**DBMS (R20) 2nd Year 1st Semester 2021-22 AY**

**Unit-2 Syllabus**

**Entity Relationship Model**: Introduction, Representation of Entities, Attributes, Entity Set, Relationship, Relationship Set, Key Constraints, Participation Constraints, Weak Entity Sets, Sub Classes, Super Class, Inheritance, Specialization, Generalization, Aggregation using ER Diagrams.

**Relational Model**: Introduction to Relational Model, Concepts of Domain, Attribute, Tuple, Relation, Importance of Null Values, Integrity Constraints (Domain, Key constraints, Referential Integrity constraints) and their Importance.

**BASIC SQL**: Simple Database Schema, Data Types, Table Definitions (Create, Alter), Different DML Operations (Insert, Delete, Update), Basic SQL Querying using Where Clause, Arithmetic Expressions in select clause, SQL Functions (Date and Time, Numeric, String Conversion).

**Entity-Relationship Model**: This is a semantic data model used to represent a database as a collection of entities and relationships among entities. This is a simple model used to come up with a good initial description of the database which is close to user’s perspective of the database. In this model, database is represented using symbols to represent various features.

**Entity**: ***Entity*** is any real world object that is unique or distinguishable from other objects. The set of all entities of same type is called an ***Entity Set***. An entity is described using properties called ***Attributes***. All entities within an entity set are described using same set of properties. An entity may have one or more values for some of its attributes. *Entity sets need not be disjoint*. i.e., an entity may be a member of two different entity sets.

**Ex** **1**: A Student is an entity who is described using attributes like S\_id, S\_Name, S\_DoB, S\_Mobile etc which stands for “id, name, date of birth, mobile number” of the Student. When a student has the voting age, he/she is considered as a Voter i.e, same person is a member of Students entity set and also Voters entity set. In the context of a voter, he/she may be described with a different set of properties like “Voter\_id, Aadhar\_No, Name, Gender, and Age” etc. The attribute “S\_Mobile” may have more than one value for some students.

**Ex 2**: A Bank account is also an entity which is described by properties like “Acc\_No, Acc\_Type, Branch\_Name, Balance” etc.

**Types of Attributes:** Attributes are categorized based on various criteria as follows.

**i**) Depending on number of values it takes:

a) Single Valued Attribute: If an attribute possess only one value for an entity, it is called single valued attribute. **Ex**: Date of Birth of a Student, Type of a Bank Account.

b) Multi Valued Attribute: If an attribute possess more than one value for an entity, it is called multi valued attribute. **Ex**: Contact number of a Person, Skill set of an Employee.

**ii**) Depending on the way the value is obtained:

a) Stored Attribute: If the value of an attribute is supplied by the user, it is called stored attribute. **Ex**: Name of a person, Contact number, Address, Skill Set etc.

b) Derived Attribute: If the value of an attribute is computed from the value of some other attribute, it is called derived attribute. **Ex**: Age of a person computed from his date of birth, Average marks of a student obtained from marks of all course etc.

**iii**) Depending on whether the attribute can be divided into parts:

a) Composite Attribute: If an attribute value can be divided into parts, it is called composite attribute. **Ex**: Name of a person can be divided into First Name, Middle Name and Last Name, Address has components Dr\_No, Stree\_Name, Area, Town etc.

b) Component Attribute: Individual parts of a composite attribute are called its components. **Ex**: First Name, Middle Name, Last Name are called components of composite attribute “Person Name”.

Entity Sets are represented using a *Rectangle* and attributes are represented using different outlines of *Ellipse*.

Students

**Figure 2.1 Entity Set.**

Figure 2.1 shows the representation of an entity set. Name of the entity set is written in the rectangle. Attributes are represented using ellipse and are connected to entity set using lines. Single valued attributes are represented using ellipse. Double ellipse represents multi valued attributes. Component attributes are attached to corresponding composite attribute using ellipses. Derived attributes are represented using dotted ellipse. Name of the attribute appears inside the ellipse.

**Relationship**: A relationship is an association between entities of one or more entity sets. It represents in what way the entities are associated with each other. A relationship defined among entities of one entity set is called ***Unary*** relationship. A relationship defined among entities of two entity sets is called ***Binary*** relationship. A relationship defined among entities of three entity sets is called ***Ternary*** relationship. A set of relationships of same type is called a *Relationship set*.

**Ex 1**: Consider the following entity sets along with their descriptive properties. ***Employees*** (Emp\_ID, Emp\_Name, Batch) ***Departments*** (Dept\_Name, Head\_Office, Total\_Offices, Dept\_Budget) and ***Offices*** ( Office\_Address, Dept\_Name, Capacity) . Each *Employee* is described using ID of the employee, Name of the employee and the Batch to which he/she belong. Each *Department* is described by Name of the department, Location of its head office, Total number of offices the department has at various locations and Total budget of the department. Each *Office* is described by Office address, which department the office belongs to and total number of employees working there.

