UNIT-II

Logical Database Design: Relational DBMS - Codd's Rule - Entity-Relationship model - Extended ER Normalization - Functional Dependencies - Anomaly - 1NF to 5NF - Domain Key Normal Form–Denormalization.

Course Outcome: Implement storage of data

Relational DBMS

Structure of Relational Database: A relational database consists of a collection of *tables*. Each table is assigned a unique name. A row in a table represents a *relationship* among a set of values. For basic structure, consider the *deposit* table of the followingfigure:

Branch-Name	Account-Number	Customer-Name	Balance
A	101	Dileep	2500
B	215	Kapil	6200
C	305	Ravi	3400
D	217	Raghu	7400
E	786	Bharath	4000

Table: The *deposit* relation

Domain: A set of permitted values of each attribute of a table is called as domain. The above table has four attributes: *Branch Name, Account Number, Customer Name*, and *Balance*. For each attribute, there is a set of permitted values, called the Domain of that attribute.

For example, for the attribute *Branch Name,* the *domain* is the set of all branch names. Similarly for the

Balance, the domain is the all balance values.

Relation: A relation is subset of a Cartesian product of a list of domains.

Ex: In the above relation Deposit, there are four attributes. D1 is the domain of branch Names, D2 is the set of all account numbers, D3 is the set of all customer names and D4 is the set of all balance values.

Every row in a deposit relation consists of 4-tuples (v1, v2, v3, v4) where v1 is the branch Name (i.e., v1 is in domain D1), v2 is an account number (i.e., v2 is in domain D2), v3 is the customer name (i.e., v3 is in domain D3) and v4 is the balance (i.e., v4 is in domain D4). In general, deposit will contain only a subset of all possible rows.

i.e., Deposit is a subset of D1 X D2 X D3 X D4

TUPLE: Row of a given flat file (table) is called a tuple of the relation.

Ex: In the above relation, there are five tuples (rows), i.e., the given relation cardinality is five.

Degree of a Relation: The number of the attributes in a given relation is called degree of the relation.

Ex: The degree of the given relation is 4.

KEYS IN A RELATIONAL DATABASE:

KEY: A data item (attribute) is used to identify or locate a record is called a key (entity identifier).
VARIOUS KEYS IN A RELATIONAL DATABASE

Primary key 2) Candidate key 3) Alternate key 4) Secondary key 5) Super key 6) Foreign key

1) **PRIMARY KEY:** The primary key is defined as a data item (attribute) which uniquely identifies a record (one row) in a relation.

Roll-No	Name	Date-of-Birth	Second-Language	Division
95	Swetha	31st Dec	Hindi	First
96	Dhatri	26th Jan	Sanskrit	Second
97	Shivani	15th Aug	Telugu	First
98	Kavya	01st Nov	Hindi	Second
99	Jagruti	29th Feb	Sanskrit	First
100	Shravani	26th Jan	Telugu	First

The *student* Relation

In the above relation *student*, we can choose roll number attribute as primary key because for each row, there is a unique value of Roll number attribute i.e., same roll number is not repeated in another rows.

Therefore *Roll Number* is primary key for given relation.

1.CANDIDATE KEY:

In a relation in which there is more than one attribute combination possessing the unique identification property, all the various combinations of attributes, which serve as a primary key are called the candidate keys of the given relation.

Note: Subset of Candidate keys is not a primary Key. Ex:-In the given relation student:

{Roll number} can identify each row uniquely. So it is one candidate key.
{Second Language, Date of Birth} can identify each row uniquely; it is one more candidate key.
{Date of Birth, Division} can identify each row uniquely so it is one more candidate key.
{Roll Number, Date of Birth} can identify each row uniquely. But subset of this key is {Roll Number}

{Date of Birth}.
So Here {Roll Number] is a primary key. As per definition, subset of candidate keys not a primary key. So {Roll Number, Date of Birth} is not a candidate key.

2) ALTERNATE KEY:

A candidate key that is not the primary key, called as alternate key. Ex: In a given relation student{Roll number} is unique for every student. Similarly all student names are unique (no students having the same name). Here we can choose either Roll number or name as a primary key.In such a case we may arbitrarily choose one of the candidates. Say *Roll Number* as the primary key for the relation. A candidate key that is not the primary key, such *Name* in the relation student is

called alternate key.i.e., if Roll number is primary key, then Name is alternate key. If Name is primary key, then Roll Number is alternate key.

3) **SECONDARY KEY:** System may also use a key which does not identify a unique record or tuple but which identifies all those which have certain property. This is referred to as a Secondary Key.

Ex: In a given relation STUDENT the value of the attribute "second language" may be used as a secondary key. This key could be used to identify those entities (students), who belong to second language Hindi, Telugu or Sanskrit.

Second-Language	Roll-No
Hindi	95
	98
Sanskrit	96
	99
Telugu	97
	100

Secondary Keys

4) **SUPER KEY:** More than one attribute combined together for unique identification of the record is defined as a Super Key As shown in figure below, neither supplier no. (S#), nor product no. (P#) are enough to identify the each row. To get unique information for each row, we need combined attributes s#, p#. i.e. {s# + p#} is a Super key (or) concatenate key.

S#	P#	Qty
S1	P1	500
S1	P2	700
S1	P3	450
S2	P4	920
S3	P1	650
S3	P5	400

Super Keys

5) **FOREIGN KEY:** A foreign key is an attribute or group of fields in a database record that points to a key field or group of fields forming a key of another database record in a different table. Usually a foreign key in one table refers to the primary key of another table. This way references can be made to link. For Ex: For an Account relation Customer_ID is considered as the foreign key.

Codd's Rule

Dr Edgar F. Codd, after his extensive research on the Relational Model of database systems, came up with twelve rules of his own, which according to him, a database must obey in order to be regarded as a true relational database.

These rules can be applied on any database system that manages stored data using only its relational capabilities. This is a foundation rule, which acts as a base for all the other rules.

Rule 1: Information Rule

The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

Rule 2: Guaranteed Access Rule

Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data.

Rule 3: Systematic Treatment of NULL Values

The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following − data is missing, data is not known, or data is not applicable.

Rule 4: Active Online Catalog

The structure description of the entire database must be stored in an online catalog, known as **data dictionary**, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

Rule 5: Comprehensive Data Sub-Language Rule

A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

Rule 6: View Updating Rule

All the views of a database, which can theoretically be updated, must also be updatable by the system.

Rule 7: High-Level Insert, Update, and Delete Rule

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.

Rule 8: Physical Data Independence

The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.

Rule 9: Logical Data Independence

The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rule to apply.

Rule 10: Integrity Independence

A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.

Rule 11: Distribution Independence

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

Rule 12: Non-Subversion Rule

If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

Entity Relationship model

Database systems are designed to manage large bodies of information. Database design mainly involves the design of the database schema. The design of a complete database application environment that meets the needs of the enterprise being modeled requires attention to a broader set of issues.

Design Process: A high-level data model provides the database designer with a conceptual framework in which to specify the data requirements of the database users, and how the database will be structured to fulfill these requirements. The initial phase of database design is to characterize fully the data needs of database users. The outcome of this phase is a specification of user requirements.

- ✓ Next, the designer chooses a data model, and by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database.
- ✓ The schema developed at this conceptual-design phase provides a detailed overview of the enterprise. The designer reviews the schema to confirm that all data requirements are indeed satisfied and are not in conflict with one another.

  The designer can also examine the design to remove any redundant features. The focus at this point is on describing the data and their relationships, rather than on specifying physical storage details.

- ❖ In terms of the relational model, the conceptual-design process involves decisions on what attributes we want to capture in the database and how to group these attributes to form the various tables. The "what" part is basically a business decision and the "how" part is mainly a computer-science problem. There are principally two ways to tackle the problem.
- ❖ First one is to use ***entity-relationship model***; the other is to employ a set of algorithms collectively known as ***normalization*** that takes as input the set of all attributes and generates a set of tables.
- ❖ The process of moving from an abstract data model to the implementation of the database

proceeds in two final design phases. In the **logical-design phase**, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The designer uses the resulting system-specific database schema in the subsequent **physical-design phase**, in which the physical features of the database are specified.

Database Design for a University Organization:

The description that arises from this design phase serves as the basis for specifying the conceptual structure of the database. Here are the major characteristics of the university.

- ✓ University is organized into departments. Each department is identified by a unique name (dept_name).
- ✓ Each department has a list of courses it offers. Each course has associated with it a course_id, title, dept_name, and credits, and may also have have associatedprerequisites.
- ✓ Instructors are identified by their unique ID. Each instructor has name, associated department (dept_name), andsalary.
- ✓ Students are identified by their unique ID. Each student has a name, an associated major department (dept_name), and tot_cred (total credit hours the student earned thusfar).
- ✓ The university maintains a list of classrooms, specifying the name of the building, room number, and roomcapacity.
- ✓ The university maintains a list of all classes (sections) taught. Each section is identified by a course_id, sec_id, year, and semester, and has associated with it a semester, year, building, room number, andtime_slot_id.
- ✓ The university has a list of all student course registrations, specifying, for each student, the courses and the associated sections that the student has taken (registeredfor).

