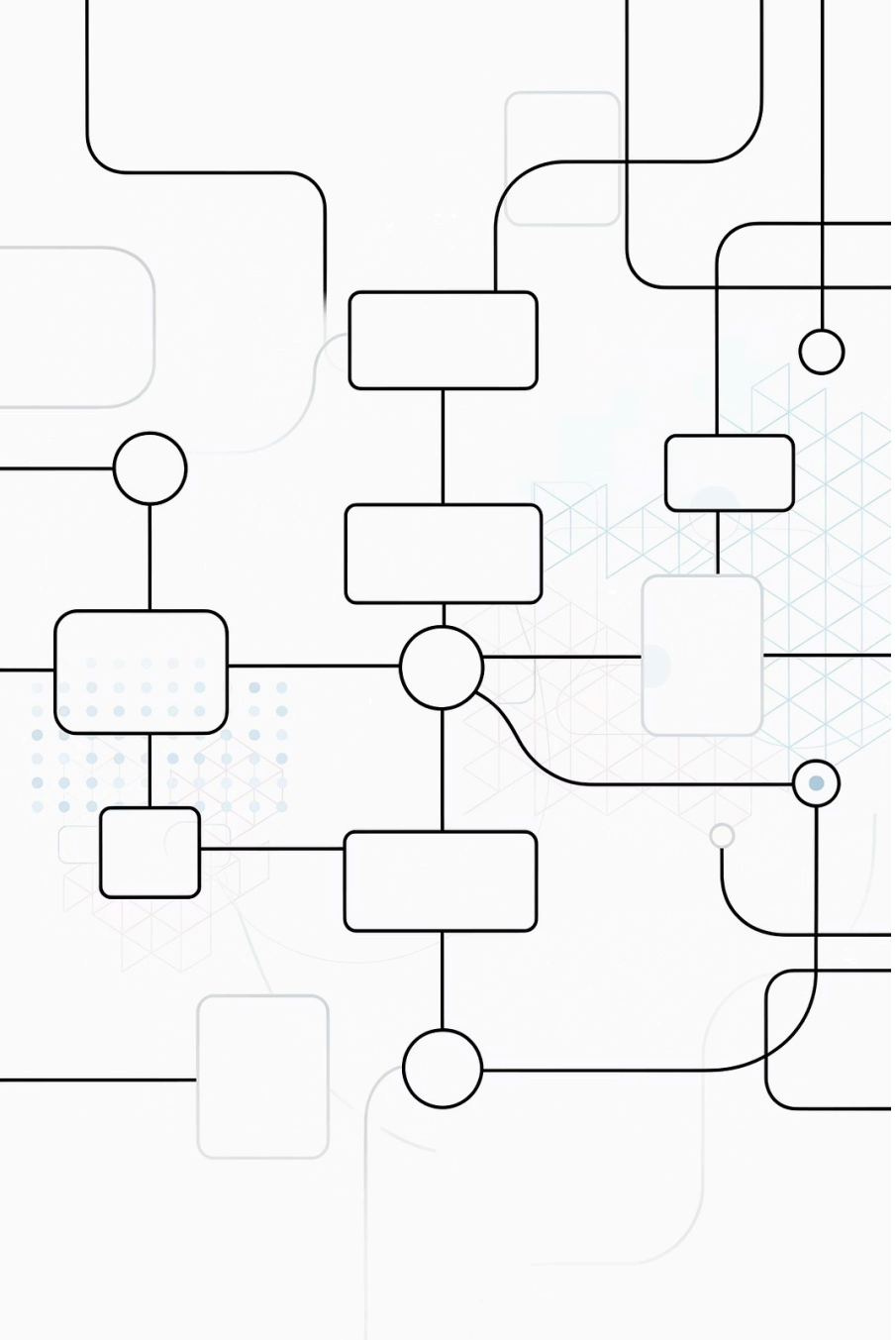


ER Model and Relational Algebra

Database Management System - Unit II

DIPLOMA ENGINEERING

INFORMATION & COMMUNICATION TECHNOLOGY

Course Overview

Program

Diploma Engineering

Branch

Information & Communication Technology

Subject Code

DI04032011

Subject

Database Management System

This unit focuses on Entity-Relationship modeling and the transformation of conceptual designs into functional database structures. Understanding these concepts is fundamental to designing efficient, scalable database systems.

Unit II Learning Objectives

01

Master ER Model Fundamentals

Understand entities, attributes, and relationships that form the foundation of database design

02

Apply Mapping Cardinality

Learn to define and implement one-to-one, one-to-many, and many-to-many relationships

03

Design ER Diagrams

Create comprehensive visual representations of database structures using standard notation

04

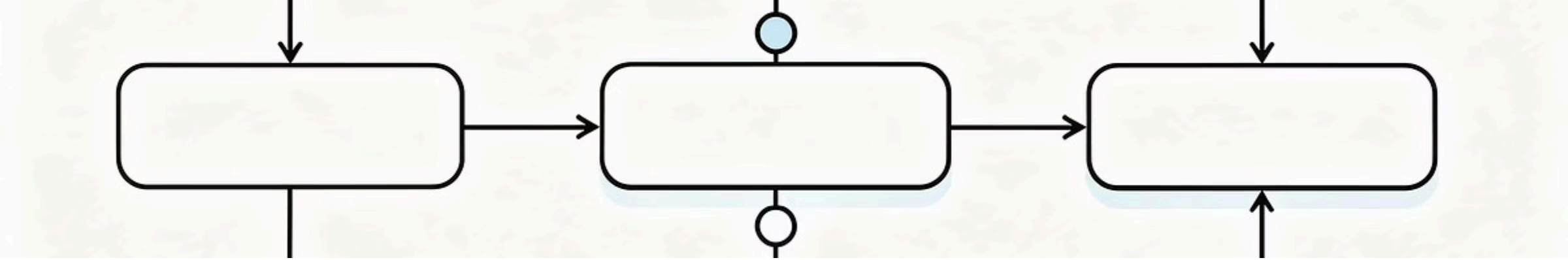
Implement Enhanced Features

Work with advanced concepts including generalization, specialization, and aggregation

05

Convert to Database Schema

Transform ER diagrams into functional relational database implementations



CHAPTER 2.1

Basic Concepts of Entity-Relationship Model

The Entity-Relationship (ER) model is a high-level conceptual data model that helps define the data elements and relationships for a specified system. Developed by Peter Chen in 1976, it provides a graphical representation of the logical structure of databases.

The ER model serves as a blueprint for database design, allowing developers and stakeholders to visualize how data is organized and related before implementation. This approach reduces errors, improves communication, and ensures the database meets business requirements effectively.

The Three Pillars of ER Modeling

Entities

Objects or things in the real world that are distinguishable from other objects

Attributes

Properties or characteristics that describe entities

Relationships

Associations or connections between two or more entities

These three fundamental components work together to create a complete representation of the data structure. Mastering each pillar is essential for effective database design and ensures that all aspects of the real-world scenario are accurately captured in the model.

SECTION 2.1.1

Understanding Entities

What is an Entity?

An entity is a real-world object or concept that can be distinctly identified and about which data can be stored. Entities represent the "nouns" of your database—the things you want to track.

Key Characteristics:

- Must be distinguishable from other objects
- Has a set of associated attributes
- Can be physical or conceptual
- Represented by rectangles in ER diagrams

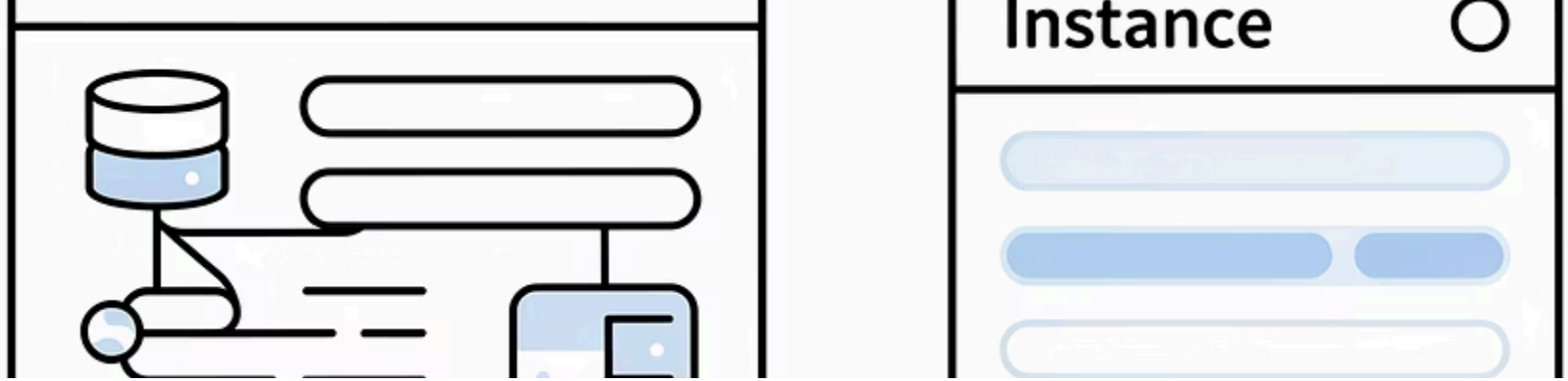
Entity Examples

Physical Entities:

- Student
- Employee
- Product
- Vehicle

Conceptual Entities:

- Course
- Department
- Order
- Account



Entity Types vs Entity Instances

Entity Type

A category or classification of entities that share common properties. It defines the structure and attributes that all instances will have.

Example: STUDENT is an entity type that represents all students in a college database.

Entity Instance

A specific occurrence or example of an entity type. Each instance has unique values for its attributes.

Example: "John Smith, ID: 12345" is a specific instance of the STUDENT entity type.

Think of entity types as the template or blueprint, while entity instances are the actual data entries that follow that template. A single entity type can have thousands or millions of instances in a real database.