Now, consider the relationship “*An employee reports to other employee.*” which is defined between two employees of *Employees* entity set. This is an example of *unary relationship* defined between entities of one entity set. In this relationship, the reporting employee plays the role of a *subordinate* where as the other employee plays the role of a *supervisor*. In this example, *subordinate* and *supervisor* are called role indicators. In unary relationship, as the participating entities are from the same entity set, we need to specify the roles explicitly. For others, roles are implicit.

Consider the relationship “*An employee works in some department*.” which is defined between an employee of *Employees* entity set and a department of *Departments* entity set. This is an example of *binary relationship* defined between entities of two entity sets.

Consider the relationship “*An employee works in some department located at a particular address.”* which is defined between an employee of *Employees* entity set and a department of *Departments* entity set and an entity of *Offices* entity set. This is an example of *ternary relationship* defined between entities of three entity sets.

Employees

Reports\_To

Supervisor Subordinate

Figure 2.2 **Unary Relationship**

Employees

Departments

Works\_in1

Figure 2.3 **Binary Relationship**

Employees

Departments

Works\_in2

Offices

Figure 2.4 **Ternary Relationship**

Relationship sets are represented using *Diamond*. Name of the relationship appears inside the diamond. Figures 2.2, 2.3 and 2.4 show Unary, Binary and Ternary relationships respectively. *Works\_in1* is binary as it involves two entity sets where as *Works\_in2* is ternary as it involves three entity sets.

-*Sometimes we can define more than one relationship between same set of entity sets*.

**Ex 1**: Consider entity sets *Employees* and *Departments*. Every employee works in some department. Also, every department is managed or headed by some employee. These two situations are represented as two relationships as follows.

Works\_in

Departments

Employees

Manages

Figure 2.5 **Two Relationships between two Entity sets**

**Ex 2**: Consider that *Customers* of a bank are allowed to take *Loans*. However, each loan must be guaranteed by some other customer who acts as *Surety* for that loan. In this context, a customer taking a loan is considered as a *Borrower* and the customer acting as surety is called the *Guarantor* for that loan. This situation can be modelled using two binary relationships between entity sets *Customers* and *Loans* as shown in Figure 2.6.

Borrowers

Loans

Customers

Guarantors

Figure 2.6 **Borrower and Guarantor Relationships**

**Key Constraints**: Consider *Employees* entity set which consists of 100 employees. Let each Employee entity is described using properties “*Emp\_ID”,”Emp\_Name”,”Batch”*. Given a *Batch* value, it is possible that more than one employee may have the same value for attribute *Batch*. Similarly more than one employee may have same name. Each entity in an entity set must be uniquely identified. The uniqueness is provided by key constraint. A ***key constraint*** is a condition which states that a minimal set of attributes of an entity set is a unique identifier to identify an entity. Considering *Employees* entity set, the attribute *Emp\_ID* forms a unique identifier. Hence it is called a key for *Employees* entity set. Sometimes two or more attributes together forms a key and such a key is called *Composite Key*. Usually entity sets have a single attribute as the key. Some entity sets may have more than one key.

**Ex**: Consider entity set *Students* described by Std\_ID, Std\_Name, Mobile\_Number, DoB. In this example, { Std\_ID } is a key. As mobile number of each student is also unique, { Mobile\_Number } forms second key. However, it is meaningful to identify a student based on Std\_ID rather than Mobile\_Number. Hence, we consider {Std\_ID} as the key. {DoB} cannot be a key as two students may have same date of birth.

-Just like entities, relationships also must be uniquely identified. A relationship involves a minimum of two entities. Hence, the key for a relationship is determined based on the cardinalities of participating entity sets.

**Mapping Cardinalities**: Let ***A*** and ***B*** are two entity sets. Let ***R*** is a relationship among entities of A and B. i.e., *R*: *A*→*B*. Mapping Cardinality of *R* determines one entity of A connects with how many entities of B and one entity of B connects with how many entities of A. Possible cardinalities are:

-***One to One***: If one entity of *A* associates with at most one entity of *B* and one entity of  *B* associates with at most one entity of *A*, then *R* is said to be one to one relationship.

**Ex**: One Employee manages only one Department and one Department is managed by only one Employee.

-***One to Many***: If one entity of *B* associates with at most one entity of *A* and one entity of *A* associates with more than one entity of *B*, then *R* is said to be one to many relationship.

**Ex**: One Person may own several Insurance Policies. But, one Policy is owned by only one Person.

-***Many to One***: If one entity of *A* associates with at most one entity of *B* and one entity of *B* associates with more than one entity of *A*, then *R* is said to be many to one relationship.

**Ex**: Many Voters cast their votes to one Candidate. But, one Voter cast his vote to one Candidate only.

-***Many to Many***: If one entity of *A* associates with one or more entities of *B* and one entity of *B* associates with one or more entities of *A*, then *R* is said to be many to many relationship.

**Ex**: One Student may enrol for many Courses. One Course may be enrolled by many Students.

Figure 2.7 shows variations of mapping cardinalities. In the figure, *A* and *B* are entity sets. *R* is the relationship between entities of *A* and *B*. *a1, a2, a3, a4* are entities of *A*. *b1, b2, b3, b4* are entities of *B*.