DATABASE DESIGN FOR UNIVERSITY

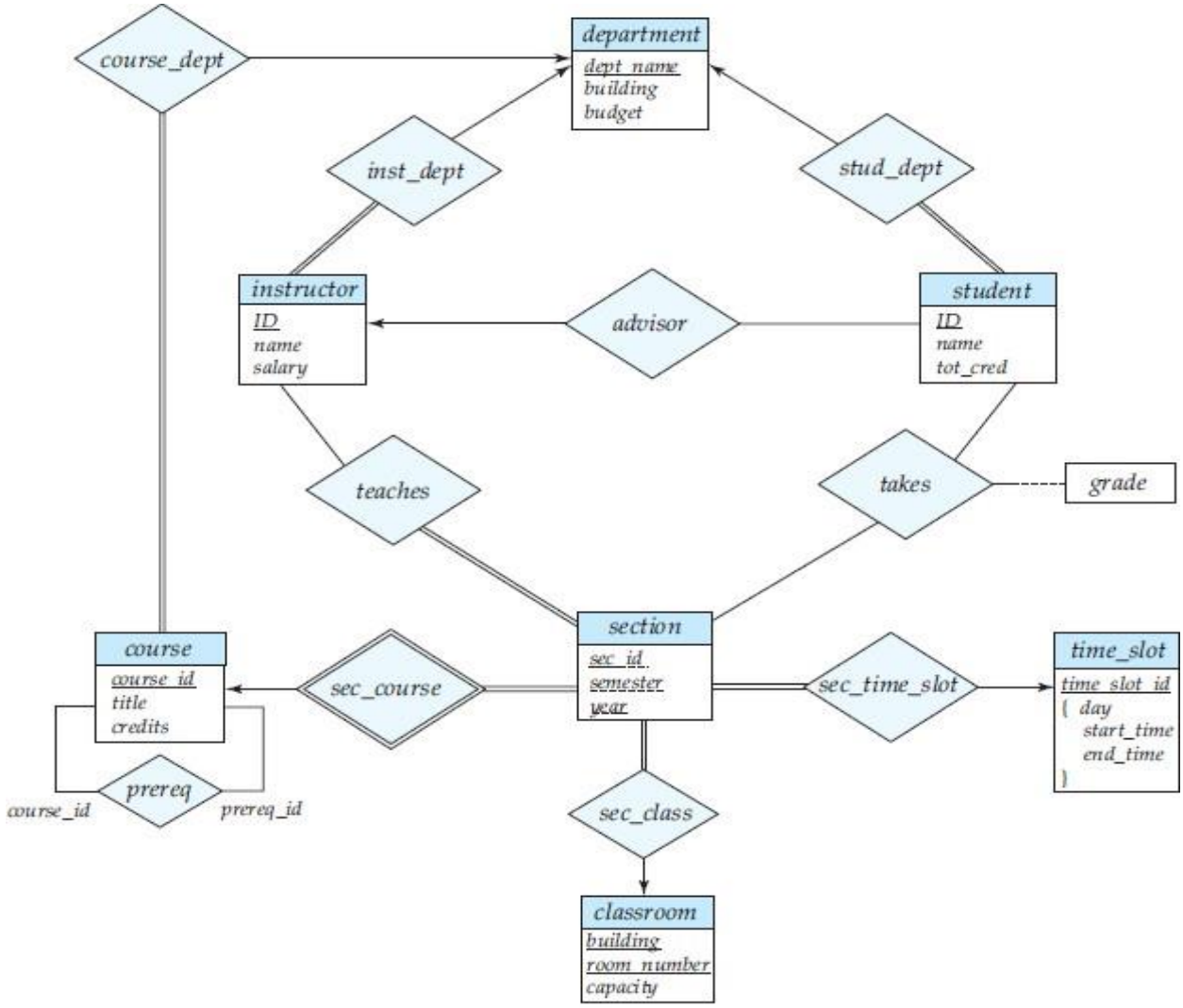


Figure E-R diagram for a university enterprise.

The Entity-Relationship Model: The entity-relationship (E-R) data model uses a collection of basic objects, called ***entities***, and ***relationships*** among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. For example, each person is an entity, and bank accounts can be considered as entities. Entities are described in a database by a set of attributes. For example, the attributes ID, name, and salary may describe an instruct or entity.

The extra attribute ID is used to identify an instructor uniquely (since it may be possible to have two instructors with the same name and the same salary). In the United States, many organizations use the social-security number of a person (a unique number the U.S. government assigns to every person in the United States) as a unique identifier.

❖ A relationship is an association among several entities. For example, a member relationship

associates an instructor with her department. The set of all entities of the same type and the set of all relationships of the same type are termed an ***entity se***t and ***relationship set***,respectively.

The overall logical structure (schema) of a database can be expressed graphically by an entity- relationship (E-R) diagram. There are several ways in which to draw these diagrams. One of the most popular is to use the Unified Modeling Language (UML). In the notation we use, which is based on UML, an E-R diagram is represented as follows:

- ***Entity sets*** are represented by a partitioned rectangular box with the entity set name in the header and the attributes listed below it.
- ***Relationship sets*** are represented by a diamond connecting a pair of related entity sets. The name of the relationship is placed inside the diamond.

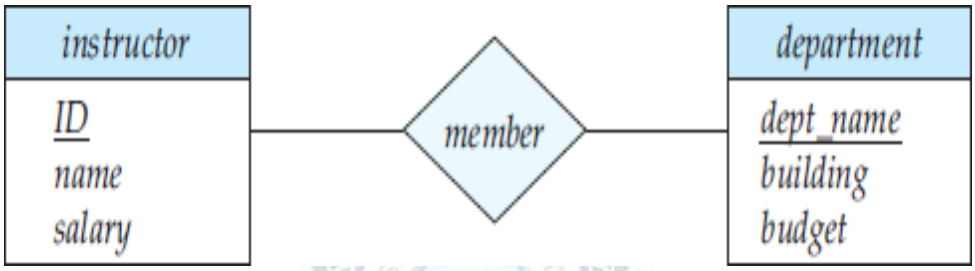


Fig: E-R Diagram

In addition to entities and relationships, the E-R model represents certain constraints to which the contents of a database must conform. One important constraint is mapping cardinalities, which express the number of entities to which another entity can be associated via a relationship set. For example, if each instructor must be associated with only a single department, the E-R model can express that constraint.

The entity-relationship model is widely used in database design.

Normalization: Another method for designing a relational database is to use normalization. To understand the need for normalization, let us look at what can go wrong in a bad database design. Among the undesirable properties that a bad design may have are:

- Repetition of information
- Inability to represent certain information

The goal of Normalization is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. The approach is to design schemas that are in an appropriate normal form. To determine whether a relation schema is in one of the desirable normal forms, we need additional information about the real-world enterprise that we are modeling with the database. The most common approach is to use functional dependencies.

DATABASE DESIGN AND THE E-R MODEL

Overview of the Design Process: The task of creating a database application is a complex one, involving design of the database schema, design of the programs that access and update the data, and design of a security scheme to control access to data. ***Needs of users*** play a central role in design process. **Design Phases:** For small applications, it may be feasible for a database designer who understands the application requirements to decide directly on the relations to be created, their attributes, and constraints on the relations. A high-level and complex data model provides database designer with a conceptual framework to specify data requirements of database users, and how database will be structured to fulfill these requirements.

- ✓ The initial phase of database design is to characterize fully the data needs of database users. The outcome of this phase is a specification of user requirements.
- ✓ Next, the designer chooses E-R data model, and by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database.
- ✓ The schema developed at this conceptual-design phase provides a detailed overview of the enterprise. The entity-relationship model is typically used to represent the conceptual design. By specifying the entities, the attributes of the entities, the relationships among the entities, and constraints on the entities and relationships.

Typically, the conceptual-design phase results in the creation of an entity-relationship diagram that provides a graphic representation of the schema.

- The designer reviews the schema to confirm that all data requirements are indeed satisfied and are not in conflict with one another. Designer can also examine the design to remove any redundant features. The focus at this point is on describing the data and their relationships, rather than on specifying physical storage details.

The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases.

- ❖ In the ***logical-design phase***, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used.
- ❖ Finally, the designer uses the resulting system-specific database schema in the subsequent ***physical- design phase***, in which the physical features of the database are specified.

Note: The physical schema of a database can be changed relatively easily after an application has been built. However, changes to the logical schema are usually harder to carry out, since they may affect a number of queries and updates scattered across application code. It is therefore important to carry out the database design phase with care, before building the rest of the database application.

Design Alternatives: A major part of the database design process is deciding how to represent in the design the various types of "things" such as people, places, products, and the like. We use the term entity to refer to any such distinctly identifiable item. In a university database, examples

of entities would include instructors, students, and departments. Various entities are related to each other in a variety of ways, all of which need to be captured in database design.

In designing a database schema, we must avoid two major pitfalls:

1. **Redundancy:** A bad design may repeat information. Redundancy can also occur in a relational schema. The biggest problem with redundant information is that the copies of a piece of information can become inconsistent if the information is updated without taking precautions to update all copies of the information. Ideally, information should appear in exactly one place.
2. **Incompleteness:** A bad design may make certain aspects of enterprise difficult/impossible to model. For example, suppose that, we only had entities corresponding to course offering, without having an entity corresponding to courses. Equivalently, in terms of relations, suppose we have a single relation where we repeat all of the course information once for each section that the course is offered. It would then be impossible to represent information about a new course, unless that course is offered.

THE ENTITY-RELATIONSHIP MODEL

The entity-relationship (E-R) data model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database. The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the E-R model. The E-R data model employs three basic concepts: entity sets, relationship sets, and attributes.

Entity Sets: An ***entity*** is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in a university is an entity. An entity has a set of properties, and the values for some set of properties may uniquely identify an entity. For instance, a person may have a person id property whose value uniquely identifies that person. An entity may be concrete, such as a person or a book, or it may be abstract, such as a course, a course offering, or a flight reservation.

An ***entity set*** is a set of entities of the same type that share the same properties, or attributes. The set of all people who are instructors at a given university, for example, can be defined as the entity set instructor. We use the term extension of the entity set to refer to the actual collection of entities belonging to the entity set. Thus, the set of actual instructors in the university forms

76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

the extension of the entity set instructor. The above distinction is similar to the difference between a relation and a relation instance.An entity is represented by a set of ***attributes***.

Attributes are descriptive properties possessed by each member of an entity set. Possible attributes of instructor entity set are ID, name, dept_name, and salary. Each entity has a ***value*** for each of its attributes. For instance, a particular instructor entity may have value 12121 for ID, the value Wu for name, the value Finance for dept_name, and the value 90000 for salary.

Fig: Instructor Entity Set

Relationship Sets: A ***relationship*** is an association among several entities. A ***relationship set*** is a set of relationships of same type. Formally, it is a mathematical relation on $n \geq 2$ (possibly non-distinct) entity sets.

If E1, E2,..., En are entity sets, then a relationship set R is a subset of

{(e1,e2,...,en) | e1 ∈ E1,e2 ∈ E2,...,en ∈ En} where (e1,e2,...,en) is a relationship.

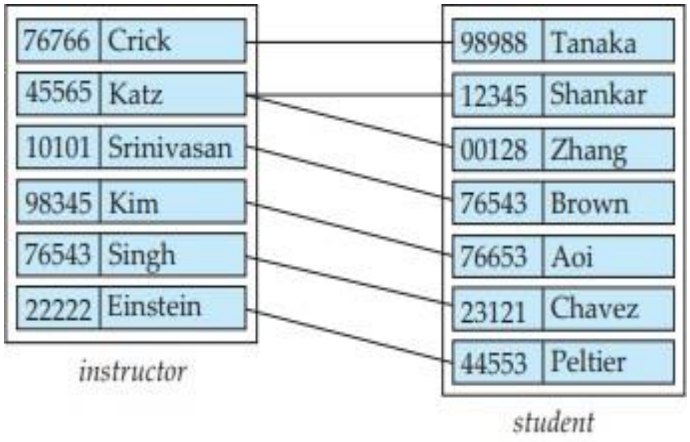


Figure Relationship set *advisor*.

Consider the two entity sets instructor and student in Figure. We define the relationship set advisor to denote the association between instructors and students. The association between entity sets is referred to as ***participation***; that is, the entity sets E1, E2,..., En participate in relationship set R. A relationship instance in an E-R schema represents an association between the named entities.

Ex: The individual instructor entity Katz, who has instructor ID 45565, and the student entity Shankar, who has student ID 12345, participate in a relationship instance of advisor. This relationship instance represents that in the university, the instructor Katz is advising student Shankar.

The function that an entity plays in a relationship is called that entity's ***role***. Since entity sets participating in a relationship set are generally distinct, roles are implicit and are not usually specified. However, they are useful when the meaning of a relationship needs clarification.

A relationship may also have attributes called ***descriptive attributes***. Consider a relationship set advisor with entity sets instructor and student. We could associate the attribute date with that relationship to specify the date when an instructor became the advisor of a student. Below figure shows the relationship set advisor with a descriptive attribute date.

