GATE in Data Science and AI Study Materials
Database Management System and Warehousing
By Piyush Wairale

Instructions:

- Kindly go through the lectures/videos on our website www.piyushwairale.com
- Read this study material carefully and make your own handwritten short notes. (Short notes must not be more than 5-6 pages)
- Attempt the question available on portal.
- Revise this material at least 5 times and once you have prepared your short notes, then revise your short notes twice a week
- **If you are not able to understand any topic or required detailed explanation, please mention it in our discussion forum on webiste**
- **Let me know, if there are any typos or mistake in study materials. Mail me at piyushwairale100@gmail.com**

1 Introduction

A database is a collection of related data. By data, we mean facts that can be recorded and that have implicit meaning.
Example: Consider the names, telephone numbers and addresses of the people. We can record this data in an indexed address book and store it as an Excel file on a hard drive using a personal computer. This is a collection of related data with an implicit meaning and hence is a database.
A database management system (DBMS) is a collection of programs that enables users to create and maintain a database. The DBMS is a general-purpose software system that facilitates the processes of defining, constructing, manipulating and sharing databases among various users and applications.

- Defining the database involves specifying the data types, structures, and constraints for the data to be stored in the database.
- Constructing the database is the process of storing the data itself on some storage medium that is controlled by the DBMS
- Manipulating a database includes such functions as querying the database to retrieve specifi c data, updating the database to reflect changes.
- Sharing a database allows multiple users and programs to access the database concurrently.
- Fundamental characteristics of the database approach is that it provides some level of data abstraction by hiding details of data storage that are not needed by users.

Data Model
A data model is a collection of concepts that can be used to describe the structure of a database. It provides the necessary means to achieve abstraction.

Schemas

- In any data model, it is important to distinguish between the description of the database and the database itself. The description of a database is called the database schema, which is specified during database design and is not expected to change frequently.
- The actual data in a database may change frequently, for example the student database changes every time we add a student or enter a new grade for a student.
- The data in the database at a particular moment in time is called a database state or snapshot. It is also called the current set of occurrences or instances in the database.

- The distinction between database schema and database state is very important. When we define a new database, we specify its database schema only to the DBMS.
- At this point, the corresponding database state is the empty state with no data. We get the initial state of the database when the database is first loaded with the initial data. The DBMS stores the description of the schema constructs and constraints, also called the metadata in the DBMS catalog so that DBMS software can refer to the schema whenever it needs.
- The schema is sometimes called the intension, and the database state an extension of the schema.

2 Entity Relationship Diagram (ER Diagram)

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 database design.
The ER model is important primarily for its role in database design. It provides useful concepts that allow us to move from an informal description of what users want from their database to a more detailed, and precise, description that can be implemented in a DBMS. Within the larger context of the overall design process, the ER model is used in a phase called conceptual database design.

OVERVIEW OF DATABASE DESIGN

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

1. **Requirements Analysis:**
   The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on 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. This is usually an informal process that involves discussions with user groups, a study of the current operating environment and how it is expected to change, analysis of any available documentation on existing applications that are expected to be replaced or complemented by the database, and so on

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 that are known to hold over this data. This step is often carried out using the ER model, or a similar high-level data model.

3. **Logical Database Design:**
   We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model of the chosen DBMS. We will only consider relational DBMSs, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema.

4. **Schema Refinement:**
   The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems, and to refine it. In contrast to the requirements analysis and conceptual design steps, which are essentially subjective, schema refinement can be guided by some elegant and powerful theory.

5. **Physical Database Design:**
   In this step we must consider typical expected workloads that our database must support and further refine the database design to ensure that it meets the desired performance criteria. This step may simply involve building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema obtained from the earlier design steps.

6. **Security Design:**
   In this step, we identify different user groups and different roles played by various users (e.g., the development team for a product, the customer support representatives, the product manager). For each role and user group, we must identify the parts of the database that they must be able to access and the parts of the database that they should not be allowed to access, and take steps to ensure that they can access only the necessary parts.

ER Diagram:

Entity: An entity is a "thing" or "object" in the real world that is distinguishable from all other objects.
An entity is represented by a set of attributes.
An entity set is a set of entities of the same type that share the same properties, or attributes.
Relationship: A relationship is an association among several entities.
A relationship set is a set of relationships of the same type.

- An Entity Relationship Diagram (ERD) is a visual representation of different entities within a system and how they relate to each other.
- It is a tool used to design and model relational databases, and shows the logical structure of the database.
- ER diagrams use symbols to represent entities, attributes, and relationships, which help to illustrate the relationships between the entities in the database.
- ER diagrams are commonly used in software engineering and database design to help developers and stakeholders understand and design complex databases.

2.1 Symbols Used in ER Model:

Entities and Their Attributes:

- The basic concept that the ER model represents is an entity, which is a thing or object in the real world with an independent existence.
- An entity may be an object with a physical existence (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course).

- Each entity has attributes—the particular properties that describe it. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job.
- A particular entity will have a value for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.

Composite attributes:

- Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings.
- For example, the Address attribute of the EMPLOYEE entity can be subdivided into Street_address, City, State.
- Attributes that are not divisible are called simple or atomic attributes.

Single-Valued Attribute:

- A single-valued attribute is an attribute that can have only one value for a particular entity within a database.
- In other words, each entity instance possesses a single, unique value for a single-valued attribute.
- For example, the "Age" attribute of a person is single-valued because each person has a specific, singular age at any given time, and that age doesn't change while referring to the same person.
- Single-valued attributes are straightforward and do not allow for multiple values or variations.