**Finding keys for relationships based on cardinalities**: Let **K1** is the key of entity set ***A*** and **K2** is the key of entity set ***B***.

-If *R* is one-to one, then key for *R* is either K1 or K2.

-If *R* is one-to-many from *A* to *B*, then key for *R* is K2.

-If *R* is many-to-one from *A* to *B*, then key for *R* is K1.

-If *R* is many-to-many, then key for *R* is K1∪K2.

**Participation Constraints**: When a relationship is defined between entities of two entity sets,all entities may notparticipate in that relationship. For example, consider a relationship *works\_in* between entities of *Employees* and *Departments*. Based on semantics, we expect that every employee should work in some department. i.e., every employee entity should participate at least once in the relationship *works\_in*. On the other hand, only one employee from each department participates in the *Manages* relationship because a department is headed at most by one employee at a given time.

-Participation constraint defines whether all entities of an entity set participate in a relationship or only a subset participates. If all entities of an entity set *E*participate in the relationship *R*, then *E* is said to have ***Total Participation*** in *R*. On the other hand, if only a subset of entities of *E* participate in R*,* then *E* is said to have ***Partial Participation*** in *R*.Mapping cardinalities are represented on lines joining entity sets with relationship. Total Participation is represented using thick or double line joining the entity set having total participation and the relationship.

***A B A B***

***R R***

*One-to-One**One-to-Many*

***A B A B***

***R R***

*Many-to-One Many-to-Many*

Figure 2.7 **Variations of Mapping Cardinalities**

Works\_in

M 1

Departments

Employees

1 1

Manages

Figure 2.8 **Representing Cardinalities and Participation constraints**

Figure 2.8 shows how to represent cardinalities and participation constraints. The numbers on the lines connecting entity sets with relationship shows cardinalities i.e., M and 1 indicate many-to-one cardinality of *Works\_in*. 1 and 1 indicate one-to-one cardinality of *Manages*. Double lines indicate total participation. i.e., both *Employees* and *Departments* entity sets have total participation in *works\_in* relationship. Only *Departments* entity set has total participation in *Manages* relationship. The participation of *Employees* entity set in *Manages* is partial.

**Weak Entity Set**: Sometimes it may not be possible to find key attributes for some entity sets. Even all attributes together cannot form a key. **For example**, consider that customers of a bank repay their loans in multiple instalments. Let each payment is described by “*Payment\_Number, Amount, Date*”. Suppose we wish to model *Payments* as an entity set. Two payments made for different loans may have same values for all attributes. As we does not involve *Loan\_Number* in this entity set, w cannot uniquely identify a *Payment* entity.

An entity set which does not have enough attributes to form a key is called ***Weak Entity Set***. Entities of a weak entity set are uniquely identified when they are connected to other entity set called ***Strong*** or ***Owner*** entity set. The relationship connecting strong entity set and weak entity set is called ***Identifying Relationship***. The cardinality of the identifying relationship is ***Many-to-One*** from weak entity set to strong entity set. i.e., one strong entity is associated with many weak entities. But, one weak entity is associated with only one strong entity. i.e., one loan is associated with zero or many payments. But, one payment is associated with exactly one loan. The participation of ***weak*** entity set is ***total***. The participation of ***strong*** entity set may be ***partial***. i.e., every payment would be associated with some loan. But, some loans may not have any payments made for them at that moment.

Loan\_Payment

1 M

Payments

Loans

Figure 2.9 **Weak Entity Set**

Figure 2.9 shows representation of the example weak entity set. Weak entity set is represented using double rectangle. Identifying relationship is represented using double diamond. When a set of weak entities are associated with a specific owner entity, each of them are uniquely identified based on some attribute. Such an attribute is called ***Partial key*** or ***Discriminator***.

**Ex** 2: Consider Employees of a company take insurance policies to cover their dependents. Let each dependent is described by “*Dependent\_Name, Age, Relation*”. The entity set *Dependents* is a weak entity set and the entity set *Employees* is the owner entity set. *Policies* will be the identifying relationship. *Dependent\_Name* forms the partial key. Every dependent entity is associated with exactly one employee entity. Some employees may not have any dependents where as some employees may have more number of dependents.

**Descriptive Attributes**: Consider that a university offers several courses through its departments. Students register for courses of their choice as per the university regulations. Suppose that a student is required to take some courses as credit courses and some courses as non credit courses based on his choice. i.e., a course ‘*X’* may be chosen as credit course by some student and as non credit course by a different student. Now, to represent this fact we use an attribute *For\_Credit* which take values either *Yes* or *No*. Each *Enrolment* is then described by three attributes *Student\_ID, Course\_ID, For\_Credit*. Here, the attribute *For\_Credit* is neither the property of a Student nor the property of a Course. It is the property of relationship *Enrolment*. Such a property is called descriptive attribute. An attribute which is the property of a relationship and not the property of participating entity sets is called ***Descriptive Attribute***.