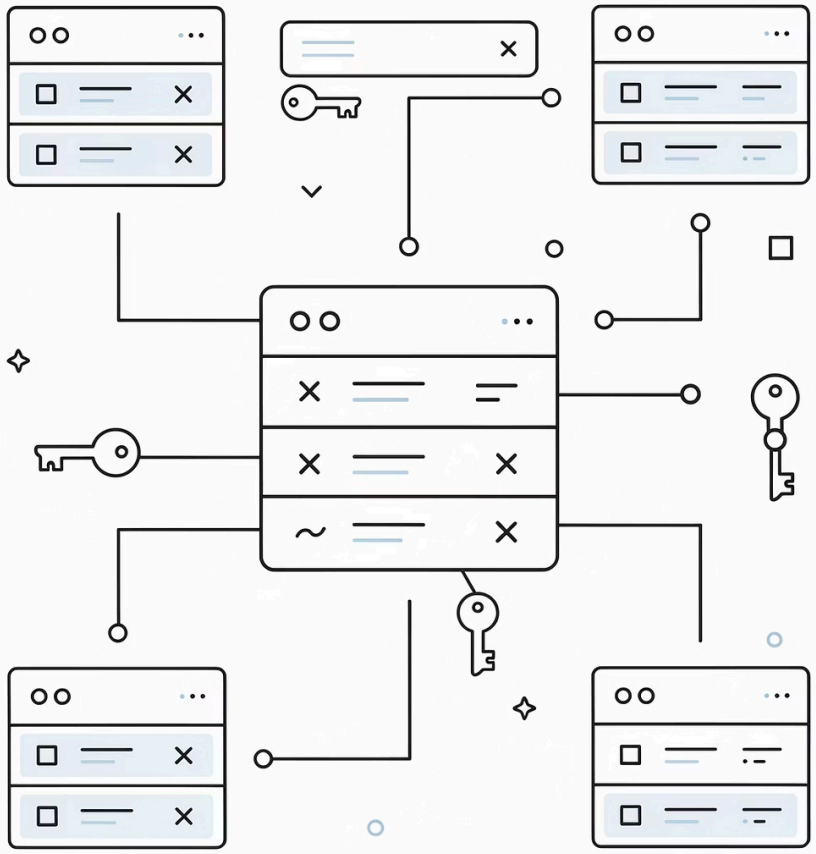
SECTION 2.1.2

Attributes: Describing Entity Properties

Attributes are the properties or characteristics that describe an entity. They represent the data we want to store about each entity instance. Every entity type has a set of attributes that define what information is captured.

For example, a STUDENT entity might have attributes such as Student_ID, Name, Date_of_Birth, Email, and Phone_Number. Each attribute holds a specific piece of information that helps fully describe the student.

Choosing the right attributes is crucial for database design. Include too few, and you won't capture necessary information. Include too many, and you create unnecessary complexity and storage overhead.



Types of Attributes

Simple Attributes

Atomic attributes that cannot be divided further. Examples: Age, Gender, Student_ID

Composite Attributes

Attributes that can be divided into smaller sub-parts. Example: Full_Name (First_Name, Middle_Name, Last_Name) or Address (Street, City, State, ZIP)

Multi-valued Attributes

Attributes that can have multiple values for a single entity. Examples: Phone_Numbers, Email_Addresses, Skills

Derived Attributes

Attributes whose values can be calculated from other attributes. Examples: Age (from Date_of_Birth), Total_Price (from Quantity × Unit_Price)

Attribute Properties and Domain

Key Concepts

Each attribute has specific properties that define how it stores and represents data:

- **Domain:** The set of allowed values for an attribute (e.g., Age domain might be 0-120)
- **Null Values:** Whether an attribute can be empty or must have a value
- **Default Values:** Predefined values assigned when no specific value is provided
- **Data Type:** The kind of data stored (integer, string, date, boolean, etc.)

Understanding these properties ensures data integrity and prevents invalid data entry. For instance, defining Age as an integer with domain 0-120 prevents nonsensical values like -5 or 500.

Domain Example

Attribute: Grade

Domain: {'A', 'B', 'C', 'D', 'F'}

Data Type: Character

Null Allowed: No

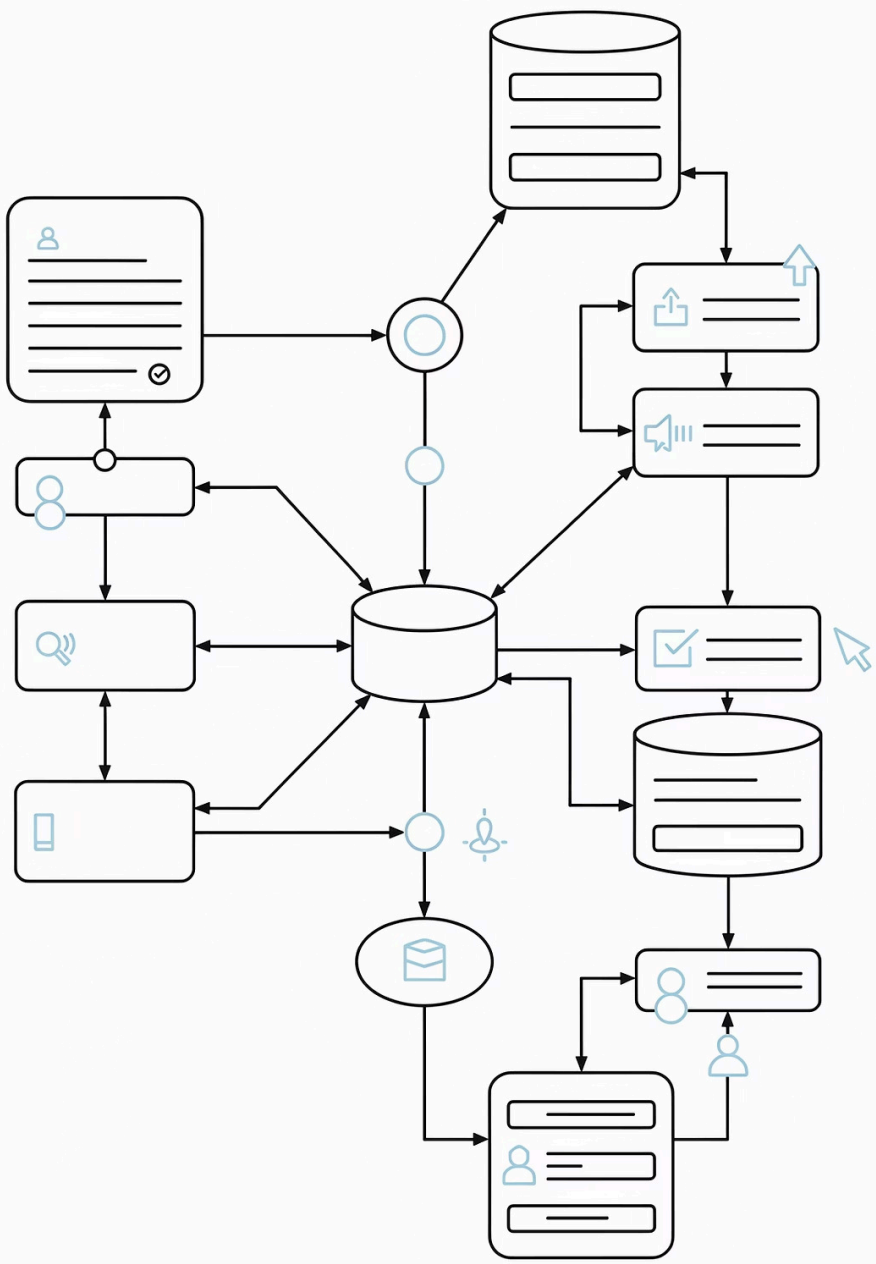
SECTION 2.1.3

Relationships: Connecting Entities

A relationship is an association or connection between two or more entities. Relationships capture how entities interact with each other in the real world and are fundamental to understanding the structure of your data.

For example, a STUDENT entity may be related to a COURSE entity through an "enrolls in" relationship. Similarly, an EMPLOYEE entity might be related to a DEPARTMENT entity through a "works in" relationship.

Relationships are represented by diamond shapes in ER diagrams and are named using verbs that describe the association. The relationship name should clearly convey the nature of the connection between entities.