Multivalued Attribute:

- A multivalued attribute is an attribute that can have multiple values for a single entity within a database. This means that an entity can have more than one value associated with the same attribute.
- For instance, the "Colors" attribute for a car is multivalued because a car can have one or more colors. A single-color car has only one value for the "Colors" attribute, whereas a two-tone car has two values for the same attribute. Similarly, the "College_degrees" attribute for a person is multivalued because different individuals may have varying numbers of college degrees, ranging from zero to multiple degrees.
- Multivalued attributes introduce complexity into the data model, and they can have constraints, such as minimum and maximum values, to limit the number of values allowed for each entity.

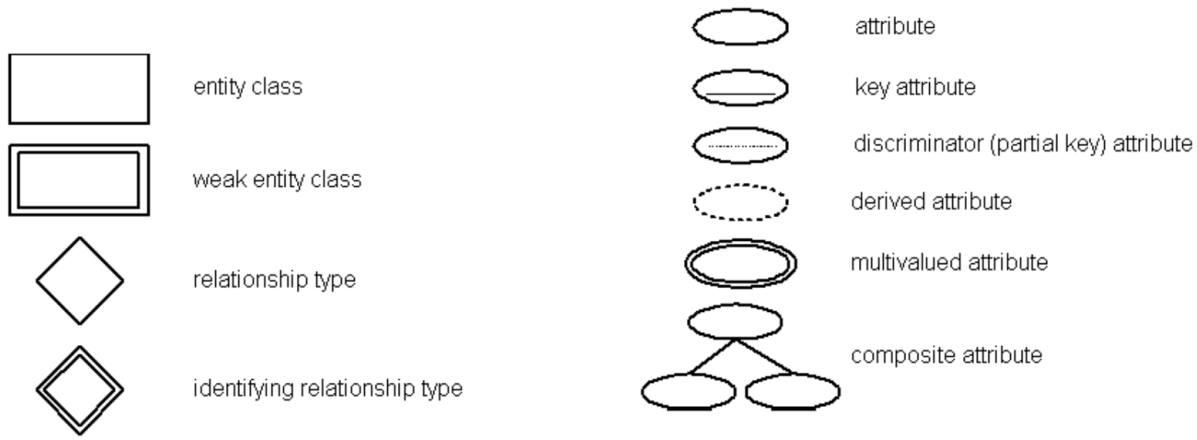
Stored Attribute:

- A stored attribute is an attribute whose value is directly and explicitly stored in the database. These values are typically entered or updated by users or processes and are maintained as part of the entity's data.
- Stored attributes are not computed or derived from other attributes or external sources. For example, in the context you provided, "Birth_date" is a stored attribute of a person because it represents an explicit value that is recorded in the database and is not calculated from other attributes or data.

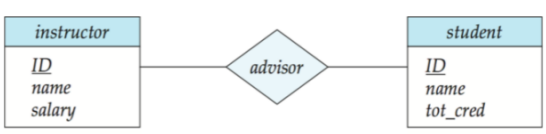
Derived Attribute:

- A derived attribute is an attribute whose value can be calculated or derived from other attributes or data within the database.
- The value of a derived attribute is not stored explicitly but is computed or derived when needed based on specific rules, algorithms, or formulas.
- In the example you mentioned, "Age" is a derived attribute of a person because it is calculated by subtracting the person's "Birth_date" from the current date.
- The "Age" attribute doesn't have a directly stored value; instead, it is derived from the stored "Birth_date" attribute and the current date when required.

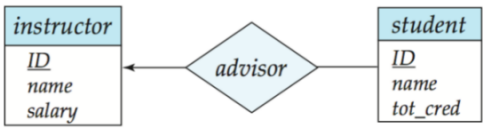
ER Diagram symbols



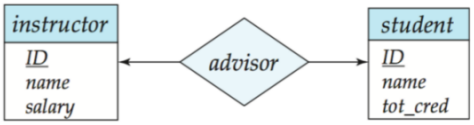
Mapping Constraints



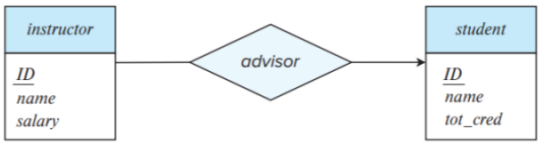
many-to-many relationship



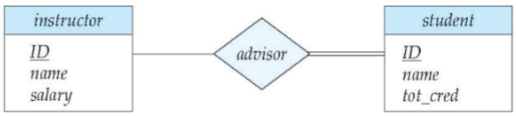
one-to-many relationship



one-to-one relationship



many-to-one relationship



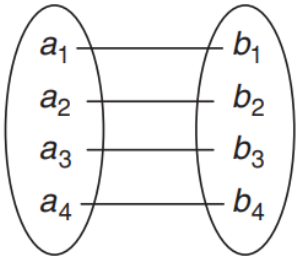
Total and Partial Participation

2.2 Cardinality Ratios:

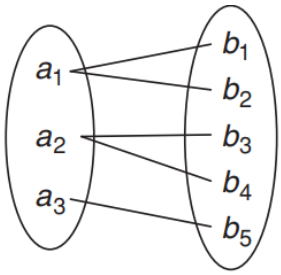
- Cardinality Ratio in the context of binary relationships in a database refer to the numerical constraints or limits that define how entities on both sides of the relationship can be related to each other.
- These ratios specify the maximum number of instances of one entity that can be associated with a single instance of another entity through the relationship.
- Cardinality ratios are crucial for modeling and understanding the relationships between entities in a database schema.