Enrolments

M N

Courses

Students

Figure 2.10 **Placing Descriptive Attributes on a Relationship**

In figure 2.10, descriptive attribute *For\_Credit* is shown attached to relationship *Enrolments* and not to the entity sets. As usual, key attributes of entity sets are underlined. Double line at Students entity set indicates that every student must enrol to some course. There may be few courses that are offered but no students enrol for them. M and N indicate many-to-many cardinality. *Descriptive attributes should not be part of key for a relationship*.

**Class Hierarchies**: All entities of an entity set are described using a common set of properties. However, along with the common properties, a subset of entities may possess additional distinguishing properties. That is, there may be several subsets with in an entity set that possess common properties as well as distinguishing properties specific to them. A ***class hierarchy*** for an entity set defines a set of subsets called sub classes. Each sub class entity *inherits* all properties of its *super class* entity set. In addition, entities in a sub class possess their own properties. Identifying sub classes within a super class entity set based on distinguishing properties is called ***Specialization*** and combining sub class entity sets into a single super class entity set based on common properties is called ***Generalization***. i.e., Specialization is a ***Top Down Approach*** where as Generalization is a ***Bottom Up Approach***.

**Ex** 1: Consider there are *Employees* in a company, who are described using “*Employee\_ID*, *Emp\_Name, Lot\_Number”* where *Lot\_Number* indicate the batch of the employee. Suppose that employees are of two categories based on the way they are paid. They are: *Hourly\_Emps* and *Contract\_Emps*. Hourly based employees are paid based on the number of hours they had worked. Contract based employees are paid based on their contract value. They work for fixed number of hours. *Hourly\_Emps* are described with additional properties *Hrly\_Wage* and *Number\_of\_Hours.*. *Contract\_Emps* are described with additional properties *Contract\_ID* and *Weekly\_Pay*. Properties mentioned here are in addition to above three properties. This situation is modelled as shown in Figure 2.11. “ISA” in the triangle symbol should be read as “*Is A*”. i.e., *Employees is a Hourly\_Emps.* and *Employees is a Contract\_Emps*. Common properties are shown on super class entity set. Distinguishing properties are shown on sub class entity sets. Upward arrow on the right hand side indicates *Generalization* and downward arrow indicates *Specialization*. This hierarchy may be extended to multiple levels provided there are subclasses identified within a subclass.

Contract\_Emps

Hourly\_Emps

ISA

Employees

Figure 2.11 **Representing Class Hierarchies**

ISA

ISA

Employees

Students

Persons

ISA

PartTime\_Students

FullTime\_Students

FullTime\_Emps

Contract\_Emps

Figure 2.12 **Class Hierarchies for Example 2**

**Ex** 2: Consider there are Persons in a University. Broadly, a person may be an employee or a student. A person is described using attributes like Name, Address, Category, Aadhar number etc. These attributes are common to employees and students. However, an employee has additional properties like Employee ID, salary, Hire date, Type etc. These properties are not applicable to students. Similarly, a student is described using properties like Student ID, Student Type, CGPA, Major etc. These properties are not applicable to employees. Also, students may be full time or part time and employees may be regular or contract based. For full time employees we mention Date of Superannuation. i.e., date of retirement. For contract employees, we mention the tenure of their contract. For full time students, their hostel room number is mentioned. For part time students, address of the office where they are working is mentioned. This situation is modelled as a class hierarchy of two levels as shown in figure 2.12.

**Note**: Attributes considered here are only few of many possible attributes. In real world databases many properties that are of interest may be considered.

**Constraints on Specialization**: While identifying sub classes for a given super class entity set, following constraints are applied.

i) Overlapping Vs Disjoint.

ii) Total Vs Partial.

iii)User defined Vs Condition defined.

Consider a scenario (a) that a team of software trainees is to be divided into 3 project teams based on their performance after successful completion of training. i.e., all trainees together form super class entity set and successful trainees are specialized into 3 sub class entity sets.

Consider a second scenario (b) that all accounts in a bank are divided into sub classes based on their type.

In scenario (a), all trainees may not be considered successful after training. Hence, some trainees may be left without being included in any project team. The specialization in this case is considered to be ***Partial***. i.e., some super class entities may not appear in any of its sub classes.

In scenario (b), every account will definitely appear in one of its types. The specialization in this case is considered to be ***Total***. i.e., every super class entity appears in at least one sub class entity set.

In scenario (a), trainees if any who show exceptional performance in training may be placed in two project teams also. Such a specialization is said to be ***Overlapping***. i.e., one super class entity may be a member of two sub class entity sets.

In scenario (b), an account appears only in one sub class entity set. Such a specialization is said to be ***Disjoint***. i.e., one super class entity appears only in one sub class.

In scenario (a), a successful trainee is placed in one of the three teams based on the choice of the management so as to balance the teams. Such a specialization is called ***User defined***. i.e., placing a super class entity in one of the sub classes is the choice of the user who does the specialization.