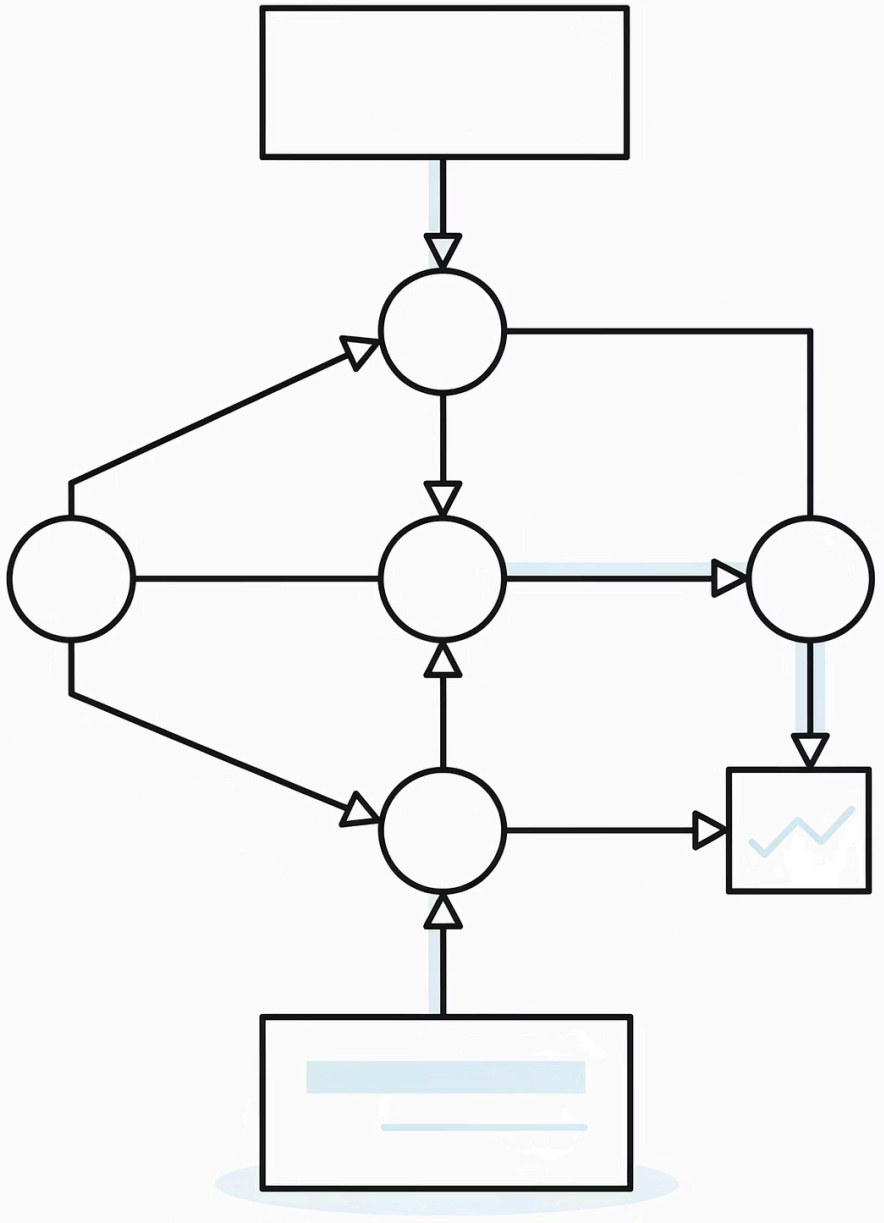


Relationship Examples

1	2	3
Student	**Enrolls In**	**Course**
Entity	Relationship	Entity

1	2	3
Employee	**Works In**	**Department**
Entity	Relationship	Entity

1	2	3
Author	**Writes**	**Book**
Entity	Relationship	Entity



SECTION 2.2.3.1

Participation Constraints

Participation constraints define whether all instances of an entity must participate in a relationship or if participation is optional. This concept is crucial for maintaining data integrity and understanding mandatory versus optional associations.

Total Participation

Every instance of the entity must participate in the relationship. Represented by a double line in ER diagrams.

Example: Every STUDENT must be enrolled in at least one COURSE. No student can exist without course enrollment.

Partial Participation

Some instances may not participate in the relationship. Represented by a single line in ER diagrams.

Example: Not every EMPLOYEE has to manage a PROJECT. Some employees are individual contributors.

SECTION 2.2.3.2

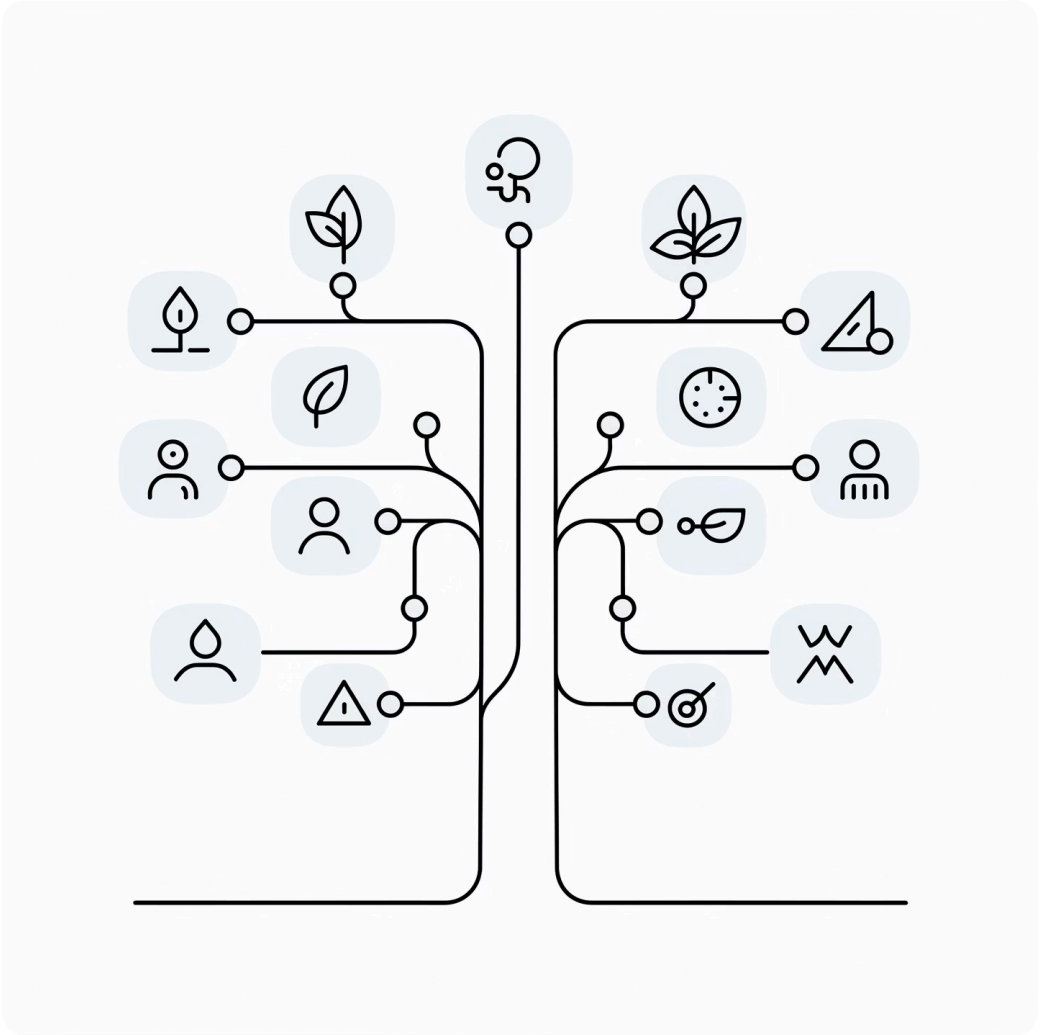
Recursive Relationships

Definition

A recursive relationship (also called a self-referencing relationship) occurs when an entity has a relationship with itself. The same entity type participates in the relationship in different roles.

Common Examples

- **Employee-Supervisor:** An EMPLOYEE supervises other EMPLOYEEs
- **Course Prerequisites:** A COURSE may be a prerequisite for other COURSEs
- **Social Networks:** A PERSON is friends with other PERSONs
- **Organizational Structure:** A DEPARTMENT may be part of another DEPARTMENT



Role Names

In recursive relationships, role names are essential to distinguish the different roles the entity plays. For example, in the Employee-Supervisor relationship, roles would be "supervisor" and "subordinate".

SECTION 2.2.3.3

Degree of Relationship Set

The degree of a relationship refers to the number of entity types that participate in the relationship. Understanding relationship degree is essential for accurate database modeling and determines the complexity of associations between entities.

Binary (Degree 2)

Most common type. Involves two entity types.

Example: STUDENT enrolls in COURSE

Ternary (Degree 3)

Involves three entity types simultaneously.

Example: SUPPLIER supplies PART to PROJECT

N-ary (Degree N)

Involves more than three entity types (rare in practice).

Example: Complex multi-entity business processes

While binary relationships are most common and easiest to understand, ternary and higher-degree relationships are sometimes necessary to accurately model complex real-world scenarios. However, many ternary relationships can be decomposed into multiple binary relationships for simplicity.

Ternary Relationship Example

Consider a scenario where we need to track which SUPPLIER supplies which PART to which PROJECT. This requires a ternary relationship because all three entities are needed to fully describe the business rule.

Supplier

The company providing materials

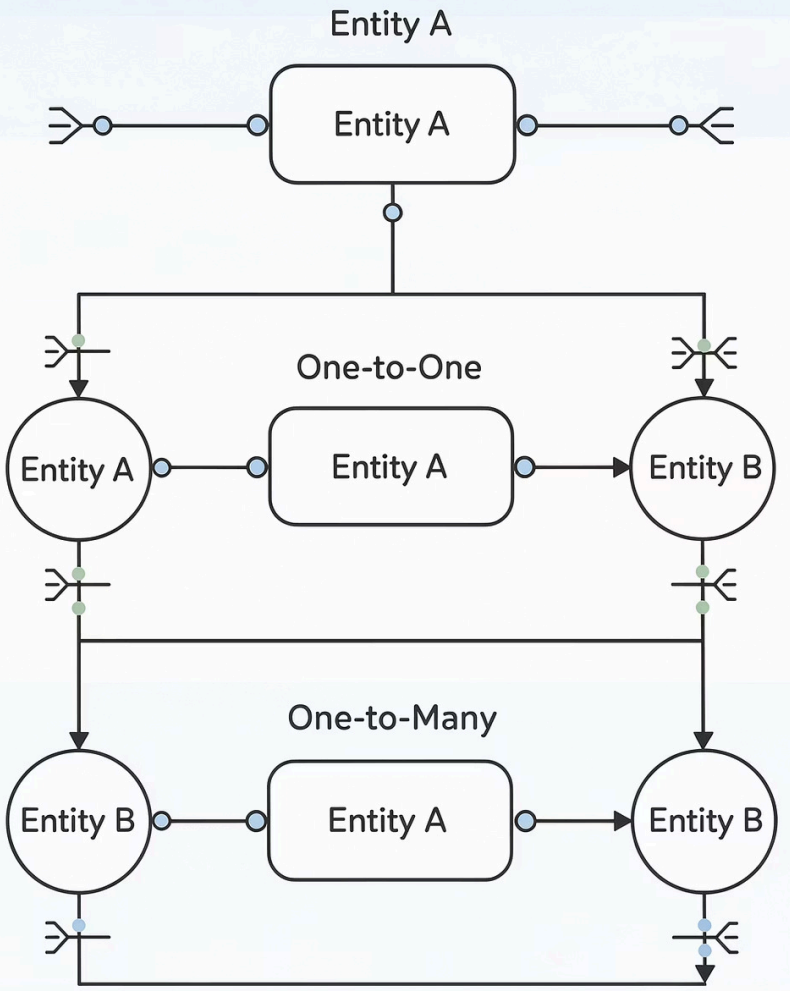
Part

The component being supplied

Project

The construction project receiving the part

Breaking this into binary relationships would lose important information. We need to know specifically which supplier provides which part for which project—all three components together form a single business transaction.