There are four common cardinality ratios for binary relationships:

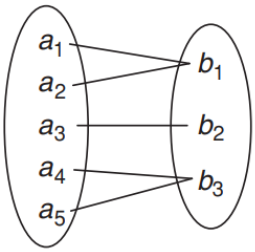
One-to-One (1:1):
In a 1:1 cardinality ratio, each entity instance on one side of the relationship is related to, at most, one entity instance on the other side. This means that for every instance of Entity A, there can be, at most, one related instance of Entity B, and vice versa.



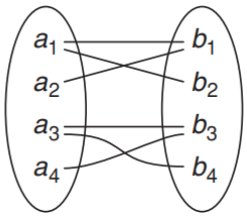
One-to-Many (1:N):
In a 1:N cardinality ratio, each entity instance on one side of the relationship can be related to multiple (N) entity instances on the other side. However, each instance on the other side can be related to, at most, one instance on the one side. This is also known as a "one-to-many" or "parent-child" relationship.



Many-to-One (N:1):
In an N:1 cardinality ratio, multiple entity instances on one side of the relationship can be related to one entity instance on the other side. However, each instance on the one side can be related to, at most, one instance on the other side.



Many-to-Many (M:N):
In an M:N cardinality ratio, multiple instances on one side of the relationship can be related to multiple instances on the other side. There is no limit to the number of relationships between entities on both sides. This often requires the use of an associative or junction table to represent the many-to-many relationship.

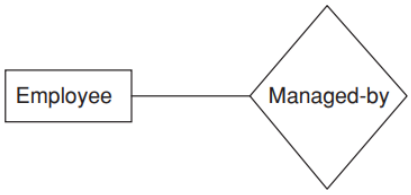


Total Participation:
Total Participation also known as Existence Dependency, is a participation constraint in a

binary relationship within a database schema that specifies that every entity on one side of the relationship must be associated with at least one entity on the other side of the relationship. In other words, it ensures that each entity in the total set of entities on one side of the relationship cannot exist independently and must have a relationship with an entity on the other side.

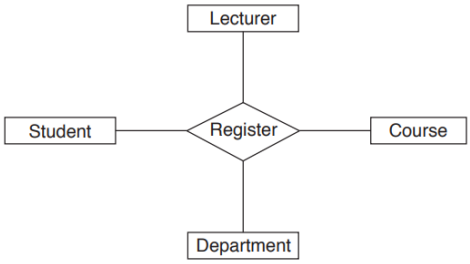
2.3 Types of Relations

1. Unary relation: If a relationship type is between entities in a single entity type then it is called a unary relationship type.

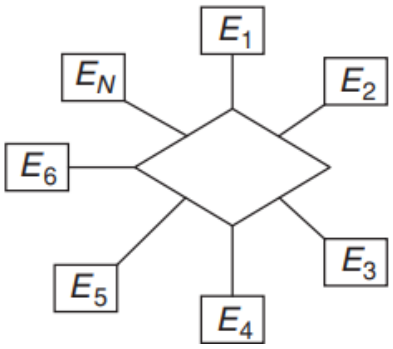


2. Binary relation: If a relationship type is between entities in one type and entities in another type then it is called a binary relation, because two entity types are involved in the relation.

3. Quadnary relation:If a relationship type is among entities of four different types, then it is called quadnary relation.



4. N-ary relation: 'N' number of entities will participate in a relation, and each entity can have a relation with all the other entities.



2.4 Translation of E-R Diagram into Relational Schema

1. **String Entity Set**

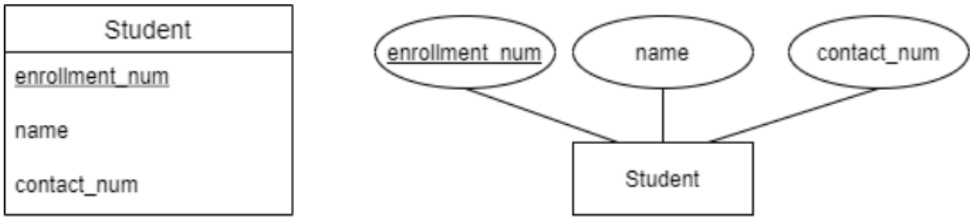


Figure: Strong entity with simple attributes

Student{*enrollment_num*, *name*, *contact_num*}

enrollment_num	name	contact_num
101	RAJ KUMAR MISHRA	222-222
102	SANAT K ROY	333-333

Figure: Table **Student**

2. **Strong Entity Set with Composite Key**

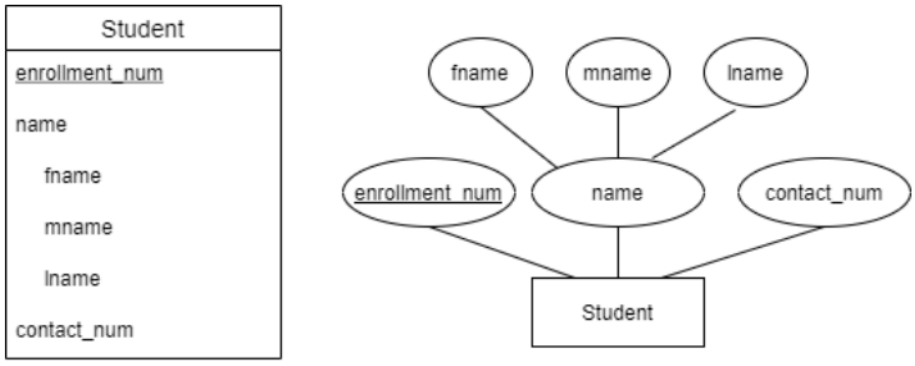


Figure: Entity set **Student** with simple and composite attributes

Student{*enrollment_num*, *fname*, *mname*, *lname*, *contact_num*}

enrollment_num	fname	mname	lname	contact_num
101	RAJ	KUMAR	MISHRA	222-222
102	SANAT	K	ROY	333-333