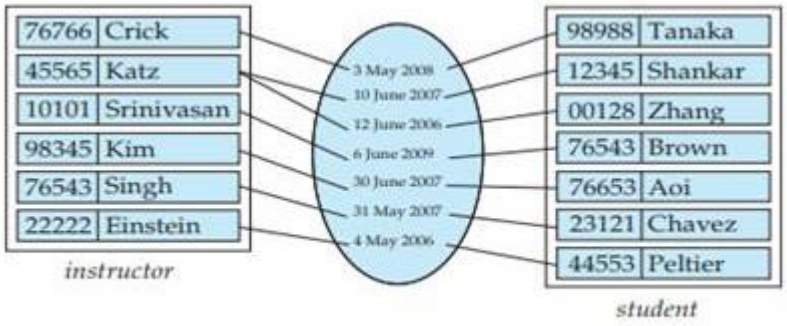


Figure *date* as attribute of the *advisor* relationship set.

The relationship set advisor provides an example of a binary relationship set— that is, one that involves two entity sets. Most of the relationship sets in a database system are binary. Occasionally, however, relationship sets involve more than two entity sets, often called as ternary relationships or non-binary orn-ary relationships. The number of entity sets that participate in a relationship set is the ***degree*** of the relationship set. A binary relationship set is of degree 2; a ternary relationship set is of degree 3.

Attributes: For each attribute, there is a set of permitted values, called the ***domain***, or ***value set***. The domain attribute semester might be strings from the set {Fall, Winter, Spring, Summer}. Since an entity set may have several attributes, each entity can be described by a set of (attribute, data value)pairs.

Ex: A particular instructor entity may be described by the set {(ID, 76766), (name, Crick), (dept_name, Biology), (salary, 72000)}, meaning that the entity describes a person named Crick whose instructor ID is 76766, who is a member of the Biology department with salary of$72,000. An attribute, as used in the E-R model, can be characterized by the following attribute types.

1. **Simple and Composite Attributes:** In our examples thus far, the attributes have been simple; that is, they have not been divided into subparts. Composite attributes, on the other hand, can be divided into subparts (that is, other attributes). For example, an attribute name could be structured as a composite attribute consisting of first name, middle name, and last name. Composite attributes help us to group together related attributes, making the modeling cleaner.

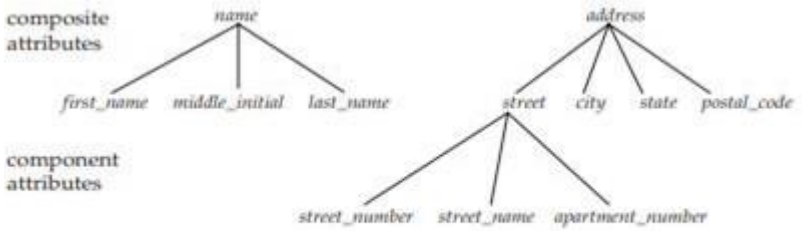


Figure Composite attributes instructor *name* and *address*.

Note: Using composite attributes in a design schema is a good choice if a user will wish to refer to an entire attribute on some occasions and to only a component of the attribute on other occasions.

2. **Single-valued and Multivalued Attributes:** There may be instances where an attribute has a set of values for a specific entity. Suppose an instructor may have one, or several phone numbers, and different instructors may have different numbers of phones. This type of attribute is said to be ***multivalued***. To denote that an attribute is multivalued, we enclose it in braces, for example
   {phone_number} or {dependent name}.

Note: Upper and lower bounds may be placed on number of values in a multivalued attribute, if needed.

3. **Derived Attributes:** Value for Derived attribute can be derived from values of other related attributes or entities. For instance, suppose that instructor entity set has an attribute age. If instructor entity set also has an attribute date_of_birth, we can calculate age from date_of_birth and current date. Thus, age is a derived attribute. In this case, date_of_birthmay be referred to as a base attribute, or a stored attribute. Value of a derived attribute is not stored but is computed when required.

- ✓ An attribute takes a null value when an entity does not have a value for it. The null value may indicate "not applicable" (that is, that the value does not exist for the entity) or "unknown". An unknown value may be either missing (the value does exist, but we do not have that information) or not known (we do not know whether or not the value actually exists). A null value for the apartment number attribute could mean that the address does not include an apartment number (not applicable), that an apartment number exists but we do not know what it is (missing), or that we do not know whether or not an apartment number is part of the instructor's address (unknown).

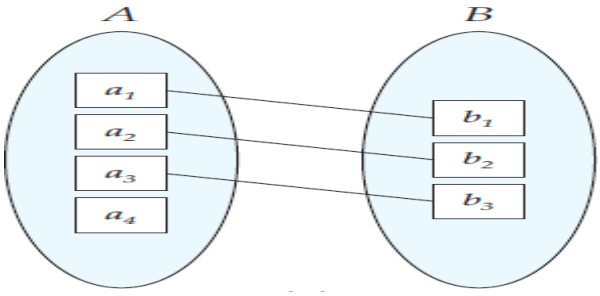
CONSTRAINTS

An E-R schema may define certain constraints to which the contents of a database must conform.

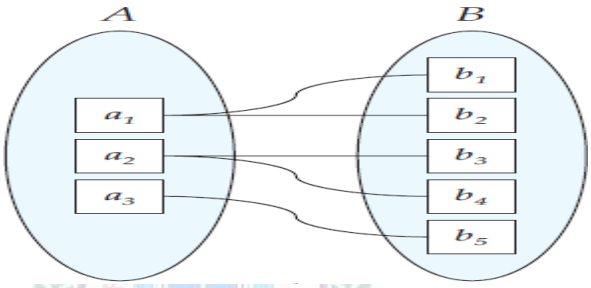
Mapping Cardinalities: Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set. Mapping cardinalities are most useful in describing binary relationship sets. For a binary relationship set R between

entity sets A and B, the mapping cardinality must be one of the following:

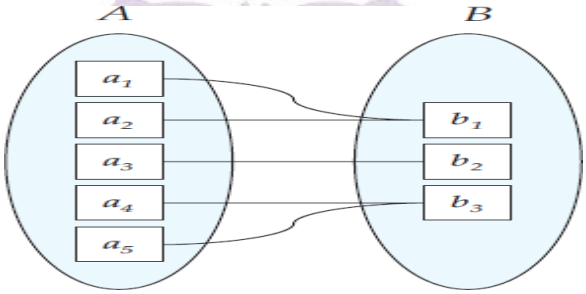
1) **One-to-One:** An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.



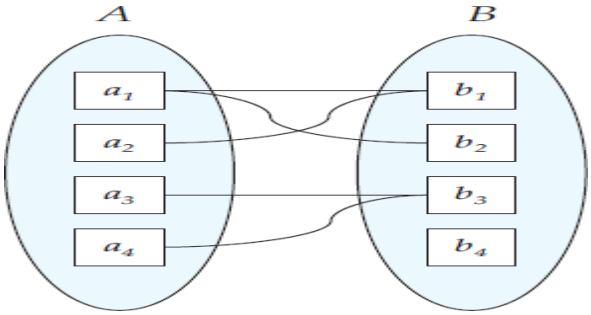
2) **One-to-Many:** An entity in A is associated with any number (zero or more) of entities in B. An entity in B, however, can be associated with at most one entity in A.



3) **Many-to-One:** An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A.



4) **Many-to-Many:** An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A.



The appropriate mapping cardinality for a particular relationship set obviously depends on the real-world situation that the relationship set is modeling. As an illustration, consider the advisor

relationship set. If, in a particular university, a student can be advised by only one instructor, and an instructor can advise several students, then the relationship set from instructor to student is one-to-many. If a student can be advised by several instructors (as in case of students advised jointly), relationship set is many-to-many.

Participation Constraints: The participation of an entity set E in a relationship set R is said to be total if every entity in E participates in at least one relationship in R. If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be partial.

Keys: We must have a way to specify how entities within a given entity set are distinguished. Conceptually, individual entities are distinct; from a database perspective, however, the differences among them must be expressed in terms of their attributes. Therefore, the values of the attribute values of an entity must be such that they can uniquely identify the entity. In other words, no two entities in an entity set are allowed to have exactly the same value for all attributes. ***A key for an entity is a set of attributes that suffice to distinguish entities from each other.*** Keys also help to identify relationships uniquely, and thus distinguish relationships from each other.

The **Primary Key** of an entity set allows us to distinguish among the various entities of the set. We use a similar mechanism to distinguish among the various relationships of a relationship set. The structure of the primary key for the relationship set depends on the mapping cardinality of the relationship set.

A ***super key*** is a set of columns (attributes) to uniquely identify rows in a table. If the attribute names of primary keys are not unique across entity sets, the attributes are renamed to distinguish them; the name of the entity set combined with the name of the attribute would form a unique name.

- If an entity set participates more than once in a relationship set, the role name is used instead of the name of the entity set, to form a unique attribute name.

REMOVING REDUNDANT ATTRIBUTES IN ENTITY SETS

When we design a database using the E-R model, we usually start by *identifying the entity sets* that should be included. For example, in the university organization, we include entity sets such as student, instructor, etc. Then we must *choose the appropriate attributes*. In the university organization, we decided that for the instructor entity set, we will include the attributes ID, name, dept_name, and salary. The choice of what attributes to include is up to the designer, who has a good understanding of the structure of the enterprise. Once the entities and their corresponding attributes are chosen, *the relationship sets among the various entities* are formed.

These relationship sets may result in a situation where attributes in the various entity sets are

redundant and need to be removed from the original entity sets. To illustrate, consider the entity sets instructor and department:

- ❖ Entity set instructor includes attributes ID, name, dept_name, and salary, with ID forming primary key.
- ❖ Entity set department includes attributes dept_name, building, and budget, with dept_name forming primary key.

The attribute dept_name appears in both entity sets. Since it is the primary key for the entity set department, it is redundant in the entity set instructor and needs to be removed. When we create a relational schema from the E-R diagram, the attribute dept_namein fact gets added to the relation instructor, but only if each instructor has at most one associated department. If an instructor has more than one associated department, the relationship between instructors and departments is recorded in a separate relationins t_dept.

Note: Treating the connection between instructors and departments uniformly as a relationship, rather than as an attribute of instructor, makes the logical relationship explicit, and helps avoid a premature assumption that each instructor is associated with only one department.

A good entity-relationship design does not contain redundant attributes.

For our university example, we list entity sets and their attributes below, with primary keys underlined:

- ✓ classroom: with attributes (building, room_number,capacity).
- ✓ department: with attributes (dept_name, building,budget).
- ✓ course: with attributes (course_id, title,credits).
- ✓ instructor: with attributes (ID, name, salary).
- ✓ section: with attributes (course_id, sec_id, semester,year).
- ✓ student: with attributes (ID, name,tot_cred).

The relationship sets in our design are listed below:

- ✓ inst_dept: relating instructors with departments.
- ✓ stud_dept: relating students with departments.
- ✓ teaches: relating instructors with sections.
- ✓ takes: relating students with sections, with a descriptive attribute grade.
- ✓ course_dept: relating courses with departments.
- ✓ sec_course: relating sections with courses.
- ✓ sec_class: relating sections with classrooms.
- ✓ advisor: relating students with instructors.

You can verify that none of the entity sets has any attribute that is made redundant by one of the relationship sets.

ENTITY-RELATIONSHIP DIAGRAMS

An **E-R diagram** can express the overall logical structure of a database graphically. *E-R diagrams are simple and clear*—qualities that may well account in large part for the widespread use of the E-R model.

