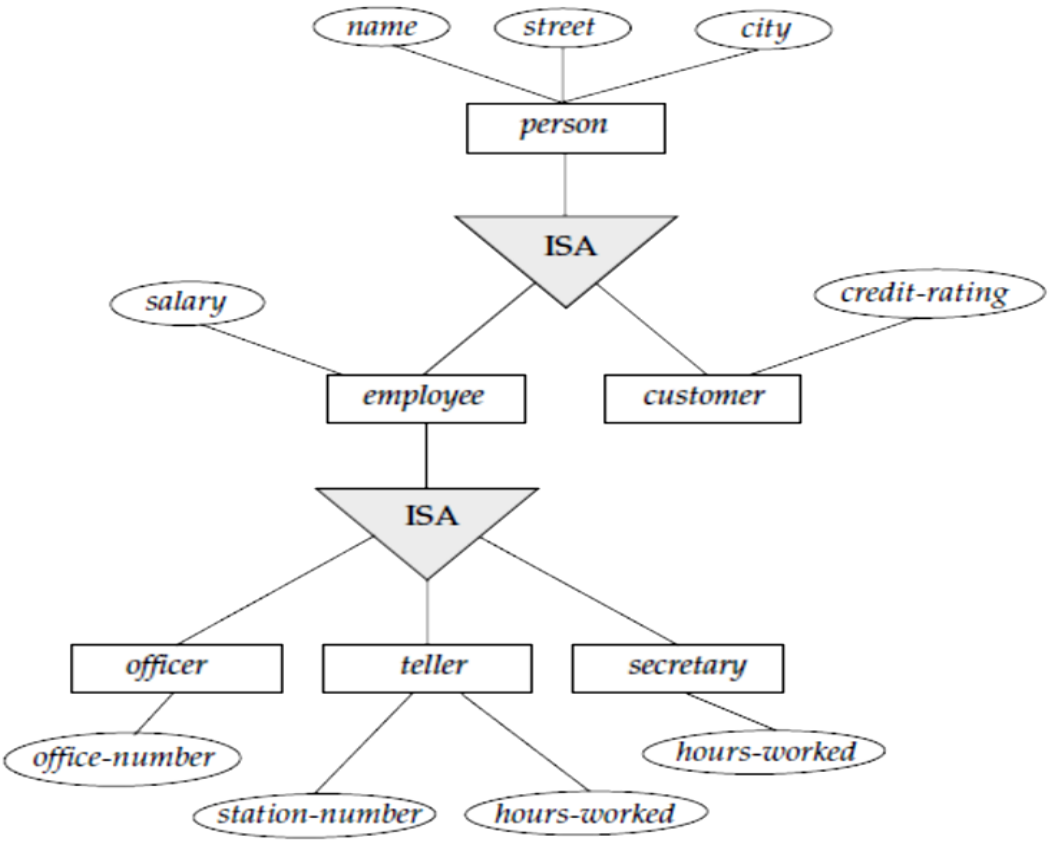


Figure : Specialization and generalization

Generalization

The refinement from an initial entity set into successive levels of entity subgroupings represents a **top-down** design process in which distinctions are made explicit. The design process may also proceed in a **bottom-up** manner, in which multiple entity sets are synthesized into a higher-level entity set on the basis of common features.

The database designer may have first identified a *customer* entity set with the attributes *name*, *street*, *city*, and *customer-id*, and an *employee* entity set with the attributes *name*, *street*, *city*, *employee-id*, and *salary*.

There are similarities between the *customer* entity set and the *employee* entity set in the sense that they have several attributes in common. This commonality can be expressed by **generalization**, which is a containment relationship that exists between a *higher-level* entity set and one or more *lower-level* entity sets. In our example, *person* is the higher-level entity set and *customer* and *employee* are lower-level entity sets. Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**, respectively. The *person* entity set is the superclass of the *customer* and *employee* subclasses.