In scenario (b) an account should be placed in a particular sub class only irrespective of who does the specialization. i.e., we can’t place a savings account in an entity set representing current accounts. A specialization in which a super class entity is placed in a particular sub class based on a condition is called ***Condition defined***.

Finally, dividing a set of trainees into three teams is *partial, overlapping* and *user defined* specialization.

Dividing bank accounts based on their type is *total, disjoint* and *condition defined* specialization.

**Aggregation**: Usually, relationships are defined between two or more entity sets. Sometimes it may be necessary to express a relationship between an entity set and a relationship set. In such cases aggregation is used. ***Aggregation*** is an abstraction through which relationship sets are treated as high level entity sets in defining new relationships. Consider a scenario in which departments of a university sponsors projects. The sponsorship related issues are monitored by some employees of the department. This situation is modelled using aggregation. *Departments* and *Projects* are entity sets. The relationship between *Departments* and *Projects* is defined by *Sponsors*. Now *Employees* of Departments *Monitors* the *Sponsorships*. Now, *Monitors* is the new relationship between the entity set *Employees* and the relationship set *Sponsors*. This is represented in figure 2.13.

Sponsors

Projects

Departments

Figure 2.13 **Representing Aggregation**

Employees

Monitors

**Developing complete E-R diagram**: E-R model is a semantic data model which allows users to come up with a good initial description of the database. Given the requirements and description of the organization, E-R diagram is developed in the following steps.

i) Identifying entity sets.

ii) Identifying relationships.

iii) Identifying key attributes and other attributes.

iv) Draw the complete E R diagram.

**Case Study** 1: Draw a complete E R diagram for University database.

Let us first describe a University and its activities at a fundamental level.

* A University has several departments.
* Each department has several professors and one among them is HOD.
* Each department offers multiple courses.
* A student can register for multiple courses offered by several departments.

Step 1: Identifying entity sets.

1) Departments (From line1), 2) Professors (From line2), 3) Courses (From line3),

4) Students (From line4)

Step 2: Identifying relationships.

1. Departments *Has* Professors. (One to Many cardinality, Total participation of both the entity sets.)
2. Departments *Headed By* Professors. (One to One cardinality, Total participation of Departments and Partial participation of Professors.)
3. Departments *Offers* Courses. (One to Many cardinality, Total participation of both the entity sets.)
4. Students *Belongs To* Departments. (Many to One cardinality, Total participation of both the entity sets (Assuming that every department has students admitted.))
5. Students *Enrolled For* Courses. (Many to Many cardinality, Total participation of both the entity sets (Assuming that every course is enrolled by some student.))

Step 3: Identifying Key and additional attributes.

Departments( *Dept\_Name*, Building, Budget)

Professors( *Emp\_ID*, E\_Name, Salary, Hiredate)

Students( *Std\_ID*, S\_Name, Dob, CGPA)

Courses(*C\_ID*, C\_Name, Pre\_Requisite, Credits)

( ***Key attributes are underlined and italicized***)

Step 4: Complete E R diagram.

Headed By

Professors

Departments

Offers

Has

Teaches

Belongs To

Enrolled For

Courses

Students

**Relational Data Model**

Relational model is a famous data model supported by many commercial DBMSs. It is simple to represent, easy to interpret and understand. A relation or a table is identified by a name. It contains a set of *columns* or *fields* or *attributes*.

A database for a given organization is a collection of linked relations or tables. A ***relation schema*** defines name of the relation, name of each attribute, type of each attribute and other integrity constraints that are applied to that relation.

The collection of schemas of all relations defines the ***database schema***.

Each relation has an ***instance*** which is a collection of *records* or *rows* or *tuples* at a given moment. The collection of instances of all relations at a given moment is called a ***database instance***.

The number of columns of a relation is called its ***Degree*** or ***Arity***. The number of rows of a relation instance is called its ***cardinality***.

Each tuple of a relation contain the same number of fields as defined by the relation. A relation is defined by ***Data Definition Language* (DDL)**which is a subset of Structured Query language used to work with relational databases. The database developed in terms of an E-R diagram should be translated into a collection of relations. E-R model cannot represent all features of a database. Relational model is well defined to capture all features of a database.

***Null Value***: A null value is used to indicate that an attribute value for a record is either unknown or not applicable.

Ex: A null value in Date\_of\_Birth column indicates that date of birth is unknown.

A null value in Middle\_Name column indicates that the person does not have middle

name.

**Integrity Constraints**: Relational model supports several integrity constraints used to model all features of a database. An ***Integrity Constraint*** (**IC**) is a condition which states that certain kinds of data only should enter a database. ICs ensure that consistency of the database would not be lost due to user operations. ICs are specified by the user at the time of database creation and are enforced by the DBMS at the time of data manipulation. A database which obeys all specified ICs is called ***legal database***. A DBMS allows only legal data to be stored in the database. Various ICs supported by relational model are:

* Key Constraint.
* Not null Constraint.
* Unique Constraint.
* Domain Constraint.
* Check Constraint.
* Foreign key Constraint or Referential Integrity.