Figure: Table **Student**

3. **Strong Entity Set with Composite Key + Multivalued Attribute**

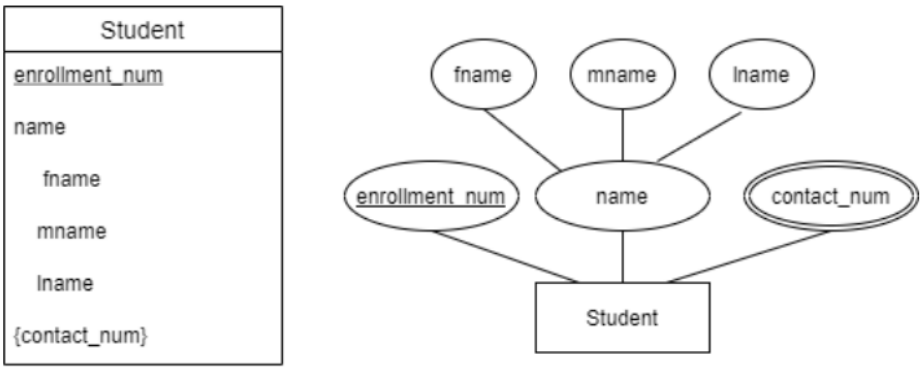


Figure: Entity set **Student** with simple, composite, and multivalued attributes

Student{*enrollment_num*, *fname*, *mname*, *lname*, *contact_num*}

enrollment_num	fname	mname	lname	contact_num
101	RAJ	KUMAR	MISHRA	222-222, 777-777
102	SANAT	K	ROY	333-333, 999-999, 666-666

Note: In order to convert the ER diagram having multivalued attribute, we need to create a separate relation to store a multivalued attribute, such that the key of parent table and that multivalued attribute form a composite key of the new relation.

Student{*enrollment_num*, *fname*, *mname*, *lname*, *contact_num*}

enrollment_num	fname	mname	lname	contact_num
101	RAJ	KUMAR	MISHRA	222-222
101	RAJ	KUMAR	MISHRA	777-777
102	SANAT	K	ROY	333-333
102	SANAT	K	ROY	999-999
102	SANAT	K	ROY	666-666

Figure: Table **Student**

Student{*enrollment_num*, *fname*, *mname*, *lname* }

enrollment_num	fname	mname	lname
101	RAJ	KUMAR	MISHRA
102	SANAT	K	ROY

Figure: Table **Student**

Contacts{*enrollment_num*, *contact_num*}

enrollment_num	contact_num
101	222-222
101	777-777
102	333-333
102	999-999
102	666-666

Figure: Table **Student**

4. **Strong Entity Set with Composite Key + Multivalued Attribute + Derived Attribute**

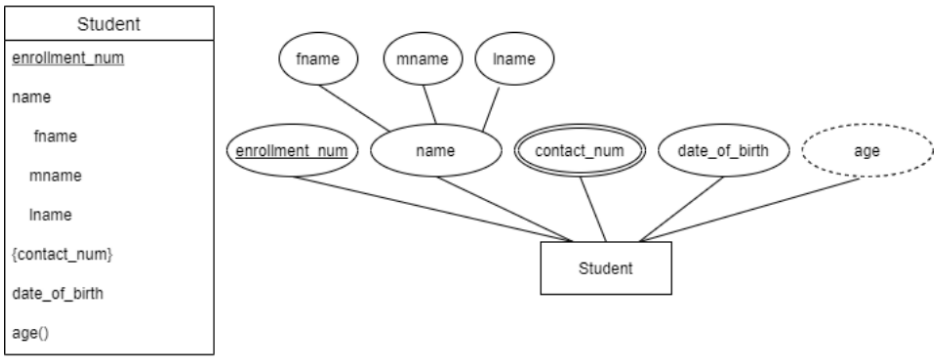


Figure: Entity set **Student** with simple, composite, multivalued attributes, and derived attribute

Student{*enrollment_num*, *fname*, *mname*, *lname*, *data_of_birth* }
Contacts{*enrollment_num*, *contact_num*}

5. Relationship: Cardinality Constraint (many-to-many)

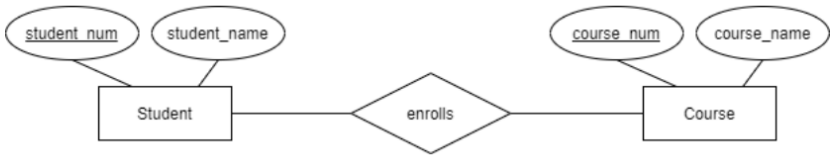


Figure: A many-to-many relationship between **Student** and **Course**

- **Student**{*student_num*, *student_name*}
- **Course**{*course_num*, *course_name*}
- **enrolls**{ *student_num*, *course_num* }

student_num	student_name
101	RAJ
102	SANAT

Student

student_num	course_num
101	CS101
102	CS101
102	MT110

enrolls

course_num	course_name
CS101	Computer Science
MT110	Mathematics

Course

Figure: Table: **Student**, **Course** and **enrolls**

6. Relationship: Cardinality Constraint (many-to-many) with Descriptive Attributes

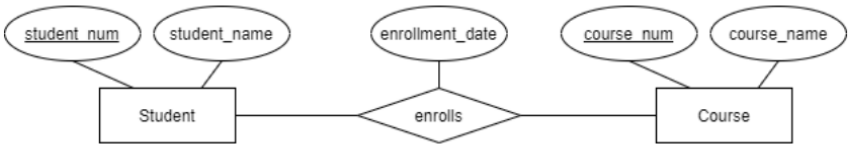


Figure: A many-to-many relationship between **Student** and **Course**

- **Student**{*student_num*, *student_name*}
- **Course**{*course_num*, *course_name*}
- **enrolls**{ *student_num*, *course_num*, *enrollment_date* }