Aggregation

One limitation of the E-R model is that it cannot express relationships among relationships. Consider the ternary relationship *works-on*. Now, suppose we want to record managers for tasks performed by an employee at a branch; that is, we want to record managers for (*employee*, *branch*, *job*) combinations. Let us assume that there is an entity set *manager*.

One alternative for representing this relationship is to create a quaternary relationship *manages* between *employee*, *branch*, *job*, and *manager*.

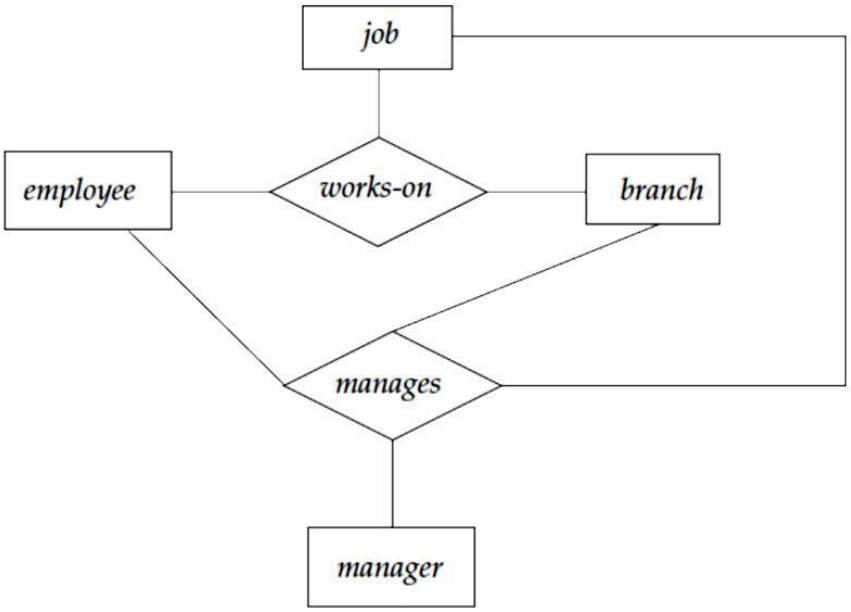


Figure : E-R diagram with redundant relationships

It appears that the relationship sets *works-on* and *manages* can be combined into one single relationship set. Nevertheless, we should not combine them into a single relationship, since some *employee*, *branch*, *job* combinations many not have a manager.

There is redundant information in the resultant figure, however, since every *employee*, *branch*, *job* combination in *manages* is also in *works-on*. If the manager were a value rather than an *manager* entity, we could instead make *manager* a multi-valued attribute of the relationship *works-on*. But doing so makes it more difficult (logically as well as in execution cost) to find, for example, employee-branch-job triples for which a manager is responsible. Since the manager is a *manager* entity, this alternative is ruled out in any case.

The best way to model a situation such as the one just described is to use aggregation.

Aggregation is an abstraction through which relationships are treated as higher level entities. Thus, for our example, we regard the relationship set *works-on* (relating the entity sets *employee*, *branch*, and *job*) as a higher-level entity set called *works-on*. Such an entity set is treated in the same manner as is any other entity set. We can then create a binary relationship *manages* between *works-on* and *manager* to represent who manages what tasks. Figure-7 shows a notation for aggregation commonly used to represent the above situation.