***Key Constraint***: A key constraint is a condition which states that certain minimal set of attributes of a relation is a unique identifier to identify one record. i.e., the values of key attributes together are unique for a record.

Ex (i) : Consider the table Students( Std\_ID String, Std\_Name String, Std\_DoB Date). In tis table, each student record is uniquely identified using “Std\_ID”. Hence, the attribute “Std\_ID” forms key for this table.

Ex (ii): Consider the table Enrolled (Std\_ID String, Course\_ID String, For\_Credit String). In this table, each record is an enrolment of a student to some course. Each enrolment is either a credit enrolment or a non credit enrolment. One student will enrol for multiple courses and one course is enrolled by multiple students. Hence, { Std\_ID, Course\_ID } is a unique identifier for this able.

A key is also called ***Candidate key***. A table may contain more than one candidate keys. In such a case, one of them is taken as the basis for database design. Such a candidate key is called ***Primary key***. A super set of a candidate key is called ***Super key***.

Consider the relation: Students( Std\_ID, Std\_Name, DoB, GPA, Mail\_ID)

{ Std\_ID } and { Mail\_ID } are candidate keys.

{Std\_ID } is primary key.

{Std\_ID, Std\_Name } is a super key.

{ Std\_Name } is not a key. If a primary key contain more than one attributes, it is called ***Composite Key***.

***Not Null Constraint***: This is a column level constraint and is used to mark columns for which every record must have some value other than *null* value. Such a column may contain duplicate values.

Ex: In a *Students* table, the column *Std\_Name* can be marked as not null.

***Unique Constraint***: This is a column level constraint and is used to mark columns for which every record must have a unique value. Such a column may contain null values for some records. But two records cannot have same value in that column. The difference between “Primary key” constraint and “Unique” constraint is that a column designated as primary key does not contain duplicate values and also does not contain null values.

Ex: In a *Students* table, the column *Mail\_ID* can be marked as unique. Two students are not allowed to have same mail ID. But, some students may not have any mail ID for few days.

***Domain constraint***: This is a column level constraint and is used to specify the domain from which a column can take on values. i.e., domain constraint specifies the data type of the column.

Ex: In a table “Students”, the domain of the column “Std\_ID” may be “String’. Various domain types supported by SQL are: Integer, Number(m,n) where m is the total number of digits of the number and n is the number of digits after decimal place, Date, Varchar(n) where n is the maximum size of the string, Char(n), Time, Timestamp etc.

***Check Constraint***: A check constraint is used to specify a predicate on a column. A predicate generally specify a subset of some domain. It restricts the values of a column to fall within a subset of the domain specified for that column.

Ex: check( gender in(‘Male’,’Female’) restricts column “*gender”* to take any one from two specified values. The domain of the column is: varchar(6)

check( Salary between 10000 and 50000) restricts the *Salary* value to be in the specified range.

***Referential Integrity*** or ***Foreign key constraint***: This constraint is used to link the data between two tables. This constraint allows checking whether a value in a column of a table is present in primary key column of a linked table. If so, the value is allowed. If not, the value is not allowed.

Consider the following tables: Students (**Std\_ID**, Std\_Name, DoB),

Courses (**C\_ID**, C\_Name, Credits), Enrolments (**Std\_ID, C\_ID**)

We want each student enrolled for some course should be a bonafide student and each course enrolled by some student must be an offered course. *Students* table contains data of all bonafied students and *Courses* table contains all offered courses. Hence, *Std\_ID* values of *Enrolled* table must present in *Students* table and *C\_ID* values of *Enrolled* table must present in *Courses* table. This kind of linking is established between two tables using referential integrity or foreign key constraint. *Std\_ID* column of *Enrolled* table is referred to as foreign key referencing *Students* table. Similarly, *C\_ID* of *Enrolled* table is referred to as foreign key referencing *Courses* table.

---*Students* table and *Courses* table are called ***Parent table***s.

---*Enrolled* table is called ***Child table***.

---A column declared as foreign key always refer to primary key column of its parent table. Hence, the two columns may have different names.

---There will be one-to-many cardinality from parent to child table. i.e., one parent record points to multiple child records.

--- A record can always be inserted into parent table. But, a record can be inserted into a child table only when there is a corresponding parent record.

---A record can always be deleted from a child table. But, a record can be deleted from a parent table only when there are no child records pointing to this parent record.

---Foreign key constraint violation is possible in two cases: i) When we try to insert a child record without having its parent record. ii) When we try to delete a parent record which has child records.

---To deal with foreign key constraint violation caused due to deletion of a parent record, following options are available:

i) ***On Delete No Action***: This is the default option. In this option, a parent record deletion is not allowed when there are corresponding child records.

ii) ***On Delete Cascade***: In this option, when a parent record is deleted, the deletion is allowed and all corresponding child records are also deleted.