7. Relationship: Cardinality Constraint (many-to-one)

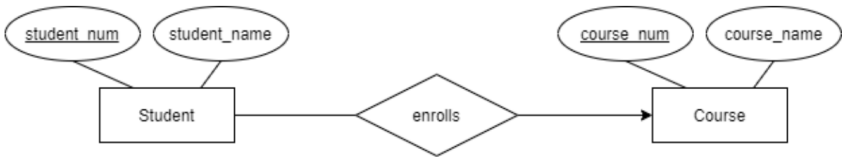


Figure: An many-to-one relationship between **Student** and **Course**

- **Student**{*student_num*, *student_name*, *course_num*}
- **Course**{*course_num*, *course_name*}

8. Relationship: Cardinality Constraint (one-to-one)

Figure: An one-to-one relationship between **Department** and **Manager**

- **Department**{*dept_num*, *dept_name*}
- **Manager**{*mgr_num*, *mgr_name*, *dept_num*}

OR

- **Department**{*dept_num*, *dept_name*, *mgr_num*}
- **Manager**{*mgr_num*, *mgr_name* }

9. Relationship: Cardinality Constraint (many-to-one) with Participation Constraint

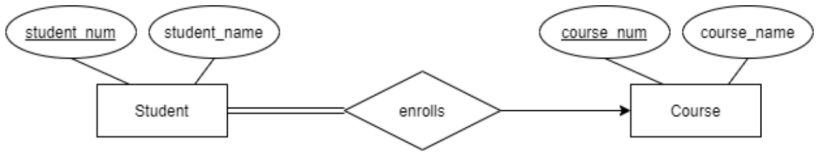


Figure: An many-to-one relationship between **Student** and **Course**

- **Student**{*student_num*, *student_name*, *course_num*}
- **Course**{*course_num*, *course_name*}

10. **Relationship: Cardinality Constraint (one-to-one) with Participation Constraint**

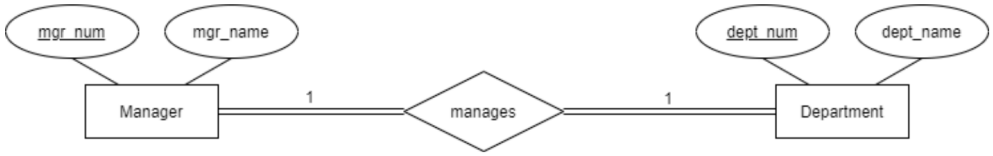


Figure: A one-to-one relationship between **Manager** and **Department**

- **Mgr_Dept**{*mgr_num*, *dept_num*, *mgr_name*, *dept_name* }

11. **Weak Entity Set**

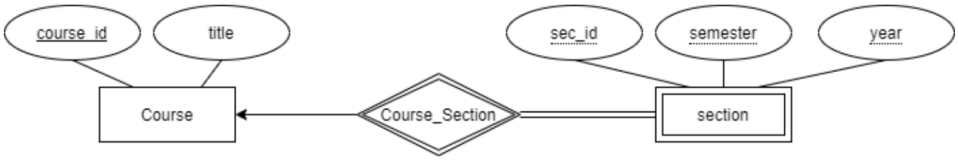


Figure: A one-to-one relationship between **Course** and **Section**

- **Course**{*course_id*, *title* }
- **Section**{*course_id*, *sec_id*, *semester*, *year* }

12. **Ternary Relationship**

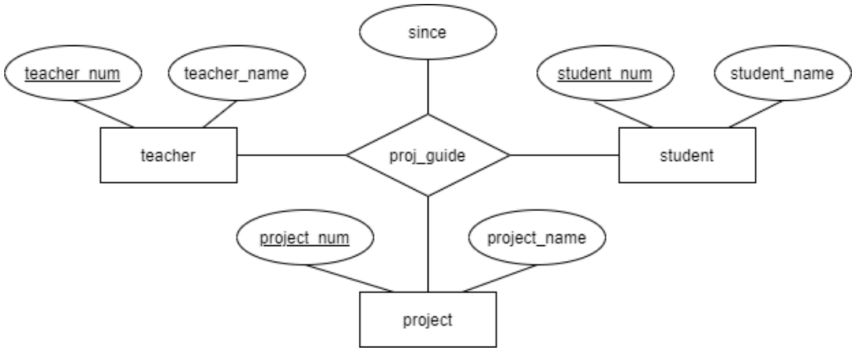


Figure: Example of ternary relationship

- **teacher**{*teacher_num*, *teacher_name* }
- **student**{*student_num*, *student_name* }
- **project**{*project_num*, *project_name* }
- **proj_guide**{*teacher_num*, *student_num*, *project_num*, *since* }

3 Relational Algebra

- Relational algebra refers to a procedural query language that takes relation instances as input and returns relation instances as output.
- It performs queries with the help of operators.
- It gives a step by step process to obtain the result of the query.
- **Select Operator (σ):**
  Select operator is an unary operator. It can be used to select those tuples of a relation that satisfy a given condition.
  Notation: $\sigma_\theta(R)$
  σ: Select operator(read as sigma)
  θ: Selection condition
  R: Relation name
  Result is a relation with the same scheme as R consisting of the tuples in R that satisfy condition θ

  Syntax: $\sigma_{condition}(relation)$

- **Project Operator** Π
  The projection operator Π is one of the unary operators in relational algebra (RA) and is used to project columns from a relation. It can select specific columns from a given relation.
  Note: It gives **distinct** fields only.