SECTION 2.2

Mapping Cardinality

Mapping cardinality, also called cardinality ratio, specifies the number of entity instances to which another entity instance can be associated through a relationship. It defines the numerical constraints on relationships and is one of the most important aspects of database design.

Cardinality constraints help enforce business rules and ensure data integrity. They determine whether relationships are one-to-one, one-to-many, or many-to-many, which directly impacts how tables are structured in the physical database.

Understanding and correctly implementing cardinality is crucial for preventing data anomalies and maintaining consistency across the database.

Types of Cardinality

1

One-to-One (1:1)

Each instance of Entity A is associated with at most one instance of Entity B, and vice versa.

Example: PERSON and PASSPORT - Each person has one passport, and each passport belongs to one person.

2

One-to-Many (1:N)

Each instance of Entity A can be associated with multiple instances of Entity B, but each instance of B is associated with only one instance of A.

Example: DEPARTMENT and EMPLOYEE - One department has many employees, but each employee belongs to one department.

3

Many-to-One (N:1)

Multiple instances of Entity A can be associated with one instance of Entity B.

Example: STUDENT and ADVISOR - Many students can have the same advisor.

4

Many-to-Many (M:N)

Each instance of Entity A can be associated with multiple instances of Entity B, and vice versa.

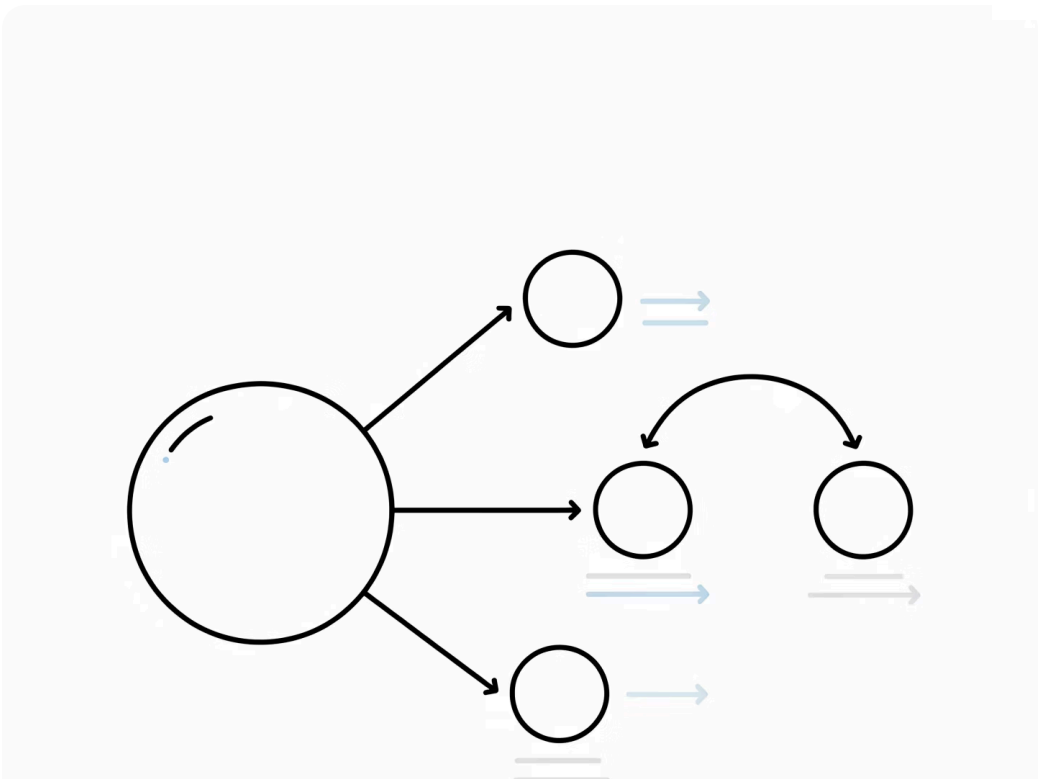
Example: STUDENT and COURSE - Students can enroll in many courses, and courses can have many students.

Cardinality in Practice

1:1 Relationship

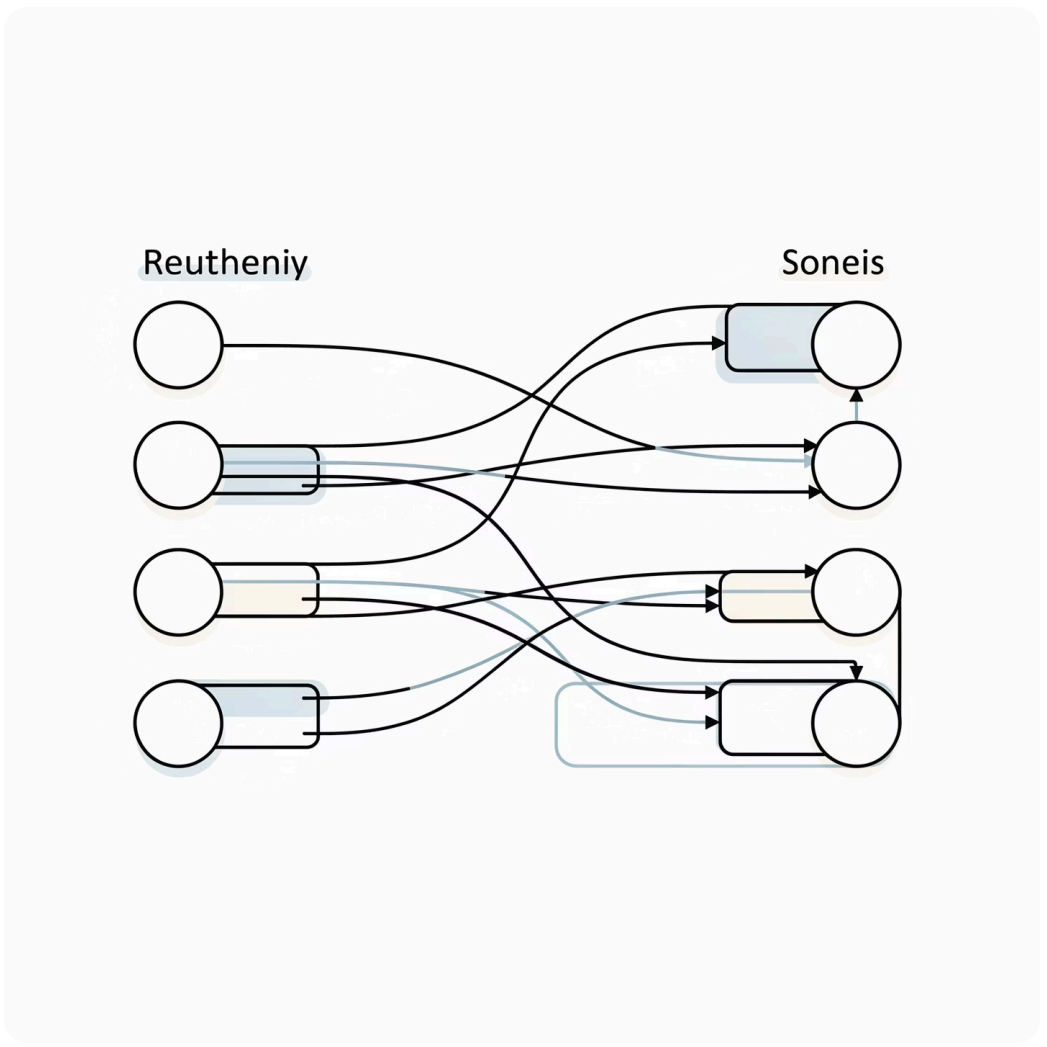
Implementation typically uses a foreign key in either table with a unique constraint, or combines both entities into a single table.

1:N Relationship



Implementation places a foreign key in the "many" side table referencing the "one" side's primary key.