Ex: Consider table Table1(X primary key, Y). A record of Table1 is: <x1,y1>. Consider table Table2( Z primary key, X references Table1). A record of Table2 is: <z1,x1>. Table2 is created as: create table Table2( Z number primary key, X number references Table1 on delete cascade). When the record <x1,y1> is deleted from Table1, the record <z1,x1> is also deleted.

iii) ***On Delete set null***: In this option, when a parent record is deleted, the deletion is allowed and all corresponding child records are updated with *null* value in foreign key column..

Ex: Consider table Table1(X primary key, Y). A record of Table1 is: <x1,y1>. Consider table Table2( Z primary key, X references Table1). A record of Table2 is: <z1,x1>. Table2 is created as: create table Table2( Z number primary key, X number references Table1 on delete set null). When the record <x1,y1> is deleted from Table1, the record <z1,x1> is updated as <z1,null>

iv) ***On Delete Set Default***: In this option, when a parent record is deleted, the deletion is allowed and all corresponding child records are updated with the default value for that column set at the time of table definition.

Ex: Consider table Table1(X primary key, Y). A record of Table1 is: <x1,y1>. Consider table Table2( Z primary key, X references Table1). A record of Table2 is: <z1,x1>. Table2 is created as: create table Table2( Z number primary key, X number default x2, foreign key(X) references Table1 on delete set default). When the record <x1,y1> is deleted from Table1, the record <z1,x1> is updated as<z1,x2>.

***Basic SQL***:

SQL is a query language used to define and manipulate relational databases. SQL has several parts each of which is intended to carry out a specific task related to database operations.

Various parts of SQL are:

- Data Definition Language (DDL)

- Data Manipulation Language (DML)

- Data Control Language (DCL)

- Triggers

- Transaction Management

- Dynamic and Embedded SQL

- Accessing Remote Databases

***Data Definition Language*** subset of SQL is used to create databases. Various commands supported by DDL are: **Create** (used to create database objects like tables, views, triggers, functions, procedures and packages etc.), **Alter** (used to modify the structure of a table), **Drop** (used to remove unwanted database objects), **Truncate** (used to drop a table and create an empty table with same name and schema).

***Data Manipulation Language*** subset of SQL is used to manipulate databases. Commands supported by DML include **Insert** (placing in new data), **Update** (modifying existing data), **Delete** (removing unwanted data) **and Select** (retrieving requested data). ***Select*** command is specifically called ***Data Query Language*** (DQL).

***Data Control Language*** subset of SQL is used to control accesses to data objects by various users. Commands supported by DCL includes **Grant** (to give a specific permission to a specific user on specific database objects, **Revoke** (to take back permissions given to users using grant command), **Commit** (to save changes made by users to database objects), **Rollback** (to undo the changes made by users), **Savepoint** (to mention a specific point during execution so that changes made starting from that point may be either committed or rolled back based on user choice). Commit, Rollback and Savepoint are specifically called ***Transaction Control Language*** (TCL).

***Triggers*** are procedures that are automatically invoked and executed by DBMS in response to user actions.

***Transaction Management*** is a part of SQL which deals with commands related to execution of transactions. These commands control concurrent execution of transactions.

***Dynamic and Embedded SQL*** is a part of SQL which deals with embedding SQL statements in host language syntax. Dynamic SQL deals with constructing SQL queries at run time.

***Remote Database Access*** deals with commands necessary to connect to a remote server to access a database.

***Basic SQL Query***:

The basic SQL query is of the following form:

**Select** [**Distinct**] select\_list (A comma separated list of columns)

**From** from\_list (A comma separated list of tables)

**Where** qualification; (A condition applied on individual records)

- “**Distinct**”: an optional keyword to remove duplicate records from final result.

***Evaluation of SQL Query***:

Step 1: The cross product of tables in the from\_list is computed i.e., the data present in all these tables is brought into a single table by concatenating every record of a table with every record of other table.

Step 2: The qualification in the where clause is applied on each record of the table formed at the end of step 1. Those records that do not qualify the condition are removed.

Step 3: Unwanted columns are eliminated from the result of step 2. Those columns that are mentioned in the select clause are only retained.

Step 4: Duplicate records are removed from the result of step 3 if required. (Distinct)

**Consider the following database**:

Sailors (**Sid** Number, Sname String, Rating Number, Age Number) Boats (**Bid** Number, Bname String, Colour String) Reserves (**Sid** Number, **Bid** Number, **Day** Date).

In terms of E-R model, *Sailors* and *Boats* are entity sets. “***Reserves***” is the binary relationship between them.

Each Sailor is uniquely identified by “***Sid***” and described by “Sname, Rating and Age”.

Each Boat is uniquely identified by “***Bid***” and described by “Bname and Colour”

A Sailor can reserve a boat on a day. The primary key of Reserves table is a composite key which include {***Sid, Bid, Day***} together. A Sailor can reserve same boat on different days.

***Queries***:

1) Find IDs of sailors who have reserved a Red boat.

**Select** Sid **from** Reserves, Boats **where** Colour = ‘Red’ and Boats.Bid = Reserves.Bid;

2) Find the names of sailors who have reserved a Red boat.