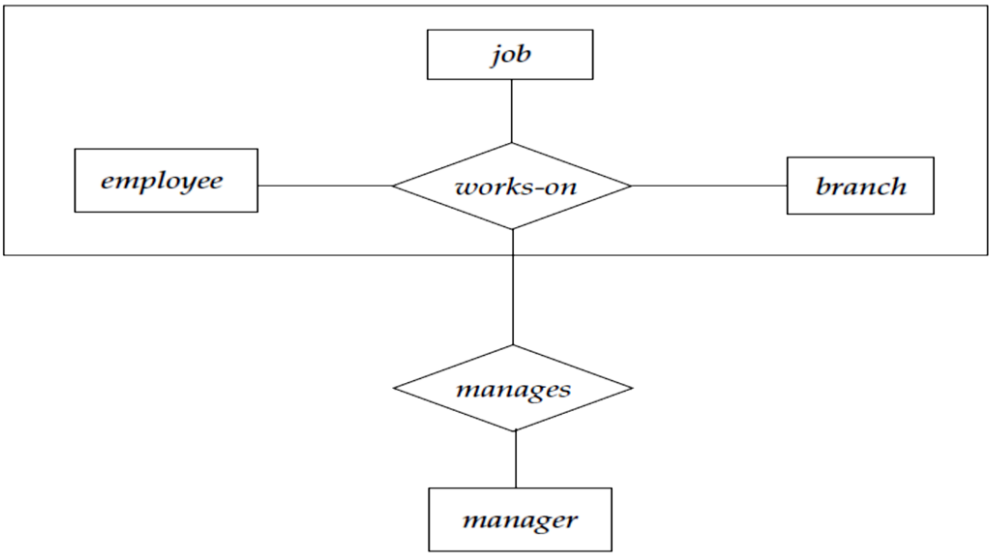
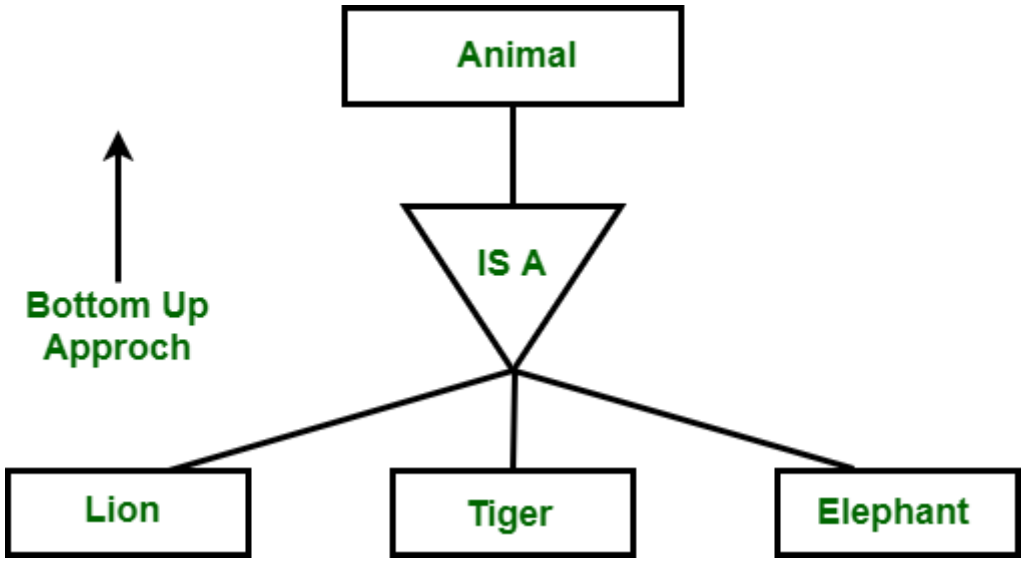


Figure : E-R diagram with aggregation

Constraints on Generalization

To model an enterprise more accurately, there are some constraints that are applicable onto the database on the particular generalization



Generalization

There are three types of constraints on generalization which are as follows:

1. First one determines which entity can be a member of the low-level entity set.
2. Second relates to whether or not entities belong to more than one lower-level entity set.
3. Third specifies whether an entity in the higher level entity set must belong to at least one of the lower level entity set within generalization.