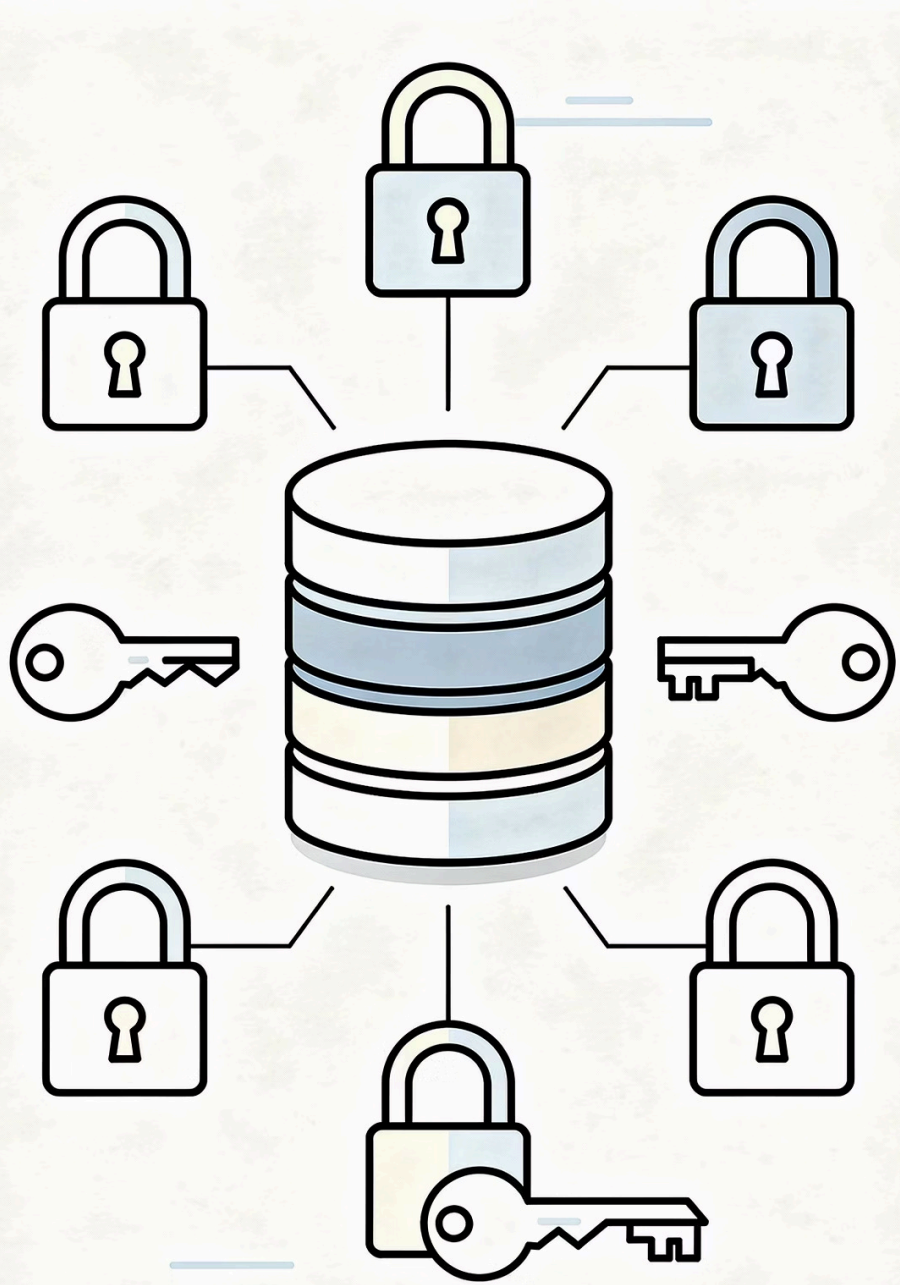
M:N Relationship



Implementation requires a junction table (also called bridge table or associative entity) containing foreign keys from both related tables.

Junction Table Example

STUDENT_COURSE table with Student_ID and Course_ID creates the many-to-many relationship between STUDENT and COURSE tables.



SECTION 2.3

Database Keys

Keys are attributes or sets of attributes that uniquely identify entity instances and establish relationships between tables. They are fundamental to maintaining data integrity, enabling efficient data retrieval, and enforcing constraints in relational databases.

Proper key design ensures that each record can be uniquely identified, prevents duplicate entries, and maintains referential integrity across related tables. Understanding different types of keys is essential for effective database design and normalization.

SECTION 2.3.1

Types of Keys

Primary Key

A unique identifier for each record in a table. Cannot contain NULL values. Each table has exactly one primary key.

Example: Student_ID in STUDENT table

Foreign Key

An attribute that creates a link between two tables by referencing the primary key of another table.

Example: Department_ID in EMPLOYEE table references DEPARTMENT table

Candidate Key

Any attribute or minimal set of attributes that can uniquely identify records. A table can have multiple candidate keys.

Example: Student_ID, Email, or SSN could all be candidate keys

Super Key

Any set of attributes that can uniquely identify records. May contain additional attributes beyond what's necessary.

Example: {Student_ID, Name, Email} is a super key

Key Relationships and Hierarchy

Primary Key

Selected from candidate keys as the main identifier

Candidate Keys

Minimal super keys that uniquely identify records

Super Keys

Any combination of attributes that ensures uniqueness

The hierarchy shows how different key types relate to each other. All primary keys are candidate keys, all candidate keys are super keys, but not all super keys are candidate keys (they may contain redundant attributes).

Primary Key Selection Criteria

Choosing the right primary key is a critical decision in database design. The primary key should be selected from available candidate keys based on specific criteria that ensure long-term stability and efficiency.

Uniqueness

Must uniquely identify each record without any possibility of duplication

Minimality

Should contain the minimum number of attributes necessary for unique identification

Stability

Value should not change over time; avoid attributes that might be updated

Not NULL

Must always have a value; NULL values are not allowed in primary keys

Simplicity

Prefer simple data types (integer) over complex ones (composite attributes)

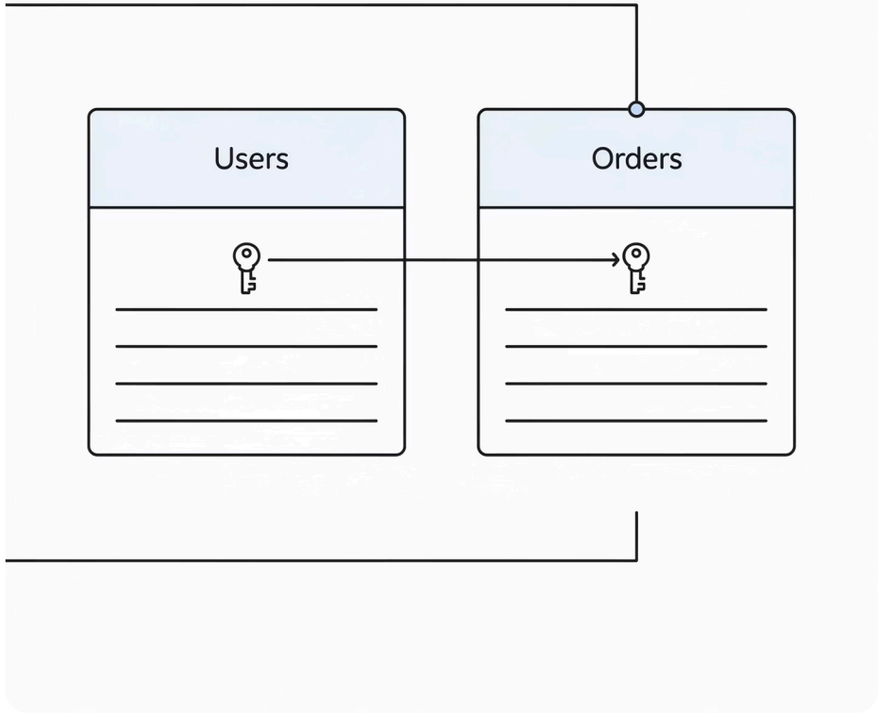
Many modern databases use surrogate keys (auto-incremented integers) as primary keys because they satisfy all these criteria perfectly and are independent of business data.

Foreign Key Constraints

Foreign keys enforce referential integrity between tables, ensuring that relationships remain valid and consistent. They prevent orphaned records and maintain the logical connections between related data.

Referential Integrity Rules

- **Insert Rule:** Cannot insert a record with a foreign key value that doesn't exist in the referenced table
- **Update Rule:** Changes to primary keys must be reflected in all related foreign keys (CASCADE) or prevented
- **Delete Rule:** Deletion of a primary key record requires handling of related foreign key records (CASCADE, SET NULL, or RESTRICT)



CASCADE Operations

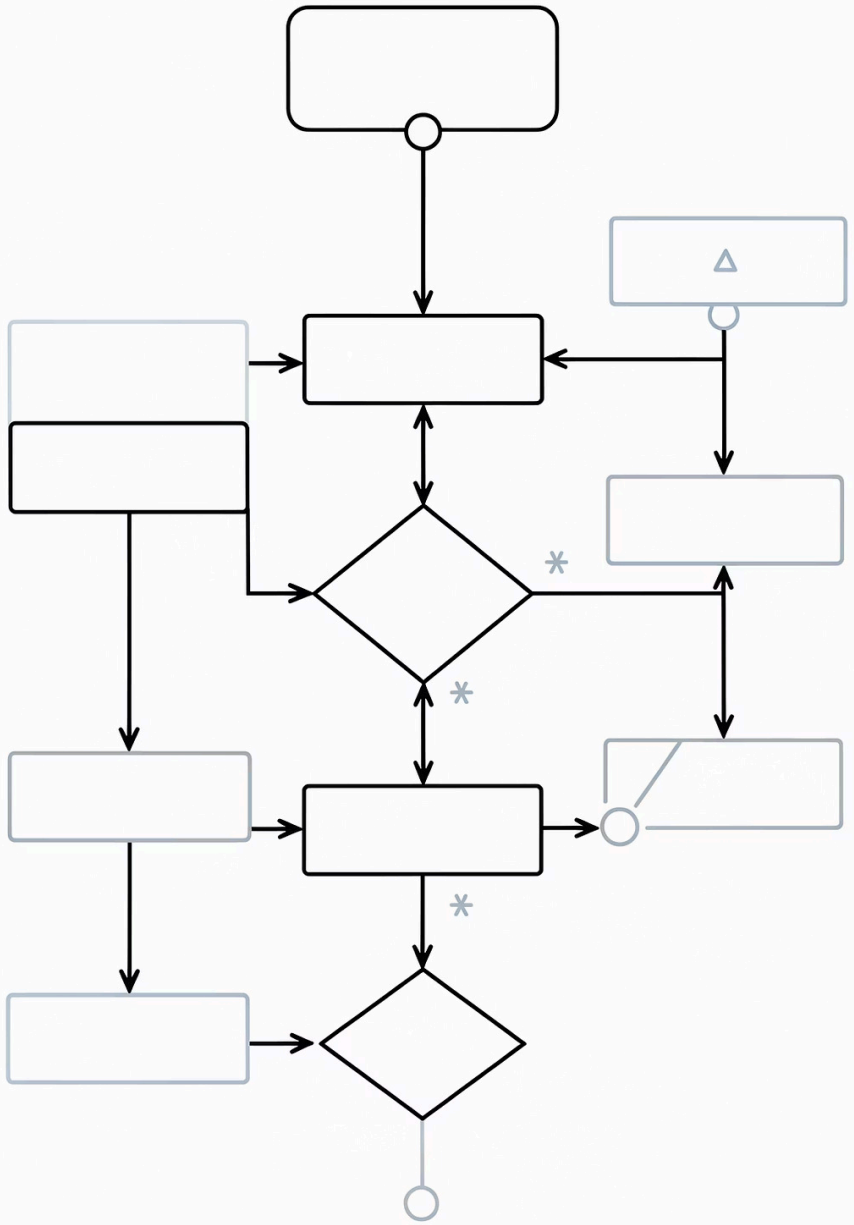
ON DELETE CASCADE automatically deletes related records. ON UPDATE CASCADE automatically updates foreign key values when primary keys change.