Basic Structure: An E-R diagram consists of the following major components:

- **Rectangles divided into two parts** represent entity sets. The first part, contains the name of the entity set. The second part contains the names of all the attributes of the entity set.
- **Diamonds** represent relationship sets.
- **Undivided rectangles** represent the attributes of a relationship set. Attributes that are part of the primary key are underlined.
- **Lines** link entity sets to relationship sets.
- **Dashed lines** link attributes of a relationship set to the relationship set.
- **Double lines** indicate total participation of an entity in a relationship set.
- **Double diamonds** represent identifying relationship sets linked to weak entity sets.

Figure E-R diagram corresponding to instructors and students.

Consider the E-R diagram in figure, which consists of two entity sets, *instructor* and *student* related through a binary relationship set *advisor*. The attributes associated with *instructor* are *ID*, *name*, and *salary*. The attributes associated with *student* are *ID*, *name*, and *tot_cred*. In figure, attributes of an entity set that are members of the primary key are underlined.

If a relationship set has some attributes associated with it, then we enclose the attributes in a rectangle and link the rectangle with a dashed line to the diamond representing that relationship set. For example, in below figure, we have the *date* descriptive attribute attached to the relationship set *advisor* to specify the date on which an instructor became the advisor.

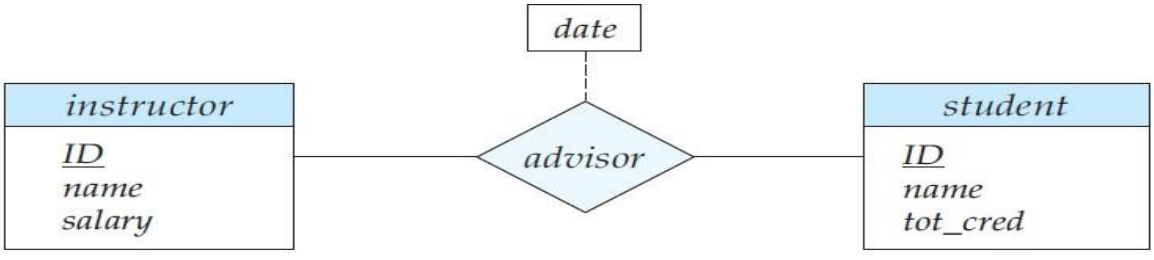


Figure E-R diagram with an attribute attached to a relationship set.

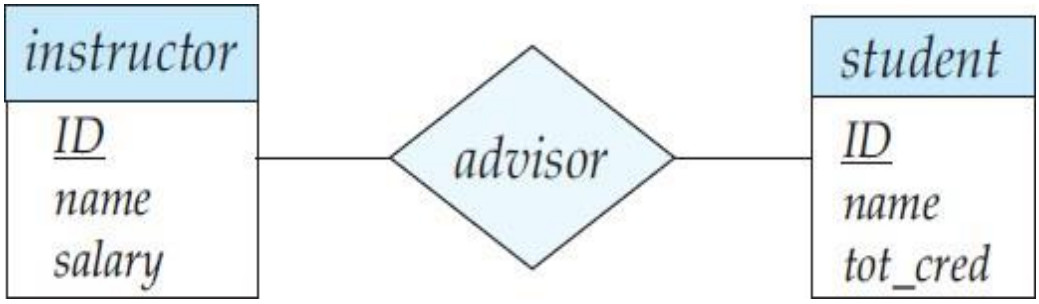
Mapping Cardinality: The relationship set *advisor*, between the *instructor* and *student* entity sets may be one-to-one, one-to-many, many-to-one, or many-to-many. To distinguish among these types, we draw either a directed line (→) or an undirected line (—) between the relationship set and the entity set in question, as follows:

1) One-to-one: We draw a directed line from the relationship set *advisor* to both entity sets *instructor* and *student*. This indicates that an instructor may advise at most one student, and a student may have at most one advisor.

2) One-to-many: We draw a directed line from the relationship set *advisor* to the entity set *instructor* and an undirected line to the entity set *student*. This indicates that an instructor may advise many students, but a student may have at most one advisor.

3) Many-to-one: We draw an undirected line from the relationship set *advisor* to the entity set *instructor* and a directed line to the entity set *student*. This indicates that an instructor may advise at most one student, but a student may have many advisors.

4) Many-to-many: We draw an undirected line from the relationship set *advisor* to both entity sets *instructor* and *student*. This indicates that an instructor may advise many students, and a student may have many advisors.



E-R diagrams also provide a way to indicate more complex constraints on the number of times each entity participates in relationships in a relationship set. ***A line may have an associated minimum and maximum cardinality, shown in the form l..h, where l is the minimum and h the maximum cardinality.*** A minimum value of 1 indicates total participation of the entity set in the relationship set; that is, each entity in the entity set occurs in at least one relationship in that relationship set. A maximum value of 1 indicates that the entity participates in at most one relationship, while a maximum value ∗ indicates no limit.

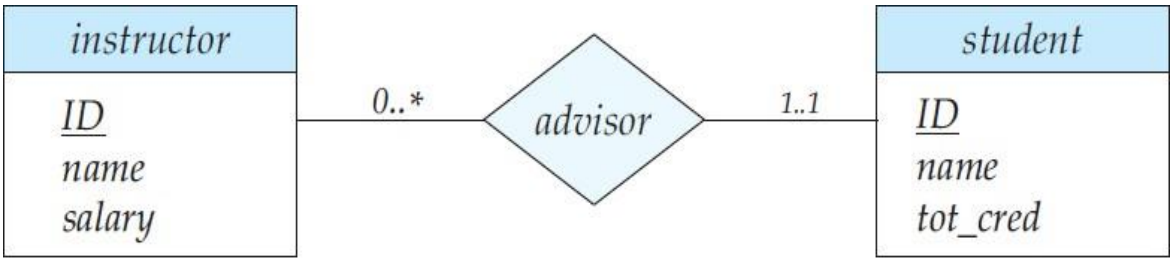


Figure Cardinality limits on relationship sets.

For example, consider the figure, the line between *advisor* and *student* has a cardinality constraint of 1..1, meaning the minimum and the maximum cardinality are both 1. That is, each student must have exactly one advisor. The limit 0..∗on the line between *advisor* and *instructor* indicates that an instructor can havezero or more students. Thus, the relationship *advisor* is one-to-many from *instructor* to *student*, and further the participation of *student* in *advisor* is total, implying that a student must have ana dvisor.

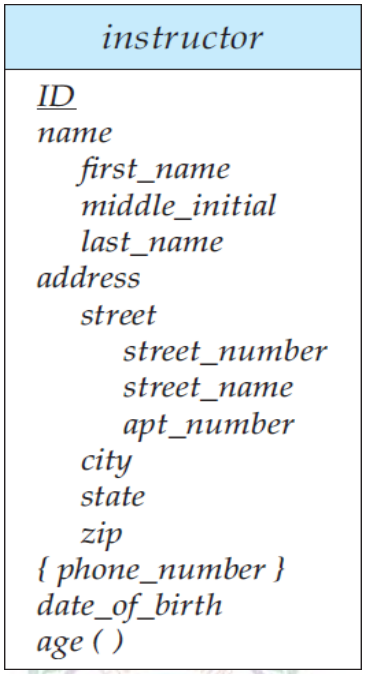
If both edges have a maximum value of 1, relationship is one-to-one. If we had specified a cardinality limit of 1..∗ on left edge, we would be saying that each instructor must advise at least one student.

Note: The E-R diagram in the figure could alternatively have been drawn with a double line from *student* to *advisor* and an arrow on the line from *advisor* to *instructor*, in place of cardinality constraints shown. This alternative diagram would enforce exactly the same constraints as the constraints shown in figure.

Complex Attributes:

Following figure shows how composite attributes can be represented in the E-R notation. As an example, suppose we were to add an address to the *instructor* entity-set. The address can be defined as the composite attribute *address* with the attributes: *street*, *city*, *state*, and *zip code*. The attribute *street* is itself a composite attribute whose component attributes are *street number*,

street name, and *apartment number*.The figure also illustrates a multivalued attribute *phone_number*, denoted by "*{phone_number}*", and a derived attribute *age*, depicted by a "*age*



( )".

Roles: We indicate roles in E-R diagrams by labeling the lines that connect diamonds to rectangles. Figure 7.12 shows the role indicators *course_id*and *prereq id* between the *course* entity set and the *prereq* relationship set.

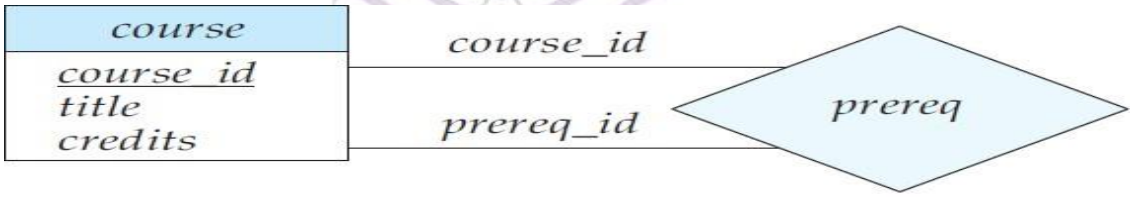


Figure E-R diagram with role indicators.

Non-binary/n-ary/Ternary Relationship Sets: Non-binary/n-ary/Ternary relationship sets can be specified easily in an E-R diagram. Figure 7.13 consists of the three entity sets *instructor*, *student*, and *project*, related through the relationship set*proj_guide*.

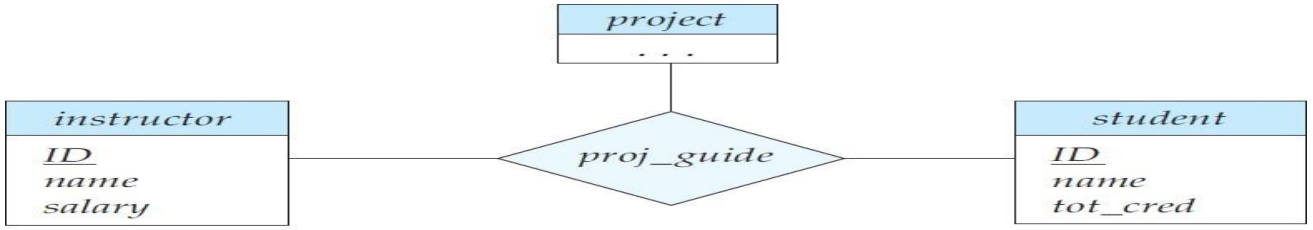


Figure E-R diagram with a ternary relationship.

We can specify some types of many-to-one relationships in the case of non-binary relationship sets. Suppose a *student* can have at most one instructor as a guide on a project. This constraint can be specified by an arrow pointing to *instructor* on the edge from *proj_guide*.

We permit at most one arrow out of a relationship set, since an E-R diagram with two or more arrows out of a non-binary relationship set can be interpreted in two ways.

Suppose there is a relationship set R between entity sets $A_1, A_2, \ldots, A_n$, and the only arrows are on the edges to entity sets $A_{i+1}, A_{i+2}, \ldots, A_n$. Then, the two possible interpretations are:

1. A particular combination of entities from $A_1, A_2, \ldots, A_i$can be associated with at most one combination of entities from $A_{i+1}, A_{i+2}, \ldots, A_n$. Thus, the primary key for the relationship R can be constructed by the union of the primary keys of $A_1, A_2, \ldots, A_i$.