First one determines which entity can be a member of the low-level entity set:

Such kind of membership may be one of the following:

- **Condition-defined –** In this lower-level entity sets, evaluation of membership is on basis whether an entity satisfies an explicit condition or not. For example, Let us assume, higher-level entity set student which has attribute student type. All the entities of student are evaluated by definition of attribute of student. Entities are accepted by the satisfaction of condition i.e. student type = “graduate” then only they are allowed to belong to lower-level entity set i.e. graduate student. By the satisfaction of condition student type = “undergraduate” then they are included in undergraduate student. In fact, all the lower-level entities are evaluated on the basis of the same attribute, thus it is also referred as attribute-defined.
- **User-defined –** In this lower-level entity sets are not get constrained by a condition named membership; users of database assigns entities to a given entity set. For example, Consider a situation where after 3 months of employment, the employees of

the university are assigned to one of four work teams. For this purpose, we represent teams them as four lower-level entity sets of higher-level employee entity set. On the basis of an explicit defining condition, a given employee is not assigned to specific team entity. User in charge of this decision makes the team assignment on an individual basis. By adding entity to an entity set, assignment is implemented.

Second relates to whether or not entities belong to more than one lower-level entity set :
Following is one of the lower-level entity sets:

- **Disjoint –** The requirement of this constraint is that an entity should not belong to no more than one lower-level entity set. For example, the entity of student entity satisfy only one condition for student type attribute i.e. Either an entity can be a graduate or an undergraduate student, but cannot be both at the same time.
- **Overlapping –** In this category of generalizations, within a single generalization, the same entity may belong to more than one lower-level entity set. For example, in the employee work-team assume that certain employees participate in more than one work team. Thus, it offers a given employee that he may appear in more than one of the team entity sets that are lower-level entity sets of employee. Thus, generalization is overlapping.

Third specifies whether pr not an entity in the higher level entity set must belong to at least one of the lower level entity set within generalization :
This constraint may be one of the following:

- **Total generalization or specialization –** According to this constraint, each higher-level entity must belong to a lower-level entity set.
- **Partial generalization or specialization –** According to this constraint, some higher-level entities may not belong to any lower-level entity set.

Participation constraint

In a Relationship, Participation constraint specifies the presence of an entity when it is related to another entity in a relationship type. It is also called the minimum cardinality constraint.

This constraint **specifies the number of instances of an entity that are participating in the relationship type.**

There are two types of Participation constraint:

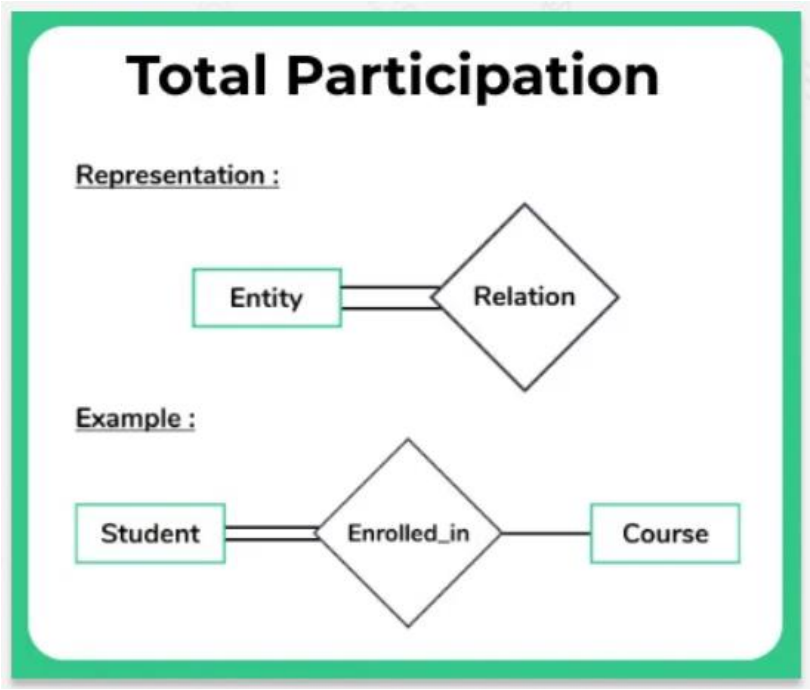
- Total participation
- Partial participation