SECTION 2.4

Entity-Relationship Diagrams

Entity-Relationship (ER) Diagrams are graphical representations of entities, attributes, and relationships in a database system. They serve as a visual blueprint that communicates database structure to both technical and non-technical stakeholders.

ER diagrams use standardized symbols and notations to represent different components, making them universally understood by database professionals. They are essential tools in the database design process, helping identify potential issues before implementation.



ER Diagram Notation Symbols

Rectangle

Represents an entity type

Example: STUDENT, EMPLOYEE, PRODUCT

Ellipse / Oval

Represents an attribute of an entity

Example: Name, Age, Address

Diamond

Represents a relationship between entities

Example: Enrolls, Works_In, Manages

Line

Connects entities to relationships and attributes to entities

Double Ellipse

Represents multi-valued attributes

Example: Phone_Numbers, Skills

Dashed Ellipse

Represents derived attributes

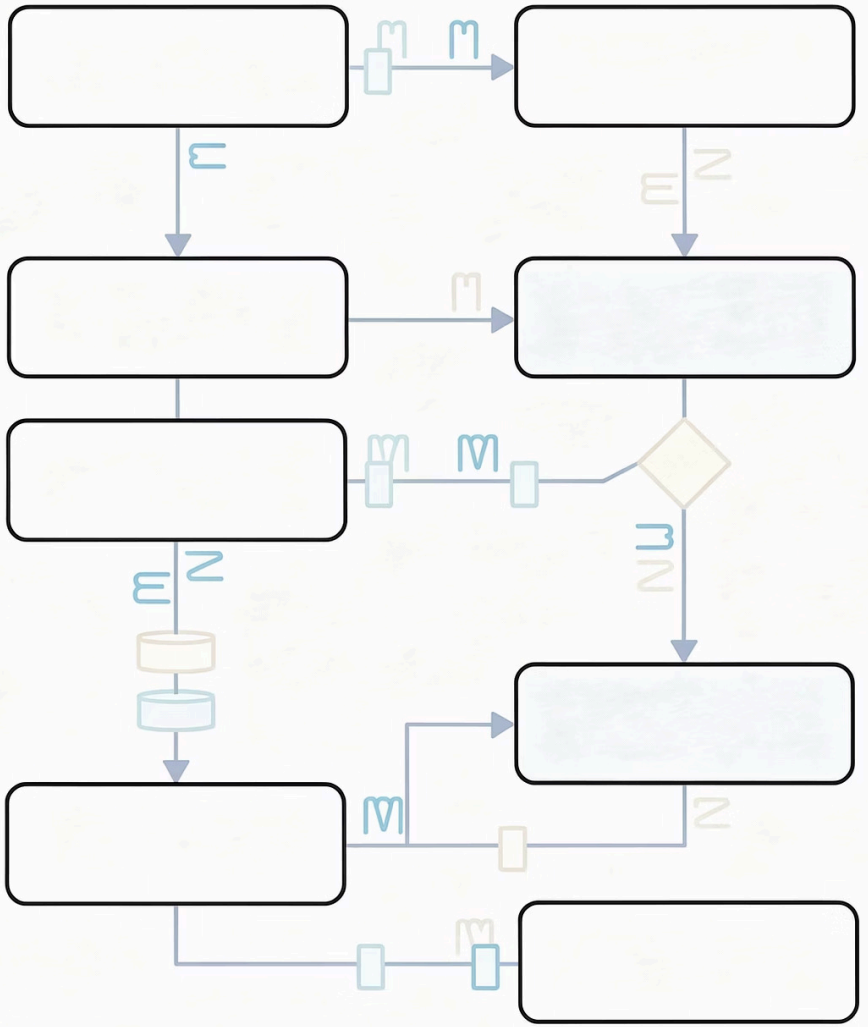
Example: Age (derived from DOB)

Double Rectangle

Represents weak entity types

Double Diamond

Represents identifying relationship for weak entities



Cardinality Notation in ER Diagrams

Different notations exist for representing cardinality in ER diagrams. The two most common are Chen notation and Crow's Foot notation.

Chen Notation

Uses numbers to show cardinality:

- 1 indicates "one"
- N or M indicates "many"
- Placed near the entity on the relationship line

Example: DEPARTMENT (1) ----< employs >---- (N) EMPLOYEE

Crow's Foot Notation

Uses graphical symbols:

- Single line: one
- Crow's foot (three prongs): many
- Circle: optional (zero)
- Perpendicular line: mandatory (one or more)

Choose a notation style and use it consistently throughout your diagram. Crow's Foot notation is more popular in modern database design tools.

Creating Effective ER Diagrams

01

Identify Entities

List all the objects or concepts that need to be stored in the database

02

Define Attributes

Determine what information needs to be stored about each entity

03

Identify Relationships

Determine how entities are related to each other

04

Determine Cardinality

Specify the number constraints for each relationship

05

Identify Keys

Select primary keys and define foreign keys

06

Review and Refine

Check for completeness, accuracy, and potential improvements

Creating an ER diagram is an iterative process. Expect to revise your diagram multiple times as you better understand the requirements and identify edge cases.

Best Practices for ER Diagrams

- **Use clear, descriptive names**

  Entity names should be nouns (singular form), relationship names should be verbs that clearly describe the association

- **Keep it simple and readable**

  Avoid crossing lines when possible, organize entities logically, and don't overcrowd the diagram

- **Document assumptions**

  Include notes about business rules, constraints, or special considerations that aren't obvious from the diagram

- **Validate with stakeholders**

  Review the diagram with users and business analysts to ensure it accurately represents their requirements

- **Use consistent notation**

  Stick to one notation style throughout the entire diagram to avoid confusion

- **Include all relevant constraints**

  Show participation constraints, cardinality, and key attributes clearly

Common ER Diagram Mistakes

Redundant Relationships

Creating multiple relationships that represent the same association between entities, leading to data inconsistency

Missing Relationships

Failing to identify all necessary connections between entities, resulting in incomplete data models

Incorrect Cardinality

Misidentifying relationship constraints, which can lead to data integrity issues in implementation

Attributes as Entities

Mistakenly representing simple attributes as separate entities when they should be properties of existing entities

Missing Primary Keys

Forgetting to identify unique identifiers for entities, making it impossible to distinguish individual instances

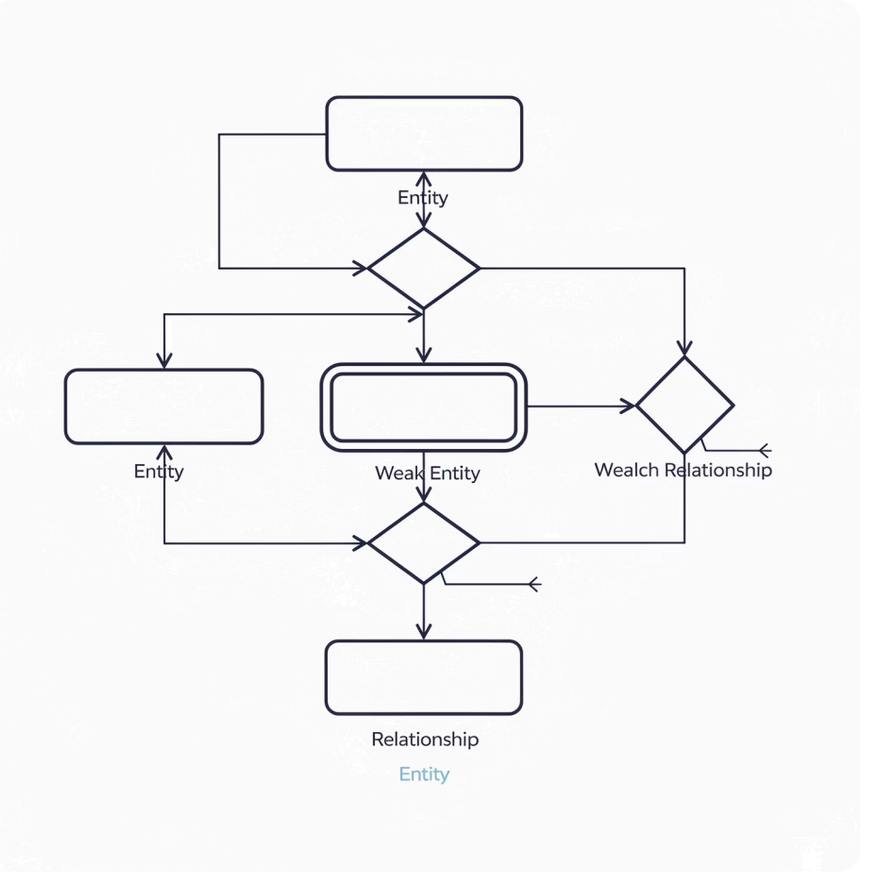
SECTION 2.5

Weak Entity Sets

A weak entity is an entity that cannot be uniquely identified by its own attributes alone. It depends on another entity (called the owner or strong entity) for its existence and identification.

Characteristics

- Does not have a primary key of its own
- Has a partial key (discriminator) that uniquely identifies it within the context of its owner
- Existence depends on its relationship with the strong entity
- Represented by a double rectangle in ER diagrams
- Connected through an identifying relationship (double diamond)



Example

DEPENDENT is a weak entity related to EMPLOYEE. A dependent cannot exist without an employee and is identified by {Employee_ID + Dependent_Name}.