**Select** S.Sname **from** Sailor S, Boats B, Reserves R **where** Colour = ‘Red’ and S.Sid = R.Sid and B.Bid = R.Bid;

In this answer, using “S” for “Sailors”, “B” for “Boats”, “R” for “Reserves” is called aliasing. i.e., instead of referring to the table by its name, we refer to it using alias name in *where* and *select* clauses so that query writing becomes simple.

3) Find the IDs of sailors who have reserved at least one boat.

**Select** *distinct* Sid **from** Reserves;

\*\*\*When all rows are considered for the result, *where* clause is ***omitted***.

4) Find the IDs of sailors who have reserved at least two boats.

**Select** R1.Sid **from** Reserves R1, Reserves R2 **where** R1.Sid = R2.Sid and R1.Bid != R2.Bid;

5) Find IDs of sailors who have reserved either a Red boat or a Green boat.

**Select** *distinct* R.Sid **from** Reserves R, Boats B **where** R.Bid = B.Bid and B.Colour = ‘Red’ or B.Colour = ‘Green’;

6) Find IDs of sailors who have reserved both Red and Green colour boats.

Select R1.Sid from Reserves R1, Reserves R2, Boats B1, Boats B2 where R1.Sid=R2.Sid and R1.Bid=B1.Bid and R2.Bid=B2.Bid and R1.Colour=’Red’ and R2.Colour=’Green’;

***String and Pattern matching in SQL***: SQL supports pattern matching in strings. This feature is useful to retrieve data that satisfy a pattern. When the exact string is not known, pattern matching is useful. Following are few patterns that can be matched:

…where Sname ***like*** ‘A%’: Matches any string whose first character is “A”.

…where Sname ***like*** ‘A%B’: Matches any string whose first character is “A” and last character is “B”

…where Sname ***like*** ‘A\_\_: Matches any string whose first character is “A” and is 3 characters long.

…where Sname ***like*** ‘%A%’: Matches any string that contain the character “A”.

…where Sname ***like*** ‘\_\_\_%’: Matches any string that contains at least 3 characters.

7) Find the details of sailors whose names begin with a character “A”.

Select \* from Sailors where Sname like ‘A%’;

***Expressions in Select Clause***: SQL supports the use of expressions and built-in functions in *Select* clause along with columns from underlying base tables. Consider the table: Employees ( ***Emp\_ID***, Ename, DA, HRA, Hire\_Date) Consider the following query:

8) Display each Employee ID, total pay he receive and year of joining.

Select Emp\_ID, (DA+HRA) as TotalPay, To\_Char(Hire\_Date,’YYYY’) as YearofJoining from Employees;

**Date and Time Conversion Functions**: These functions are used to get different elements in a date and display the elements in different formats. Following are various conversions that are applicable on dates.

|  |  |
| --- | --- |
| 'YYYY' | Full year in Numbers |
| 'YEAR' | Year spelled out |
| 'MM' | Two digit value for month |
| 'MONTH' | Full name of the month |
| 'MON' | Three Letter abbreviation of the month |
| 'DY' | Three letter abbreviation of the day of the week |

9) Select To\_Char(DoB,'YEAR') from Emp where Deptno=20;

TO\_CHAR(DOB,'YEAR')

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

NINETEEN NINETY

NINETEEN NINETY-TWO

NINETEEN NINETY-THREE

NINETEEN NINETY-FOUR

10) Select To\_Char(DoB,'YYYY') from Emp where Deptno=20;

TO\_C

----

1990

1992

1993

1994

11) Select To\_Char(DoB,'MONTH') from Emp where Deptno=20;

TO\_CHAR(D

---------

OCTOBER

APRIL

FEBRUARY

SEPTEMBER

12) Select To\_Char(DoB,'DY') from Emp where Deptno=20;

TO\_

---

WED

THU

TUE

SUN

**--TO\_DATE** function is used to convert a character string to a DATE value. One advantage of the **TO\_DATE** function is that it allows user to specify a format for the value returned.

Create Table Date\_Convert(DoB varchar2(11), DB Date);

Table created.

Insert into date\_Convert values('12-oct-2021','12-Oct-2021');

1 row created.

Select To\_Char(DoB,'MM'), To\_Char(DB,'MM') from Date\_Convert

\*

ERROR at line 1:

ORA-01722: invalid number

12) Select To\_Char(To\_Date(DoB),'MM'), To\_Char(DB,'MM') from Date\_Convert;

TO TO

-- --

10 10

--CAST function is used to get time from a column of type “TimeStamp”. A column of type “TimeStamp” takes Date and Time together. (Ex: ’10-Aug-2020 04:52:36’). The syntax of CAST function is CAST(Col\_Name as Time).

13) Select Cast(BTime as Time) Time from Date\_Convert;

TIME

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

03.52.40 AM

**Number Conversion Functions**: These functions are used to convert string representation of a number to a numeric value.

To\_Number(): A string function that converts a string expression to a value of NUMERIC data type.

14) Select To\_Number(Expr) from Number\_Convert;

TO\_NUMBER(EXPR)

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

123.435

3241.14