2. For each entity set A_k, $i < k \leq n$, each combination of the entities from other entity sets can beassociated with at most one entity from A_k. Each set $\{A_1, A_2, \ldots, A_{k-1}, A_{k+1}, \ldots, A_n\}$, for $i < k \leq n$, then forms a candidate key.

To avoid confusion, we permit only one arrow out of a relationship set, in which case the two interpretations are equivalent.

Weak Entity Sets: An entity set that is dependent on other entity set (or) an entity set that does not have a primary key is referred to as a **weak entity set**. An entity set that has a primary key is termed a **strong entity set**.

For a weak entity set to be meaningful, it must be associated with another entity set, called the **identifying** or **owner entity set**. The identifying entity set is said to **own** the weak entity set that it identifies. The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**. The identifying relationship is many-to-one from the weak entity set to the identifying entity set, and the participation of the weak entity set in the relationship is total.

In our example, the identifying entity set for *section* is *course*, and the relationship *sec_course*, which associates *section* entities with their corresponding *course* entities, is the identifying relationship.

The **discriminator (***partial key***)** of a weak entity set is a set of attributes that allows the distinction tobemadeasweakentitysetdoesnothavePrimaryKey.Forexample,thediscriminatoroftheweakentity set *section* consists of the attributes *sec_id*, *year*, and *semester*, since, for each course, this set of attributes uniquely identifies one single section for that course.

The primary key of a weak entity set is formed by the primary key of the identifying entity set, plus the weak entity set's discriminator. In the case of the entity set *section*, its primary key is *{course_id, sec_id, year, semester}*, where *course_id*is the primary key of the identifying entity set, namely *course*, and *{sec_id, year, semester}* distinguishes *section* entities for the same

Figure E-R diagram with a weak entity set.

course.

❖ In Figure, the weak entity set *section* depends on the strong entity set *course* via the relationship set

sec_course.

In E-R diagrams, a weak entity set is depicted with a rectangle, like a strong entity set, but there are two main differences:

✓ The discriminator of a weak entity is underlined with a dashed, rather than a solid,line.

✓ Relationship set connecting weak entity set to identifying strong entity set is depicted by a double diamond.

The above figure also illustrates the use of double lines to indicate *total participation*; finally, the arrow from *sec_course*to *course* indicates that each section is related to a single course.

Note: A weak entity set can also participate in relationships other than the identifying relationship. A weak entity set may participate as owner in an identifying relationship with another weak entity set. It is also possible to have a weak entity set with more than one identifying entity set. A particular weak entity would then be identified by a combination of entities, one from each identifying entity set. The primary key of the weak entity set would consist of the union of the primary keys of the identifying entity sets, plus the discriminator of the weak entity set. A weak entity set representation more aptly models a situation where set participates in relationships than identifying relationship, and where weak entity set has several attributes.

REDUCTION TO RELATIONAL SCHEMAS

We can represent a database that conforms to an E-R database schema by a collection of relation schemas. For each entity set and for each relationship set in the database design, there is a unique relation schema to which we assign the name of the corresponding entity set or relationship set.

Both the E-R model and the relational database model are abstract, logical representations of real- world enterprises. Because the two models employ similar design principles, we can

convert an E-R design into a relational design.

1. Representation of Strong Entity Sets with Simple Attributes: Let *E* be a strong entity set with only simple descriptive attributes $a_1, a_2, \ldots, a_n$. We represent this entity by a schema called *E* with *n* distinct attributes. Each tuple in a relation on this schema corresponds to one entity of the entity set *E*. For schemas derived from strong entity sets, the primary key of the entity set serves as the primary key of the resulting schema. This follows directly from the fact that each tuple corresponds to a specific entity in the entity set.

As an example, consider the entity set *student* of above E-R diagram. This entity set has three attributes: *ID*, *name*, *tot_cred*. We represent this entity set by a schema called *student* with three attributes: *student* (*ID*, *name*, *tot_cred*)

Note that since student ID is primary key of entity set, it is also the primary key of the relation schema.

2. Representation of Strong Entity Sets with Complex Attributes: Composite attributes are handled by creating a separate attribute for each of the component attributes; we do not create a separate attribute for the composite attribute itself.

To illustrate, consider the *instructor* entity set. For the composite attribute *name*, the schema generated for *instructor* contains the attributes *first_name*, *middle_name*, and *last_name*; there is no separate attribute for *name*. Similarly, for the composite attribute *address*, the schema generated contains the attributes *street*, *city*, *state*, and *zip code*. Since *street* is a composite attribute it is replaced by *street number*, *street name*, and *apt number*. The relational schema derived from the version of entity set *instructor* with complex attributes, without including the multivalued attribute, is thus:

instructor (*ID*, *first_name*, *middle_name*, *last_name*, *street_number*, *street_name*, *apt_number*, *city*, *state*, *zip_code*, *date_of_birth*)

- ***Multivalued attributes are treated differently from other attributes.*** We have seen that attributes in an E-R diagram generally map directly into attributes for appropriate relation schemas. Multivalued attributes, however, are an exception; new relation schemas are created for these attributes. For a multivalued attribute *M*, we create a relation schema *R* with an attribute *A* that corresponds to *M* and attributes corresponding to primary key of the entity set or relationship set of which *M* is an attribute.

As an example, consider entity set *instructor*, which includes multivalued attribute *phone_number*. Primary key of *instructor* is *ID*. For this multivalued attribute, we create a relation schema *instructor_phone*(*ID*, *phone_number*). Each phone_number of an instructor is represented as a unique tuple in relation on this schema. Thus, if we had an instructor with *ID* 22222, and phone_numbers 9849012345 and 9848012345, the relation *instructor phone* would have two tuples (22222, 9849012345)and (22222, 9848012345).

We create a primary key of the relation schema consisting of all attributes of the schema. We also create a foreign-key on the relation schema created from multivalued attribute, with the primary key of the entity set referencing the relation generated from the entity set. In the above example, the foreign-key constraint on the *instructor_phone* relation would be that attribute *ID* references the *instructor* relation.

- ❖ Derived attributes are not explicitly represented in the relational data model. However, they can be represented as "methods" in other data models such as the object-relational data model

3. **Representation of Weak Entity Sets:** Let *A* be a weak entity set with attributes a_1, $a_2, \ldots, a_m$. Let *B* be the strong entity set with attributes $b_1, b_2, \ldots, b_n$ on which *A* depends. Let the primary key of *B* is b_1. We represent the entity set *A* by a relation schema called *A* as: $\{a_1, a_2, \ldots, a_m\} \cup \{b_1\}$

For schemas derived from a weak entity set, the combination of the primary key of the strong entity set and the discriminator of the weak entity set serves as the primary key of the schema. We also create a foreign-key constraint on the relation *A*, specifying that the attribute b_1 reference the primary key of the relation *B*. The foreign key constraint ensures that for each tuple representing a weak entity, there is a corresponding tuple representing the corresponding strong entity.

Ex: *section* (*course_id*, *sec_id*, *semester*, *year*)

- ✓ Because of the "on delete cascade" specification on the foreign key constraint, if a *course* entity is deleted, then so are all the associated *section* entities.

4. **Representation of Relationship Sets:** Let *R* be a relationship set, let $a_1, a_2, \ldots, a_m$ be the set of attributes formed by the union of the primary keys of each of the entity sets participating in *R*, and let the descriptive attributes (if any) of *R* be $b_1, b_2, \ldots, b_n$. We represent this relationship set by a relation schema called *R* with one attribute for each member of the set: $\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$

The primary key is chosen as follows:

- ❖ For a binary many-to-many relationship, the union of the primary-key attributes from the participating entity sets becomes the primary key.
- ❖ For a binary one-to-one relationship set, the primary key of either entity set can be chosen as the primary key.
- ❖ For a binary many-to-one or one-to-many relationship set, the primary key of the entity set on the "many" side of the relationship set serves as the primary key.
- ❖ For an *n*-ary relationship set without any arrows on its edges, the union of the primary key- attributes from the participating entity sets becomes the primary key.
- ❖ For an *n*-ary relationship set with an arrow on one of its edges, the primary keys of the

entity sets not on the "arrow" side of the relationship set serve as the primary key for the schema. Recall that we allowed only one arrow out of a relationship set.

We can also create foreign-key constraints on the relation schema *R.* As an illustration, consider the relationship set *advisor* in the E-R diagram. This relationship set involves the following two entity sets:

- *instructor* with the primary key*ID.*
- *student* with the primary key*ID.*

Since the relationship set has no attributes, the *advisor* schema has two attributes, the primary keys of *instructor* and *student*. Since both attributes have the same name, we rename them *i_ID*and s_*ID*. Since the *advisor* relationship set is many-to- one from *student* to *instructor* the primary key for the *advisor* relation schema is *s ID.*

Note: The schema for the relationship set linking a weak entity set to its corresponding strong entity set is redundant and does not need to be present in a relational database design based upon an E-R diagram.

ENTITY-RELATIONSHIP DESIGN ISSUES

1. **Use of Entity Sets versus Attributes:** Two natural questions arise in a database design are: What constitutes an attribute, and what constitutes an entity set? Unfortunately, there are no simple answers. The distinctions mainly depend on the structure of the real-world enterprise being modeled, and on the semantics associated with the attribute in question.

A common mistake is to use the primary key of an entity set as an attribute of another entity set, instead of using a relationship. Another related mistake that people sometimes make is to designate the primary-key attributes of the related entity sets as attributes of the relationship set. This should not be done since the primary-key attributes are already implicit in the relationship set.

2. **Use of Entity Sets versus Relationship Sets:** It is not always clear whether an object is best expressed by an entity set or a relationship set. One possible guideline in determining whether to use an entity set or a relationship set is to designate a relationship set to describe an action that occurs between entities. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed asrelation ships.

3. **Binary versus *n*-ary Relationship Sets:** Relationships in databases are often binary. Some relationships that appear to be non-binary could be better represented by several binary relationships. It is always possible to replace a non-binary (*n*-ary, for $n > 2$) relationship set by a number of distinct binary relationship sets. For simplicity, consider the abstract ternary ($n = 3$) relationship set *R*, relating entity sets *A*, *B*, and *C*. We replace the relationship set *R* by an

entity set *E*, and create three relationship sets as shown in figure.

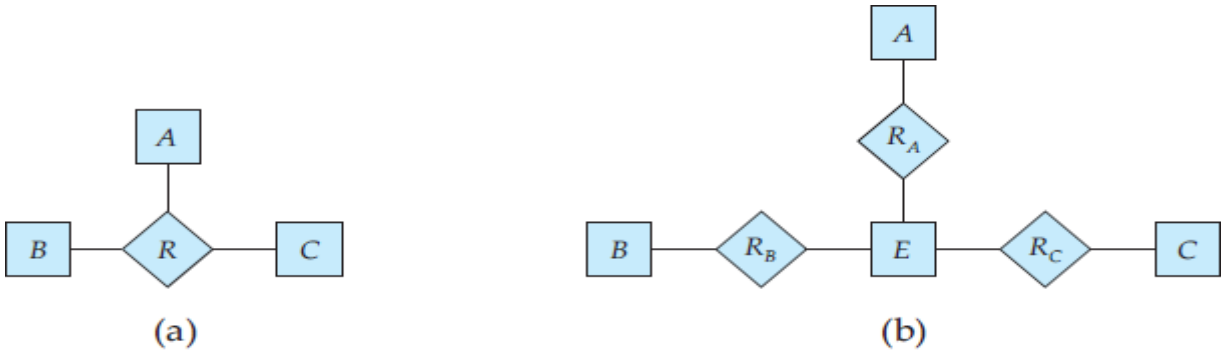


Figure Ternary relationship versus three binary relationships.