  Syntax: $\Pi_{A_1,A_2}(relation)$, this will return attribute A_1 and A_2 from the relation.

- **Rename Operator (ρ):**
  The RENAME operation is used to rename the output of a relation.
  Sometimes it is simple and suitable to break a complicated sequence of operations and rename it as a relation with different names.
  Syntax: $\rho_X(R)$, it will rename the relation R to X

  Do watch the video for better understanding

- **Set Operators:** Union ($\cup$), Intersection ($\cap$), set difference ($-$) are called set operators. Result of combining two relations with a set operator is a relation $\implies$ all its elements must be tuples having same structure. Hence scope of set operations is limited to union-compatible relations.

Union Compatible Relations

Two relations are union compatible if

1. Both have same number of columns
2. Names of attributes are same
3. Corresponding fields have same type
4. Attributes with the same name in both relations have same domain and datatype.

4 Relational Calculus

Relational calculus is an alternative to relational algebra. In contrast to algebra, which is procedural, calculus is nonprocedural, or declarative, in that it allows us to describe the set of answers without being explicit about how they should be computed.
The variant of the calculus that we present in detail is called the tuple relational calculus (TRC). Variables in TRC take on tuples as values. In another variant, called the domain relational calculus (DRC), the variables range over field values.

4.1 Tuple Relational Calculus

1. A tuple variable is a variable that takes on tuples of a particular relation schema as values. That is, every value assigned to a given tuple variable has the same number and type of fields.

2. A tuple relational calculus query has the form $\{T|p(T)\}$, where T is a tuple variable and p(T) denotes a formula that describes T; we will shortly define formulas and queries rigorously.

3. The result of this query is the set of all tuples t for which the formula p(T) evaluates to true with T = t.

4. The language for writing formulas p(T) is thus at the heart of TRC and is essentially a simple subset of first-order logic.

As a simple example, consider the following query.
Find all sailors with a rating above 7.
$\{S|S \in Sailors \wedge S.rating > 7\}$
When this query is evaluated on an instance of the Sailors relation, the tuple variable S is instantiated successively with each tuple, and the test $S.rating > 7$ is applied.

TRC is a nonprocedural query language, where each query is of the form

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

where t = resulting tuples,
P(t) = known as predicate and these are the conditions that are used to fetch t.
P(t) may have various conditions logically combined with OR ($\vee$), AND ($\wedge$), NOT($\neg$).

It also uses quantifiers:
$\exists t \in r(Q(t))$ = "there exists" a tuple in t in relation r such that predicate Q(t) is true.
$\forall t \in r(Q(t))$ = Q(t) is true "for all" tuples in relation r.

- $\{P \mid \exists S \in Students(S.CGPA > 8 \wedge P.name = S.sname \wedge P.age = S.age)\}$: returns the name and age of students with a CGPA above 8.

4.2 Domain Relational Calculus

$\{< x_1, x_2, \ldots, x_n > | P(x_1, x_2, \ldots, x_n)\}$

- $x_1, x_2, \ldots, x_n$ represent domain variables
- P represents a formula similar to that of the predicate calculus

Name	Age	Marks	Subject
David	23	78	Maths
Matthew	29	54	English
Anand	22	89	JAVA
Mitchel	21	56	Maths
Shaun	26	92	Maths

Relation **Students**

- $\{< a, b > | \exists a, b, c, d(< a, b, c, d > \in students \wedge c > 75)\}$: returns the name and age of students having marks above 75.

Note: You have to mention the domain variables in the same order as given in the table.

4.3 Examples

Name	Age	Marks	Subject
David	23	78	Maths
Matthew	29	54	English
Anand	22	89	JAVA
Mitchel	21	56	Maths
Shaun	26	92	Maths

Relation **Students**

Name	Sports	Awards	Points
David	Cricket	2	67
Matthew	Football	4	90
Anand	Cricket	5	80
Mitchel	Tennis	8	70
Shaun	Hockey	3	75

Relation **Activity**

Q). Write down the RA, TRC and DRC expressions which will return the names of students whose age is greater than 25, or who are enrolled in Maths.
RA: $\Pi_{Name}(\sigma_{age>25 \vee subject="Maths"}(Students))$
TRC: $\{t \mid \exists s \in students(s.age > 25 \vee s.subject = "Maths" \wedge t.name = s.name)\}$

$\{t.name \mid t \in students \wedge s.age > 25 \vee s.subject = "Maths")\}$
DRC: $\{< a > \mid \exists b, c, d(< a, b, c, d > \in students \wedge b > 25 \vee d = "Maths")\}$

Q). Write down the RA, TRC and DRC expressions which will return the names of students along with sports, whose age is less than 25 and who have secured more than 75 marks.