Strong vs Weak Entities

Strong Entity

- Has its own primary key that uniquely identifies each instance independently
- Can exist independently of other entities
- Represented by single rectangle in ER diagrams
- **Examples:** STUDENT, EMPLOYEE, CUSTOMER, PRODUCT

Weak Entity

- Cannot be uniquely identified by its own attributes alone
- Depends on a strong entity for existence and identification
- Represented by double rectangle in ER diagrams
- **Examples:** DEPENDENT, ROOM (in BUILDING), ORDER_ITEM (in ORDER)

Weak Entity Examples

Example 1: Employee and Dependent

An EMPLOYEE (strong entity) has DEPENDENTs (weak entity). Each dependent is identified by the combination of Employee_ID and Dependent_Name. If an employee record is deleted, all associated dependent records are also removed.

Example 2: Building and Room

A BUILDING (strong entity) contains ROOMs (weak entity). Each room is identified by Building_ID and Room_Number. Room 101 in Building A is different from Room 101 in Building B. Rooms cannot exist without a building.

Example 3: Order and Order Item

An ORDER (strong entity) contains ORDER_ITEMs (weak entity). Each item is identified by Order_ID and Item_Number. Order items have no meaning outside the context of their parent order.

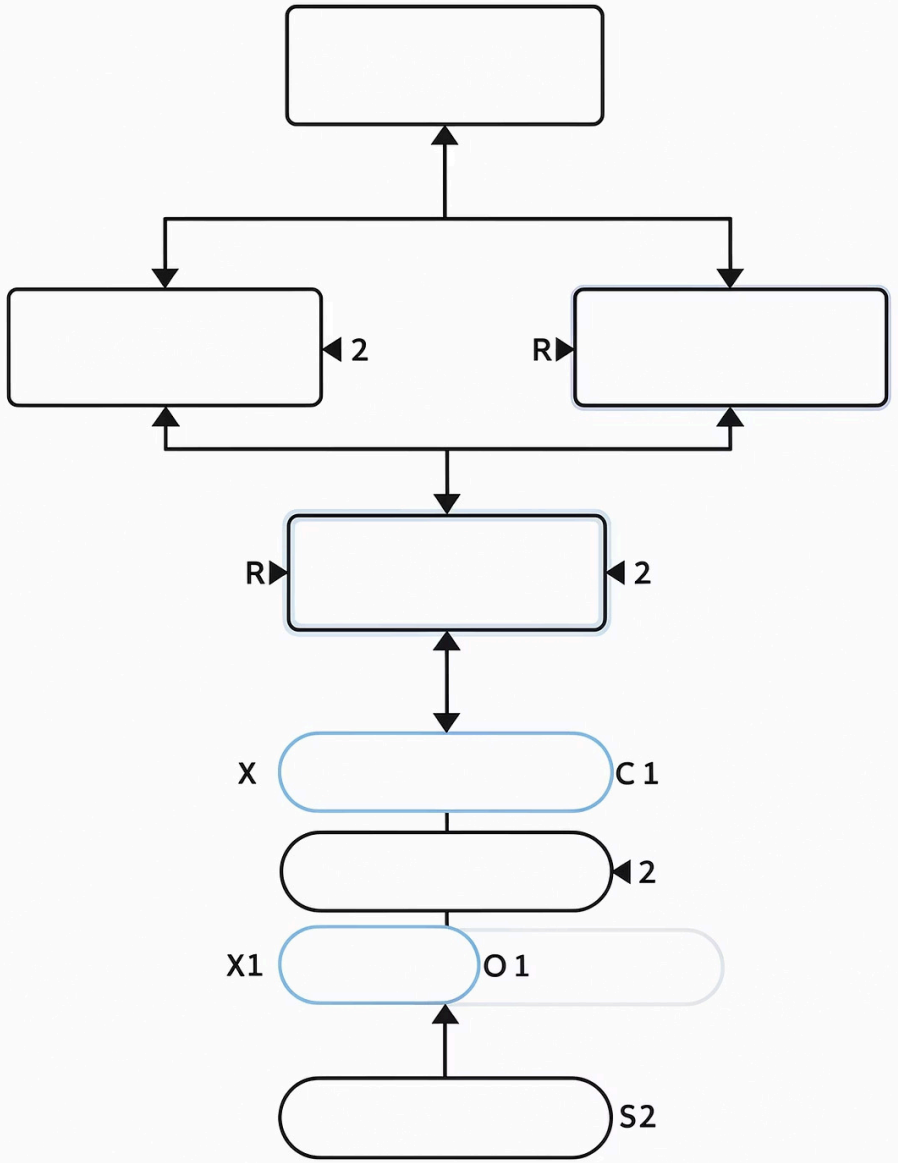
SECTION 2.6

Enhanced Entity-Relationship Model

The Enhanced Entity-Relationship (EER) model extends the basic ER model with additional concepts to better represent complex database requirements. It introduces features that capture more semantic information about the database structure.

EER models are particularly useful for large, complex databases where entities share common characteristics or have hierarchical relationships. These advanced concepts help reduce redundancy and improve the clarity of database designs.

The four key concepts in EER modeling are subclass/superclass relationships, generalization, specialization, and aggregation.



SECTION 2.6.1

Subclass and Superclass

Superclass

A superclass (also called parent class or generalized entity) is an entity type that contains common attributes and relationships shared by multiple specialized entities.

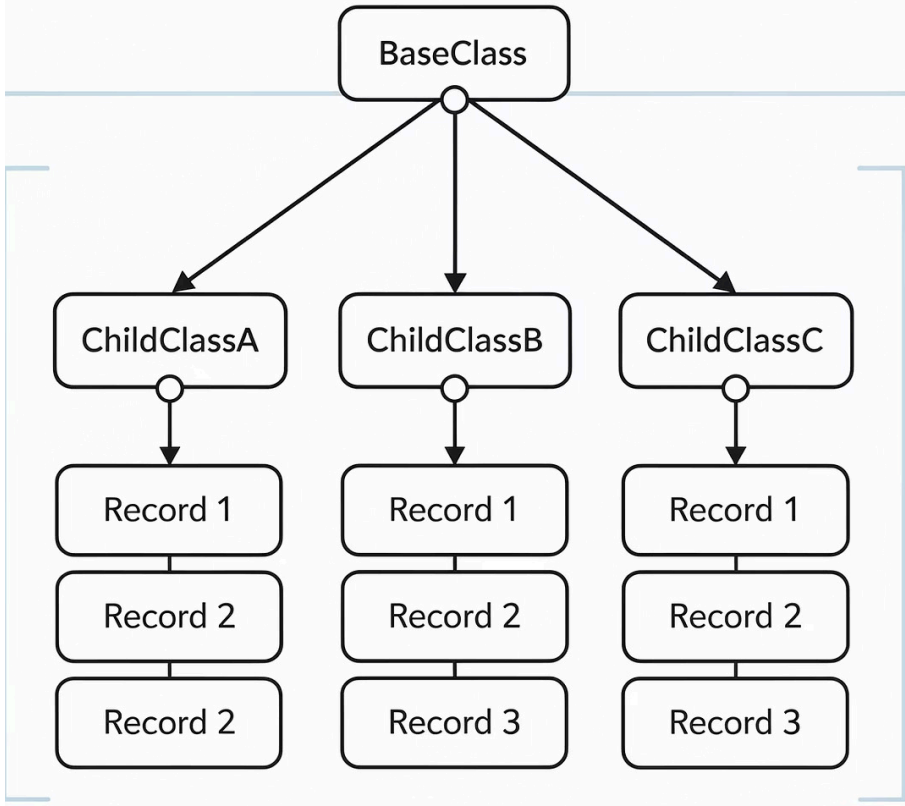
Example: PERSON is a superclass containing attributes like Person_ID, Name, Date_of_Birth, and Address that are common to all people.

Subclass

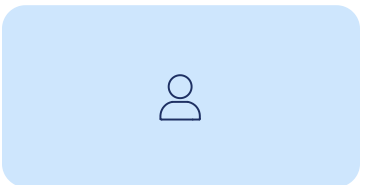
A subclass (also called child class or specialized entity) is an entity type that inherits attributes from a superclass and adds its own specific attributes.

Examples: STUDENT and EMPLOYEE are subclasses of PERSON, adding attributes like Student_ID, GPA or Employee_ID, Salary respectively.

Subclasses inherit all attributes and relationships from their superclass. This inheritance mechanism promotes reusability and reduces redundancy in the database design.

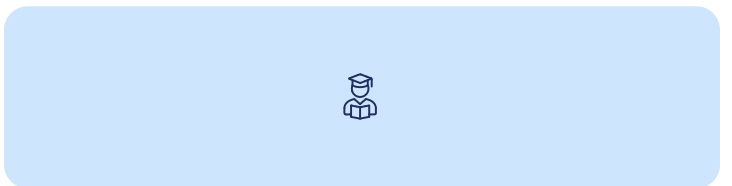


Superclass-Subclass Relationships



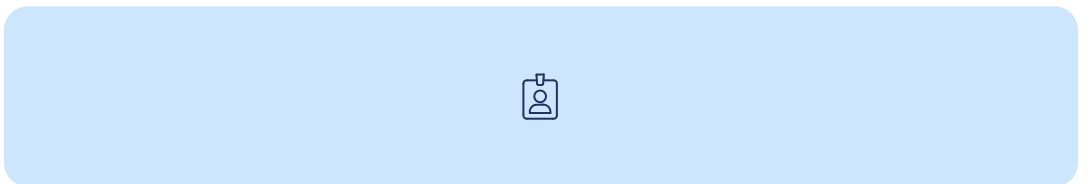
PERSON (Superclass)

Person_ID, Name, Address, Phone



STUDENT (Subclass)

Inherits all PERSON attributes + Student_ID, Major, GPA



EMPLOYEE (Subclass)

Inherits all PERSON attributes + Employee_ID, Department, Salary

The "IS-A" relationship exists between subclass and superclass. Every STUDENT "is a" PERSON, and every EMPLOYEE "is a" PERSON. This means any instance of a subclass is also an instance of the superclass.