If the relationship set *R* had any attributes, these are assigned to entity set *E*; further, a special identifying attribute is created for *E*. For each relationship (a_i, b_i, c_i) in the relationship set *R*, we create a new entity $e_i$in the entity set *E*. Then, in each of the three new relationship sets, we insert a relationship as follows:

(e_i, a_i) in*RA*.

(e_i, b_i) in*RB*.

(e_i, c_i) in*RC*.

We can generalize this process in a straightforward manner to *n*-ary relationship sets. Conceptually, we can restrict E-R model to include only binary relationship sets. But this restriction is not always desirable.

- ✓ An identifying attribute may have to be created for the entity set created to represent the relationship set to increase the complexity of the design and overall storage requirements.
- ✓ An *n*-ary relationship set shows more clearly that several entities participate in a single relationship.
- ✓ There may not be a way to translate constraints on the ternary relationship into constraints on the binary relationships.

4. Placement of Relationship Attributes: The cardinality ratio of a relationship can affect the placement of relationship attributes. Thus, attributes of one-to-one or one-to-many relationship sets can be associated with one of the participating entity sets, rather than with the relationship set.

- ✓ For one-to-one relationship sets, relationship attribute can be associated with either of the participating entities.
- ✓ Attributes of a one-to-many relationship set can be repositioned to only the entity set on the "many" side of the relationship.

- The design decision of where to place descriptive attributes in such cases, as a relationship or entity attribute, should reflect the characteristics of the enterprise being modeled.

✓ The choice of attribute placement is more clear-cut for many-to-many relationship sets. When an attribute is determined by the combination of participating entity sets, rather than by either entity separately, that attribute must be associated with the many-to-many relationship set.

EXTENDED E-R FEATURES

Although the basic E-R concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. The extended E-R features include: specialization, generalization, attribute inheritance, and aggregation.

❖ **Specialization:** "The process of designating sub groupings within an entity set is called **specialization**". An entity set may include sub groupings of entities that are distinct in some way from other entities in the set. The E-R model provides a means for representing these distinctive entity groupings. As an example, the entity set *person* may be further classified as one of the following:

- *employee*.
- *student*.

Each of these person types is described by a set of attributes that includes all attributes of entity set *person* plus possibly additional attributes. For example, *employee* entities may be described further by attribute *salary*, whereas *student* entities may be described further by attribute *tot_cred*. The specialization of *person* allows us to distinguish among person entities according to whether they correspond to employees or students: in general, a person could be an employee, a student, both, or neither.

We can apply specialization repeatedly to refine a design. For instance, university employees may be further classified as one of the following:

- *instructor*.
- *secretary*.

Each of these employee types is described by a set of attributes that includes all the attributes of entity set *employee* plus additional attributes. For example, *instructor* entities may be described further by the attribute *rank* while *secretary* entities are described by the attribute *hours per week*. An entity set may be specialized by more than one distinguishing feature. In our example, the distinguishing feature among employee entities is the job the employee performs.

We refer to this relationship as the ***ISA relationship***, which stands for "is a" and represents, for example, that an instructor "is a" employee. In terms of an E-R diagram, specialization is depicted by a hollow arrow-head pointing from the specialized entity to the other entity.

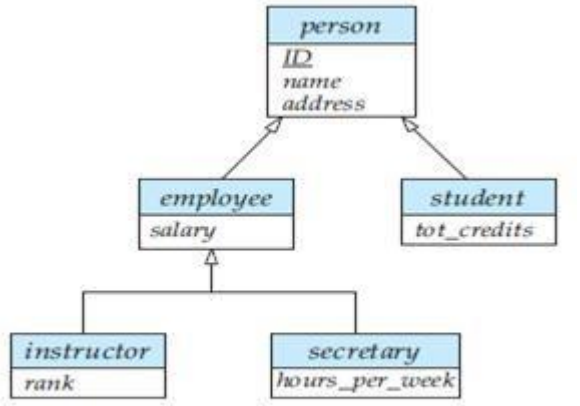


Figure Specialization and generalization.

The way we depict specialization in an E-R diagram depends on whether an entity may belong to multiple specialized entity sets or if it must belong to at most one specialized entity set. The former case (multiple sets permitted) is called **overlapping specialization**, while the latter case (at most one permitted) is called **disjoint specialization**. For an overlapping specialization (as is the case for *student* and *employee* as specializations of *person*), two separate arrows are used. For a disjoint specialization (as is the case for *instructor* and *secretary* as specializations of *employee*), a single arrow is used.

The specialization relationship may also be referred to as a **superclass-subclass** relationship. Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**, respectively. The refinement from an initial entity set into successive levels of entity sub groupings represents a **top-down** design process in which distinctions are made explicit.

- **Generalization:** "The design process in which multiple entity sets are synthesized into a higher- level entity set on the basis of common features is known as **generalization**". This is a **bottom-up** design process.

The database designer may have first identified:

- *instructor* entity set with attributes *instructor_id, instructor_name*, *instructor_salary*, and *rank.*
- *secretary* entity set with attributes *secretary_id*, *secretary_name*, *secretary_salary*, and *hours perweek.*

There are similarities between the *instructor* entity set and the *secretary* entity set in the sense that they have several attributes that are conceptually the same across the two entity sets: namely, the id, name, and salary attributes. To create a generalization, the attributes must be given a common name and represented with the higher-level entity *employee*. In our example, *employee* is the higher-level entity set and *instructor* and *secretary* are lower-level entity sets. The *employee* entity set is the superclass of the *instructor* and *secretary* subclasses.

For all practical purposes, generalization is a simple inversion of specialization. We apply both processes, in combination, in the course of designing the E-R schema for an enterprise. In terms of the E- R diagram itself, we do not distinguish between specialization and generalization. New levels of entity representation are distinguished (specialization) or synthesized (generalization) as the design schema comes to express fully the database application and the user requirements of the database. Differences in the two approaches may be characterized by their starting point and overall goal.

Constraints on Generalizations: One type of constraint involves determining which entities can be members of a given lower-level entity set. Such membership may be one of the following:

1. **Condition-defined:** In condition-defined lower-level entity sets, membership is evaluated on the basis of whether or not an entity satisfies an explicit condition or predicate. Since all lower-level entities are evaluated on the basis of the same attribute this type of generalization is said to be **attribute-defined**.
2. **User-defined:** User-defined lower-level entity sets are not constrained by a membership condition; rather, the database user assigns entities to a given entity set. The assignment is implemented by an operation that adds an entity to an entity set.

❖ A second type of constraint relates to whether or not entities may belong to more than one lower-level entity set within a single generalization. The lower level entity sets may be one of the following:

1. **Disjoint:** A *disjoint constraint* requires an entity belong to no more than one lower-level entity set.
2. **Overlapping:** In *overlapping generalizations*, the same entity may belong to more than one lower- level entity set within a single generalization.

❖ A final constraint, the **completeness constraint** on a generalization or specialization, specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within the generalization/specialization. This constraint may be one of the following:

1. **Total Generalization** or **Specialization:** Each higher-level entity must belong to a lower-level entity set.
2. **Partial Generalization** or **Specialization:** Some higher-level entities may not belong to any lower- level entity set.

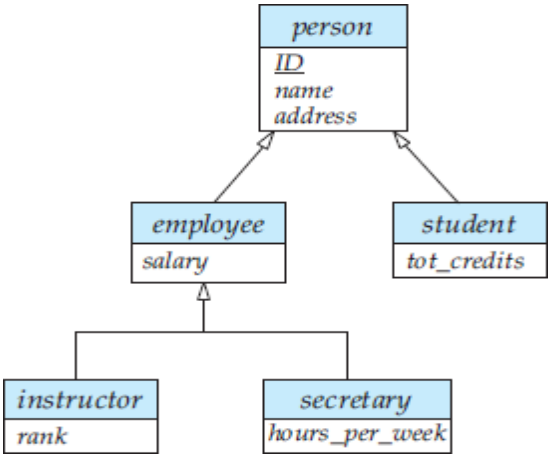
Note: Partial generalization is the default. We can specify total generalization in an E-R diagram by adding the keyword "total" in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrow-head to which it applies (for a total generalization), or to the set of hollow arrow-heads to which it applies (for an overlapping generalization).

Attribute Inheritance: A crucial property of the higher- and lower-level entities created by

specialization and generalization is ***attribute inheritance***. The attributes of the higher-level entity sets are said to be inherited by the lower-level entity sets.

For example, *student* and *employee* inherit the attributes of *person*. Thus, *student* is described by its *ID*, *name*, and *address* attributes, and additionally a *tot_cred* attribute; *employee* is described by its *ID*, *name*, and *address* attributes, and additionally a *salary* attribute. Attribute inheritance applies through all tiers of lower-level entity sets; thus, *instructor* and *secretary*, which are subclasses of *employee*, inherit the attributes *ID*, *name*, and *address* from *person*, in addition to inheriting *salary* from *employee*.

A lower-level entity set (or subclass) also inherits participation in the relationship sets in which its higher- level entity (or superclass) participates. Like attribute inheritance, participation inheritance applies through all tiers of lower-level entity sets. For example, suppose the *person* entity set participates in a relationship *person_dept* with *department*. Then, the *student*, *employee*, *instructor* and *secretary* entity sets, which are subclasses of the *person* entity set, also implicitly participate in the *person_dept* relationship with *department*.



The above figure depicts a **hierarchy** of entity sets. In the figure, *employee* is a lower-level entity set of *person* and a higher-level entity set of the *instructor* and *secretary* entity sets. In a hierarchy, a given entity set may be involved as a lower level entity set in only one ISA relationship; that is, entity sets in this diagram have only **single inheritance**. If an entity set is a lower-level entity set in more than one ISA relationship, then the entity set has **multiple inheritance**, and the resulting structure is said to be a ***lattice***.

- **Aggregation:** One limitation of the E-R model is that it cannot express relationships among relationships. To illustrate the need for such a construct, consider the ternary relationship proj_guide, between an instructor, student and project. Now suppose that each instructor guiding a student on a project is required to file a monthly evaluation report. We model the evaluation report as an entity evaluation, with a primary key evaluation id. One alternative for recording the (student, project, instructor) combination to which an evaluation

corresponds is to create a quaternary (4-way) relationship set eval_for between instructor, student, project, and evaluation.

Using the basic E-R modeling constructs, we obtain the following E-R diagram.