RA: $\Pi_{Name,Sports,}(\sigma_{age<25 \wedge Marks>75}(Students \bowtie Activity))$
TRC: $\{t \mid \exists s \in students\ \exists a \in activity(s.name = a.name \wedge s.age < 25 \wedge s.marks > 75 \wedge t.name = s.name \wedge t.sports = a.sports)\}$
DRC: $\{< a, f > \mid \exists b, c, d(< a, b, c, d > \in students \wedge b < 25 \wedge c > 75) \wedge \exists e, g, h(< e, f, g, h > \in activity \wedge a = e)\}$

5 Functional Dependency and Decomposition

Functional Dependency (FD):
A functional dependency (FD) is a kind of integrity constraints that generalizes the concept of a key. Let R be a relation schema and let X and Y be nonempty sets of attributes in R. We say that an instance r of R satisfies the FD $X \rightarrow Y$. If the following holds for every pair of tuples t_1 and t_2 in r.
If $t1.X = t2.X$, then $t1.Y = t2.Y$
$X \rightarrow Y$ is read as functionally determines Y or simply X determines Y.
An FD $X \rightarrow Y$ essentially says that if two tuples agree on the values in attribute X, they must also agree on the values in attributes Y.
If a constraints on R states that there cannot be more than one tuple with a given X value in any relation instance $r(R)$, that is X is the key of R, the definition of an FD does not require that the set X be minimal, the additional minimality condition must be met for X to be a key.

If $X \rightarrow Y$ holds, where Y is a set of all attribute and there is some subset V of X such

that $V \rightarrow Y$ holds then X is a super key.

Trivial FD: If X and Y are attribute set of R and $Y \subseteq X$ then $X \rightarrow Y$ is trivial FD.
Example: Sid Sname $\rightarrow$ sid
Every trivial FD implies in relation.

Non-trivial FD: If X and Y are attribute sets of R and no common attribute between X and Y.
$X \cap Y \neq$ empty set then $X \rightarrow Y$ is non- trivial FD.

Example: sid $\rightarrow$ gpa
Sname $\rightarrow$ sid gpa
There may be a relation with no non-trivial FD.

Closure of set of FDs
Set of all FDs that include given FDs as well as those that can be inferred from the given. FDs is called the closure of FDs. If F is the set of given FDs the F^+ is called closure of F.

The following three rules, called Armstrong's Axioms can be applied repeatedly to infer all FDs implied by a set F of FDs. Let X, Y and Z denotes sets of attributes over a relation schema R.

- **Reflexivity:** If $X \supseteq Y$, then $X \rightarrow Y$.
- **Augmentation:** If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z.
- **Transitivity:** If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Armstrong's Axioms are **sound** in that they generate only FDs in F^+ when applied to a set F of FDs. They are **complete** in that repeated application of these rules will generate all FDs in the closure F^+. (We will not prove these claims.) It is convenient to use some additional rules while reasoning about F^+:

- **Union:** If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.
- **Decomposition:** If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$.

These additional rules are not essential; their soundness can be proved using Armstrong's Axioms.

5.1 Lossless Decomposition

Lossless join decomposition is a decomposition of a relation R into relations R1, and R2 such that if we perform a natural join of relation R1 and R2, it will return the original relation R. This is effective in removing redundancy from databases while preserving the original data.

In other words by lossless decomposition, it becomes feasible to reconstruct the relation R from decomposed tables R1 and R2 by using Joins.

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r)$$

- A decomposition of R into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies

To Identify whether a decomposition is lossy or lossless, it must satisfy the following conditions:

- $R_1 \cup R_2 = R$
- $R_1 \cap R_2 \neq \phi$ and
- $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
  - Can be decomposed in two different ways
- $R_1 = (A, B), R_2 = (B, C)$
  - Lossless-join decomposition:
    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
  - Dependency preserving
- $R_1 = (A, B), R_2 = (A, C)$
  - Lossless-join decomposition:
    $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
  - Not dependency preserving
    (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

- Let F_i be the set of dependencies F^+ that include only attributes in R_i
  - A decomposition is **dependency preserving**, if

$$(F_1 \cup F_2 \cup \cdots \cup F_n)^+ = F^+$$

  - If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive

Let R be the original relational schema having FD set F. Let R_1 and R_2 having FD set F_1 and F_2 respectively, are the decomposed sub-relations of R. The decomposition of R is said to be preserving if

- $F_1 \cup F_2 \equiv F$ {Decomposition Preserving Dependency}
- If $F_1 \cup F_2 \subset F$ {Decomposition NOT Preserving Dependency} and
- $F_1 \cup F_2 \supset F$ {this is not possible}

GATE CSE 21 Question

Consider the relation $R(P, Q, S, T, X, Y, Z, W)$ with the following functional dependencies $PQ \rightarrow X; P \rightarrow YX; Q \rightarrow Y; Y \rightarrow ZW$

Consider the decomposition of the relation R into the constituent relations according to the following two decomposition schemes

D1: $R = [(P, Q, S, T); (P, T, X); (Q, Y); (Y, Z, W)]$

D2: $R = [(P, Q, S); (T, X); (Q, Y); (Y, Z, W)]$

Which one of the following options is correct?

1. D1 is a lossless decomposition, but D2 is a lossy decomposition.
2. D1 is a lossy decomposition, but D2 is a lossless decomposition.
3. Both D1 and D2 are lossless decompositions.
4. Both D1 and D2 are lossy decompositions.