Total participation constraint

- It specifies that **each entity present in the entity set must mandatorily participate in at least one relationship instance of that relationship set**,for this reason, it is also called as mandatory participation
- It is **represented using a double line** between the entity set and relationship set

Example:

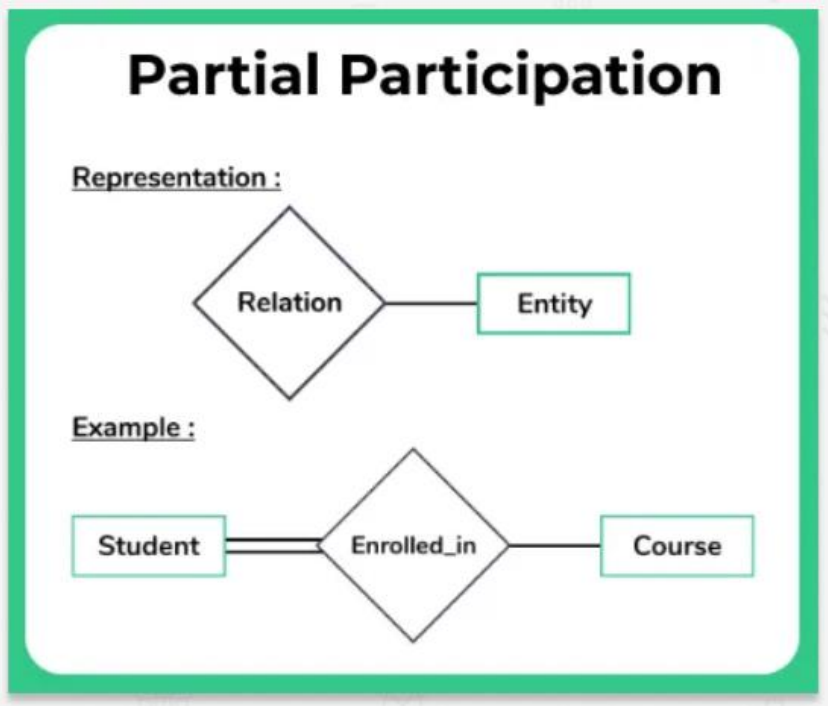
- It specifies that each student must be enrolled in at least one course where the **"student" is the entity set** and **relationship "enrolled in" signifies total participation**
- It means that **every student must have enrolled at least in one course**



Partial participation

- It specifies that **each entity in the entity set may or may not participate in the relationship instance of the relationship set**, is also called as optional participation
- It is represented using a **single line between the entity set and relationship set** in the ER diagram

Example: A **single line between the entities i.e courses and enrolled in a relationship signifies the partial participation,**which means there might be some courses where enrollments are not made i.e enrollments are optional in that case



Strong Entity

Strong Entity is independent to any other entity in the schema. A strong entity always have a primary key. In ER diagram, a strong entity is represented by rectangle. Relationship between two strong entities is represented by a diamond. A set of strong entities is known as strong entity set.

Weak Entity

Weak entity is dependent on strong entity and cannot exists without a corresponding strong. It has a foreign key which relates it to a strong entity. A weak entity is represented by double rectangle. Relationship between a strong entity and a weak entity is represented by double diamond. The foreign key is also called a partial discriminator key.

Important differences between Strong Entity and Weak Entity.

Sr. No.	Key	Strong Entity	Weak Entity
1	Key	Strong entity always have one primary key.	Weak entity have a foreign key referencing primary key of strong entity.
2	Dependency	Strong entity is independent of other entities.	Weak entity is dependent on strong entity.
3	Represented by	A strong entity is represented by single rectangle.	A weak entity is represented by double rectangle.
4	Relationship Representation	Relationship between two strong entities is represented by single diamond.	Relationship between a strong and weak entity is represented by double diamond.
5	Participation	Strong entity may or may not participate in entity relationships.	Weak entity always participates in entity relationships.