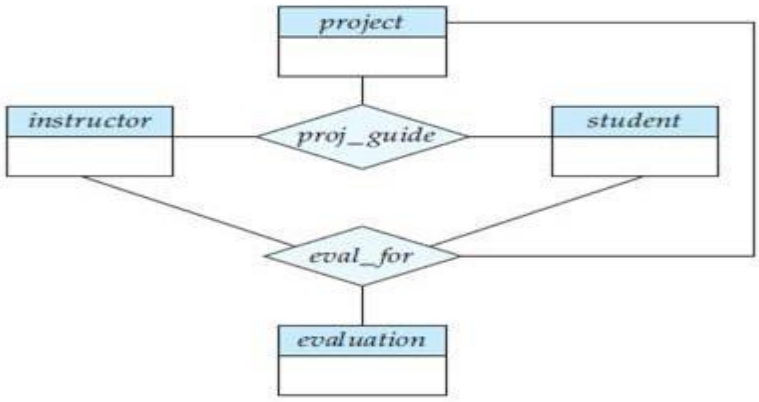


Figure E-R diagram with redundant relationships.

It appears that the relationship sets proj_guide and eval_for can be combined into one single relationship set. Nevertheless, we should not combine them into a single relationship, since some instructor, student, project combinations may not have an associated evaluation.

There is ***redundant information*** in the resultant figure, however, since every instructor, student, project combination in eval_formust also be in proj_guide. 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 proj_guide (relating the entity sets instructor, student, and project) as a higher-level entity set called proj_guide. Such an entity set is treated in the same manner as is any other entity set.

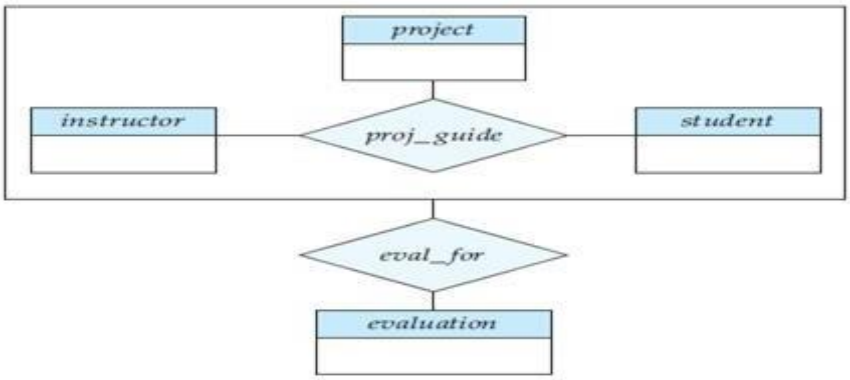


Figure E-R diagram with aggregation.

We can then create a binary relationship eval_for between proj_*guide*and *evaluation* to represent which (*student*, *project*, *instructor*) combination an *evaluation* is for, as shown in the figure.

REDUCTION TO RELATION SCHEMAS

Representation of Generalization: There are two different methods of designing relation schemas for an E-R diagram that includes generalization. We refer to Generalization figure and simplify it by including only the first tier of lower-level entity sets—that is, *employee* and *student*. We assume that *ID* is the primary key of *person*.

1) Create a schema for the higher-level entity set. For each lower-level entity set, create a schema that includes an attribute for each of the attributes of that entity set plus one for each attribute of the primary key of the higher-level entity set. Thus, for the E-R diagram of figure we have three schemas: *person* (*ID, name, street, city*), *employee* (*ID, salary*), *student* (*ID*,*tot_cred*)

❖ The primary-key attributes of the higher-level entity set become primary key attributes of all lower- level entity sets also which can be underlined. In addition, we create foreign-key constraints on the lower-level entity sets. In the above example, the *ID* attribute of *employee* would reference the primary key of *person*, and similarly for *student*.

2) An alternative representation is possible, if the generalization is disjoint and complete. Here, we do not create a schema for the higher-level entity set. Instead, for each lower level entity set, we create a schema that includes an attribute for each of the attributes of that entity set plus one for *each* attribute of the higher-level entity set. Then, for the E-R diagram of figure, we have two schemas: *employee* (*ID, name, street, city, salary*), *student* (*ID, name, street, city*,*tot_cred*)

Both these schemas have *ID*, which is the primary-key attribute of the higher level entity set *person*, as their primary key.

One drawback of second method lies in defining foreign-key constraints. To avoid this problem, we need to create a relation schema containing at least the primary-key attributes of same entity. If the second method were used for an overlapping generalization, some values would be stored multiple times.

Representation of Aggregation: Designing schemas for an E-R diagram containing aggregation is straightforward. Consider the diagram of Aggregation. Schema for the relationship set *eval_for*between the aggregation of *proj_guide*and the entity set *evaluation* includes an attribute for each attribute in the primary keys of the entity set *evaluation*, and the relationship set *proj_guide*. It also includes an attribute for any descriptive attributes, if they exist, of the relationship set *eval_for*. We then transform relationship sets and entity sets within the aggregated entity set following the rules we have already defined.

The rules for creating primary-key and foreign-key constraints on relationship sets can be applied to relationship sets involving aggregations as well, with the aggregation treated like any other entity set. The primary key of the aggregation is the primary key of its defining relationship set. No separate relation is required to represent the aggregation; the relation created from the defining relationship is used instead.

OTHER ASPECTS OF DATABASE DESIGN

Data Constraints and Relational Database Design: Constraints serve several purposes. The most obvious one is the automation of consistency preservation. By expressing constraints in the SQL DDL, the designer is able to ensure that the database system itself enforces the constraints. This is more reliable than relying on each application program individually to enforce constraints. It also provides a central location for the update of constraints and the addition of new ones.

A further advantage of stating constraints explicitly is that certain constraints are particularly useful in designing relational database schemas. Constraint enforcement comes at a potentially high price in performance each time the database is updated.

Data constraints are also useful in determining the physical structure of data. It may be useful to store data that are closely related to each other in physical proximity on disk so as to gain efficiencies in disk access. Certain index structures work better when the index is on a primary key.

Usage Requirements: Queries, Performance: Database system performance is a critical aspect of most enterprise information systems. There are two main metrics for performance:

1. ***Response time***—the amount of time a *single* transaction takes from start to finish in either the average case or the worst case.

2. ***Throughput***—the number of queries or updates (often referred to as *transactions*) that can be processed on average per unit of time.

Systems that process large numbers of transactions in a batch style focus on having high throughput. Systems that interact with people or time-critical systems often focus on response time. Most commercial database systems historically have focused on throughput; however, a variety of applications including Web-based applications require good response time.

Queries that involve joins require more resources to evaluate than those that do not. In cases where a join is required, the database administrator may choose to create an index that facilitates evaluation of that join. For queries—whether a join is involved or not—indices can be created to speed evaluation of selection predicates (SQL where clause) that are likely to appear. Another aspect of queries that affects choice of indices is relative mix of update and read operations. While an index may speed queries, it also slows updates.

Authorization Requirements:

Authorization constraints affect design of the database as well because SQL allows access to be granted to users on the basis of components of the logical design of the database. A relation schema may need to be decomposed into two or more schemas to facilitate the granting of access rights in SQL.

Data Flow, Workflow:

The term *workflow* refers to the combination of data and tasks involved in processes like a CAD system. Workflows interact with the database system as they move among users and users perform their tasks on the workflow.

In addition to the data on which workflows operate, the database may store data about the workflow itself, including the tasks making up a workflow and how they are to be routed among users.

Other Issues in Database Design:

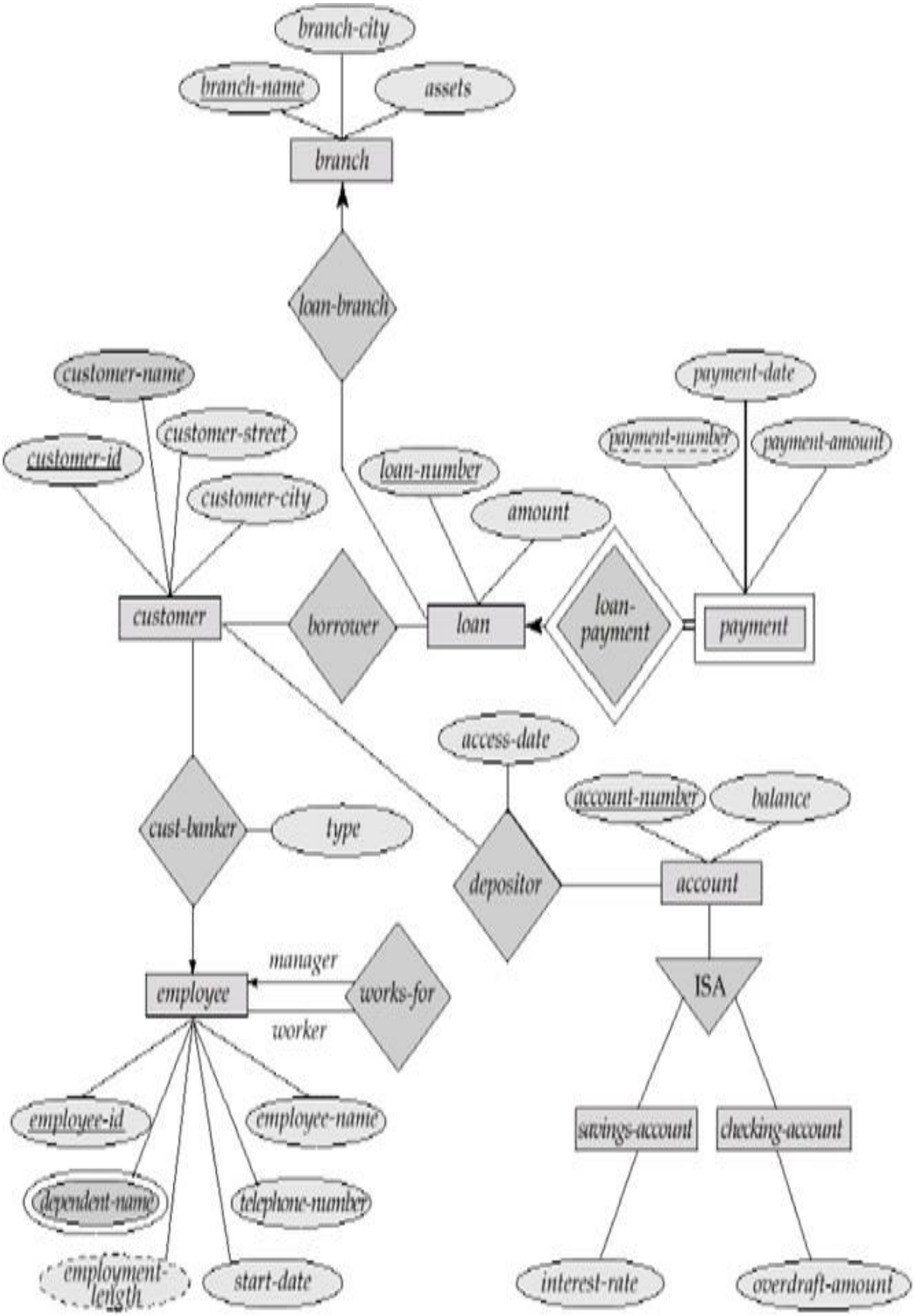
Database design is usually not a one-time activity. The needs of an organization evolve continually, and the data that it needs to store also evolve correspondingly. During the initial database-design phases, the database designer may realize that changes are required at the conceptual, logical, or physical schema levels. Changes in the schema can affect all aspects of the database application. A good database design anticipates future needs of an organization, and ensures that the schema requires minimal changes as the needs evolve.