Solution Option 1

Watch the tutorial for the solution

Advantages of Lossless Decomposition

- Reduced Data Redundancy: Lossless decomposition helps in reducing the data redundancy that exists in the original relation. This helps in improving the efficiency of the database system by reducing storage requirements and improving query performance.
- Maintenance and Updates: Lossless decomposition makes it easier to maintain and update the database since it allows for more granular control over the data.
- Improved Data Integrity: Decomposing a relation into smaller relations can help to improve data integrity by ensuring that each relation contains only data that is relevant to that relation. This can help to reduce data inconsistencies and errors.
- Improved Flexibility: Lossless decomposition can improve the flexibility of the database system by allowing for easier modification of the schema.

Disadvantages of Lossless Decomposition

- Increased Complexity: Lossless decomposition can increase the complexity of the database system, making it harder to understand and manage.
- Increased Processing Overhead: The process of decomposing a relation into smaller relations can result in increased processing overhead. This can lead to slower query performance and reduced efficiency.
- Join Operations: Lossless decomposition may require additional join operations to retrieve data from the decomposed relations. This can also result in slower query performance.
- Costly: Decomposing relations can be costly, especially if the database is large and complex. This can require additional resources, such as hardware and personnel.

6 Normalization and Normal Forms

Database design theory includes design standards called normal forms. The process of making data and tables match these standards is called normalizing data or data normalization. By normalizing data, we eliminate redundant information and organize the table to make it easier to manage the data and make future changes to the table and database structure. This process removes the insertion, deletion, and modification anomalies. In normalizing your data, we usually divide large tables into smaller, easier-to-maintain tables. We can then use the technique of adding foreign keys to enable connections between the tables.
Data normalization is part of the database design process and is neither specific nor unique to any particular RDBMS. These are in order, such as first, second, third, Boyce-Codd, fourth, and fifth normal forms. Each normal form represents an increasingly stringent set of rules; that is, each normal form assumes that the requirements of the preceding forms have been met. Many relational database designers feel that, if their tables are in third normal form, most common design problems have been addressed.

- Database normalization is the process of removing redundant data from tables to improve storage efficiency, data integrity and scalability.
- Normalization generally involves splitting existing tables into multiple ones, which must be rejoined (or) linked each time a query is issued.
- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
- Normalization divides the larger table into smaller and links them using relationships.

Anomalies
An anomaly is a variation that differs in some way from what is said to be normal, with respect to maintaining a database.

- The basic operations performed on Databases are Record insertion, Record updation, and Record deletion.
- It is desirable for these operations to be straight forward and efficient.
- When relations are not fully normalized they exhibit anomalies.
- The design goal of database is too easily to understand and to maintain.
- Anomalies are problems that occur in un-normalized databases where all the data is stored in one table.

Types of anomalies

There are three types of anomalies that can arise in the database because of redundancy as follows:

1. **Insertion anomaly:** An insertion anomaly occurs when particular attributes cannot be inserted into the database without the presence of other attributes
2. **Deletion anomaly:** Deletion anomaly occurs when some particular attributes are lost because of the deletion of other attributes.
3. **Updation anomaly:** An updation anomaly occurs when one or more instances of duplicated data are updated but not all.

Normal Forms

1. First Normal Form (1 NF):
   No multi valued attributes, it must have atomic domain only.
2. Second Normal Form (2 NF):
   - Second normal form (2NF) is based on the concept of fully functional dependency. A functional dependency $X \to Y$ is a full function dependency if the removal of any attribute A from X means that the dependency does not hold anymore; i.e., for any attribute $A \in X$, $(X - \{A\})$ does not functionally determine Y.
   - A functional dependency $X \to Y$ is a partial dependency if some attribute A can be removed from X and the dependency still holds; i.e., for some $A \in X, (X - \{A\}) \to Y$.
   - The test for 2NF involves testing for functional dependency whose left-hand side attributes are part of the primary key and the right-hand side is a non-prime attribute. If the primary key contains a single attribute the test need not be applied at all.
     Xis any candidate key
     Y: Proper subset of key
     A: None prime attribute
     Then we can say, $Y \to A$ is partial dependency.
3. Third Normal Form (3NF):

   Let R be a relation schema, X be a subset of the attributes of R, and A be an attribute of R. R is in third normal form if for every FD $X \to A$ that holds over R, one of the following statements is true:
   - $A \in X$; that is, it is a trivial FD, or
   - X is a superkey, or
   - A is part of some key for R.

If Non-Prime Attribute determines non prime attribute then we can say there is a transistive dependency

4. Boyce Codd's normal Form (BCNF)
   - Relation shloud be in 3NF first
   - $A \in X$; that is, it is a trivial FD, or
   - X is a superkey, or
5. Forth Normal Form (4NF)
   - Relation should be in BCNF
   - No **Multi Valued Dependency**
6. Fifth Normal Form (5NF):
   It deals with join dependency

A relation schema R is said to be in **fifth normal form (5NF)** if for every JD $\bowtie \{R_1, \ldots, R_n\}$ that holds over R, one of the following statements is true:

- $R_i = R$ for some i, or
- The JD is implied by the set of those FDs over R in which the left side is a key for R.

Advantages of Normalization:

- Normalization helps to minimize data redundancy.
- Greater overall database organization.
- Data consistency within the database.
- Much more flexible database design.
- Enforces the concept of relational integrity.

Disadvantages of Normalization:

- You cannot start building the database before knowing what the user needs.
- The performance degrades when normalizing the relations to higher normal forms, i.e., 4NF, 5NF.
- It is very time-consuming and difficult to normalize relations of a higher degree.
- Careless decomposition may lead to a bad database design, leading to serious problems.