A good design should account not only for current policies, but should also avoid or minimize changes due to changes that are anticipated, or have a reasonable chance of happening.

Furthermore, the enterprise that the database is serving likely interacts with other enterprises and, therefore, multiple databases may need to interact. Conversion of data between different schemas is an important problem in real-world applications. The XML data model is widely used for representing data when it is exchanged between different applications.

Finally, it is worth noting that database design is a human-oriented activity in two senses: the end users of the system are people (even if an application sits between the database and the end users); and the database designer needs to interact extensively with experts in the application domain to understand the data requirements of the application. All the people involved with the data should be taken into account for a database design and deployment to succeed within the enterprise.

DATABASE DESIGN FOR BANKING ENTERPRISE



Normalization of Database

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

- Eliminating redundant (useless) data.
- Ensuring data dependencies make sense i.e data is logically stored to maintain data consistency.

Functional Dependencies

Definition: Functional dependency is a relationship that exists when one attribute uniquely determines another attribute. It is a relationship between attributes of a table dependent on each other. It helps in preventing data redundancy and gets to know about bad designs**.** Functional dependencies are constraints on the set of legal relations.

Syntax: A -> B

Ex: Account no -> Balance for account table.

Problem without Normalization:

Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anamolies are very frequent if Database is not normalized.

- **Updation Anamoly:** We may want to update data in record, but it may exist in different tables or at different places due to redundancy.
- **Insertion Anamoly:** We may try to insert a new record but data may not be fully available or that record itself doesn't exist. Eg: to insert student details without knowing his department leads to Insertion Anamoly.
- **Deletion Anamoly:** When we try to delete a record it might have been saved or exists in some other place of database due to redundancy.

Normalization Types:

Normalization types are divided into following normal forms.

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. Fifth Normal Form

First normal form (1NF):

A relation is in first normal form if every attribute in every row can contain an atomic (only one single) value. An attribute (column) of a table cannot hold multiple values. It should old

Students

FirstName	LastName	Knowledge
Thomas	Mueller	Java, C++, PHP
Ursula	Meier	PHP, Java
Igor	Mueller	C++, Java

Startsituation

.....

Result after Normalisation

Students

FirstName	LastName	Knowledge
Thomas	Mueller	C++
Thomas	Mueller	PHP
Thomas	Mueller	Java
Ursula	Meier	Java
Ursula	Meier	PHP
Igor	Mueller	Java
Igor	Mueller	C++

only atomic values.

Domain is atomic if its elements are considered to be indivisible units Examples of non-atomic domains:

- Set of names, composite attributes
- Identification numbers like CS101 that can be broken up into parts

A relational schema R is in first normal form if the domains of all attributes of R are atomic. Non-atomic values complicate storage and encourage redundant storage of data.

- We assume all relations are in first normal form

Second normal form (2NF): A database is in second normal form if it satisfies the following conditions:

- It is in first normal form
- All non-prime attributes are fully functional dependent on the primary key An attribute that is not part of any candidate key is known as non-prime attribute.

A table that is in 1st normal form and contains only a single key as the primary key is automatically in 2nd normal form.

Example: Suppose a school wants to store the data of teachers and the subjects they teach.

They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

teacher_id	subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

Candidate Keys: {teacher_id, subject}

Non prime attribute: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "**no** non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this:

teacher_details table:

teacher_id	teacher_age
111	38
222	38
333	40

teacher_subject table:

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

Third Normal Form(3NF): A relation schema *R* is in **third normal form** (3NF) if for all: in *F*+ with at least one of the following holds:

1. is trivial
2. is a superkey for *R*
3. Each attribute *A* in is contained in a candidate key for *R.*

(**NOTE***:* each attribute may be in a different candidate key)

If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold). And, Third condition is a minimal relaxation of BCNF to ensure dependency preservation.

Boyce-Codd Normal Form(BCNF):

A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if for all functional dependencies in *F*+ of the form

where *R* and *R*, at least one of the following holds:

1. is trivial
2. is a superkey for *R*

Example schema *not* in BCNF: *instr_dept* (*ID, name, salary, dept_name, building,*

budget) because *dept_name*□ *building, budget* holds on *instr_dept,* but *dept_name* is

not a super key **Decomposing a Schema into BCNF**

Suppose we have a schema *R* and a non-trivial dependency causes a violation of BCNF.

We decompose *R* into:

In our example,

dept_name

building, budget

and *inst_dept* is replaced by

BCNF and Dependency Preservation

Constraints, including functional dependencies, are costly to check in practice unless they

pertain to only one relation. If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving.* Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form.*

How good is BCNF?

There are database schemas in BCNF that do not seem to be sufficiently normalized Ex:1 Consider a relation

inst_info (ID, child_name, phone)

where an instructor may have more than one phone and can have multiple children

ID	*child_name*	*phone*
99999 99999 99999 99999	David David William Willian	512-555-1234 512-555-4321 512-555-1234 512-555-4321

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples

  (99999, David, 981-992-3443)
  (99999, William, 981-992-3443)
- Therefore, it is better to decompose *inst_info* into:

inst_child

ID	*child_name*
99999 99999 99999 99999	David David William Willian

inst_phone

ID	*phone*
99999 99999 99999 99999	512-555-1234 512-555-4321 512-555-1234 512-555-4321

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF)

Ex: Consider a database: *classes* (*course, teacher, book*) such that (*c, t, b*) □ *classes* means that *t*
is qualified to teach *c*,and *b* is a required textbook for *c*
The database is supposed to list for each course the set of teachers any one of which can be

the course's instructor, and the set of books, all of which are required for the course (no matter who teaches it).

course	Teacher	book
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Stallings
operating systems	Pete	OS Concepts
operating systems	pete	Stallings

Classes

There are no non-trivial functional dependencies and therefore the relation is in BCNF

Insertion anomalies – i.e., if Marilyn is a new teacher that can teach database, two tuples need to be inserted

(database, Marilyn, DB Concepts) (database, Marilyn, Ullman)

Therefore, it is better to decompose *classes* into:

Teaches:

Course	teacher
database	Avi
database	Hunk
database	Sudarshan
operating systems	Avi
operating systems	Jim

Text:

Course	Book
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF),

Fourth Normal Form (4NF): A relation schema *R* is in **4NF** with respect to a set *D* of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form α->β

, where α€ *R* and β€*R*, at least one of the following hold:

α->β is trivial

α is a superkey for schema *R*

- If a relation is in 4NF it is in BCNF

Further Normal Forms

- **Join dependencies** generalize multivalued dependencies

lead to **project-join normal form (PJNF)** (also called **fifth normal form**)

- A class of even more general constraints, leads to a normal form called **domain-key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.
- Hence rarely used

Goals of Normalization

Let *R* be a relation scheme with a set *F* of functional dependencies.

- Decide whether a relation scheme *R* is in "good" form.
- In the case that a relation scheme *R* is not in "good" form, decompose it into a set of relation scheme {*R*1, *R*2, ..., *Rn*} such that
  - each relation scheme is in good form
  - the decomposition is a lossless-join decomposition
  - Preferably, the decomposition should be dependency preserving.

Domain Key Normal Form

It is basically a process in database to organize data efficiently. Basically there are two goals of doing normalization these are as follows:

1. To remove repeated data or in simple words we can say to remove redundant data.
2. Second one is to ensure that there will be data dependencies.

Steps can be done to achieve Normalization:

1. Remove repeating groups or to eliminate repeating groups.
2. Eliminate or remove repeating data.
3. Remove those columns that are not dependent on Key.
4. Multiple relationship should be isolated independently.
5. Isolate Semantically Related Multiple Relationships

There are several types of normal forms, a lower numbered normal form always weaker than the higher numbered normal form. For example, 1st normal form is weaker than that of 2nd normal form. These are as: 1st, 2nd, 3rd, Boyce-code normal form, 4th, 5th, and domain key normal form. But, in this article, we will discuss only about Domain-Key Normal Form.

Domainkeynormalform(DKNF) –

There is no Hard and fast rule to define normal form up to 5NF. Historically the process of normalization and the process of discovering undesirable dependencies were carried through 5NF, but it has been possible to define the stricter normal form that takes into account additional type of dependencies and constraints.

The basic idea behind the *DKNF* is to specify the normal form that takes into account all the possible dependencies and constraints.

In simple words, we can say that DKNF is a normal form used in database normalization which requires that the database contains no constraints other than domain constraints and key constraints.

In other words, a relation schema is said to be in DKNF only if all the constraints and dependencies that should hold on the valid relation state can be enforced simply by enforcing the domain constraints and the key constraints on the relation. For a relation in DKNF, it becomes

very straight forward to enforce all the database constraints by simply checking that each attribute value is a tuple is of the appropriate domain and that every key constraint is enforced.

Reason to use DKNF are as follows:

1. To avoid general constraints in the database that are not clear key constraints.
2. Most database can easily test or check key constraints on attributes.

However, because of the difficulty of including complex constraints in a DKNF relation its practical utility is limited means that they are not in practical use, since it may be quite difficult to specify general integrity constraints.

Let's understand this by taking an example:

Example –
Consider relations CAR (MAKE, vin#) and MANUFACTURE (vin#, country), Where vin# represents the vehicle identification number 'country' represents the name of the country where it is manufactured.
A general constraint may be of the following form:

If the MAKE is either 'HONDA' or 'MARUTI' then the first character of the vin# is a 'B' If the country of manufacture is 'INDIA' If the MAKE is 'FORD' or 'ACCURA', the second character of the vin# is a 'B" if the country of manufacture is 'INDIA'.

There is no simplified way to represent such constraints short of writing a procedure or general assertion to test them. Hence such a procedure needs to enforce an appropriate integrity constraint. However, transforming a higher normal form into domain/key normal form is not always a dependency-preserving transformation and these are not possible always.

Denormalization

Denormalization is a database optimization technique where we add redundant data in the database to get rid of the complex join operations. This is done to speed up database access speed. Denormalization is done after normalization for improving the performance of the database. The data from one table is included in another table to **reduce the number of joins** in the query and hence helps in speeding up the performance.

A denormalized database should never be confused by a database that has never been normalized.
Example: Suppose after normalization we have two tables first, Student table and second, Branch table. The student has the attributes as Roll_no, Student-name, Age, and Branch_id.

Student table

Roll_no	Student_name	Age	Branch_id
1	Andrew	18	10
2	Angel	19	10
3	Priya	20	10
4	Analisa	21	11
5	Anna	21	12

The branch table is related to the Student table with Branch_id as the foreign key in the Student table.

Branch table

Branch_id	Branch_name	HOD
10	CSE	Mr.abc
11	EC	Dr.xyz
12	EX	Dr.pqr

If we want the name of students along with the name of the branch name then we need to perform a join operation. The problem here is that if the table is large we need a lot of time to perform the join operations. So, we can add the data of Branch_name from Branch table to the Student table and this will help in reducing the time that would have been used in join operation and thus optimize the database.

Advantages of De-normalization

1. Query execution is fast since we have to join fewer tables.

Disadvantages of De-normalization

1. As data redundancy is there, update and insert operations are more expensive and take more time. Since we are not performing normalization, so this will result in redundant data.
2. Data Integrity is not maintained in de-normalization. As there is redundancy so data can be inconsistent.

This is all about de-normalization. Hope you learned something new today.