SECTION 2.6.2

Generalization

Generalization is the bottom-up process of defining a generalized (superclass) entity from multiple specialized (subclass) entities. It identifies common attributes and relationships among existing entities and creates a higher-level abstraction.

The Generalization Process

1

Identify Similar Entities

Find entities with common attributes

2

Extract Common Features

Move shared attributes to superclass

3

Create Superclass

Define the generalized entity type

Example: Given separate CAR, TRUCK, and MOTORCYCLE entities, we can generalize them into a VEHICLE superclass containing common attributes like Vehicle_ID, Make, Model, and Year, while keeping specific attributes in the subclasses.

SECTION 2.6.3

Specialization

Specialization is the top-down process of defining specialized subclass entities from a general superclass entity. It divides a high-level entity into multiple lower-level entities based on distinguishing characteristics.

The Specialization Process

1

Start with Superclass

Begin with general entity

2

Identify Distinctions

Find differentiating characteristics

3

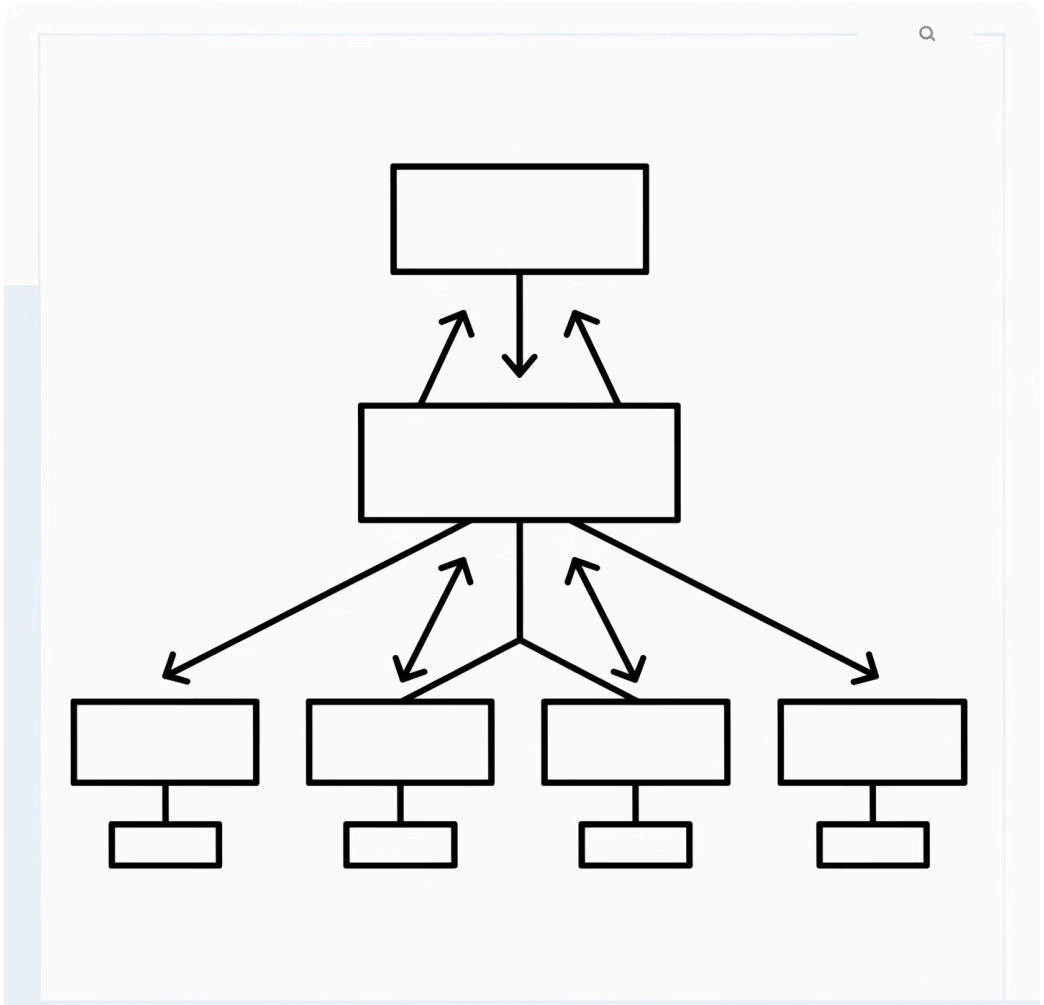
Create Subclasses

Define specialized entity types

Example: Starting with an ACCOUNT entity, we can specialize it into SAVINGS_ACCOUNT and CHECKING_ACCOUNT subclasses, each with specific attributes like Interest_Rate or Overdraft_Limit respectively.

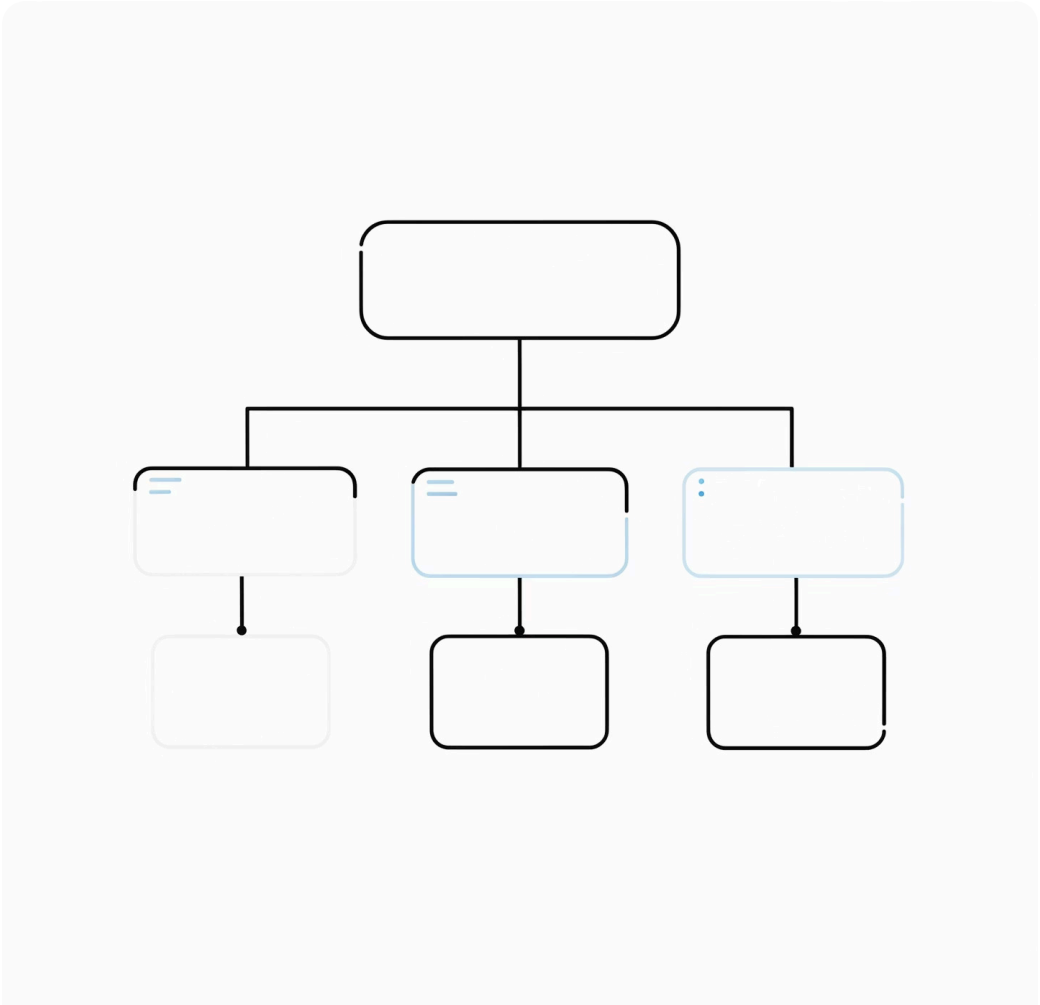
Generalization vs Specialization

Generalization (Bottom-Up)



- Starts with specific subclasses
- Combines them into a general superclass
- Emphasizes commonality
- Reduces redundancy by eliminating duplicate attributes
- **Think:** "What do these entities have in common?"

Specialization (Top-Down)



- Starts with a general superclass
- Divides it into specific subclasses
- Emphasizes differences
- Adds detail through specialized attributes
- **Think:** "What makes these instances different?"

These are complementary processes that achieve the same result: a hierarchy of entities with inheritance relationships. The choice between them depends on your design approach and how you initially identify entities.

Specialization Constraints

Disjoint vs Overlapping

Disjoint: A superclass instance can belong to at most one subclass (represented by 'd' in diagrams)

Example: An EMPLOYEE is either HOURLY or SALARIED, not both

Overlapping: A superclass instance can belong to multiple subclasses (represented by 'o' in diagrams)

Example: A PERSON can be both a STUDENT and an EMPLOYEE

Total vs Partial

Total Specialization: Every superclass instance must belong to at least one subclass (double line)

Example: Every ACCOUNT must be either SAVINGS or CHECKING

Partial Specialization: Some superclass instances may not belong to any subclass (single line)

Example: Not all EMPLOYEES are MANAGERs or TECHNICIANs

SECTION 2.6.4

Aggregation

Aggregation is an abstraction concept that allows treating a relationship as a higher-level entity. It represents the idea that relationships themselves can participate in other relationships.

Aggregation is used when you need to create relationships between a relationship and another entity. This is necessary when the relationship has attributes or participates in other relationships.

When to Use Aggregation

- When a relationship needs to be treated as an entity
- When a relationship participates in another relationship
- When you need to represent relationships between relationships
- To avoid creating redundant entities and relationships

Aggregation Example

Consider a scenario where EMPLOYEES work on PROJECTs, and we need to track which MANAGER supervises each employee-project assignment.

Employee

The person assigned to work

Project

The task being worked on

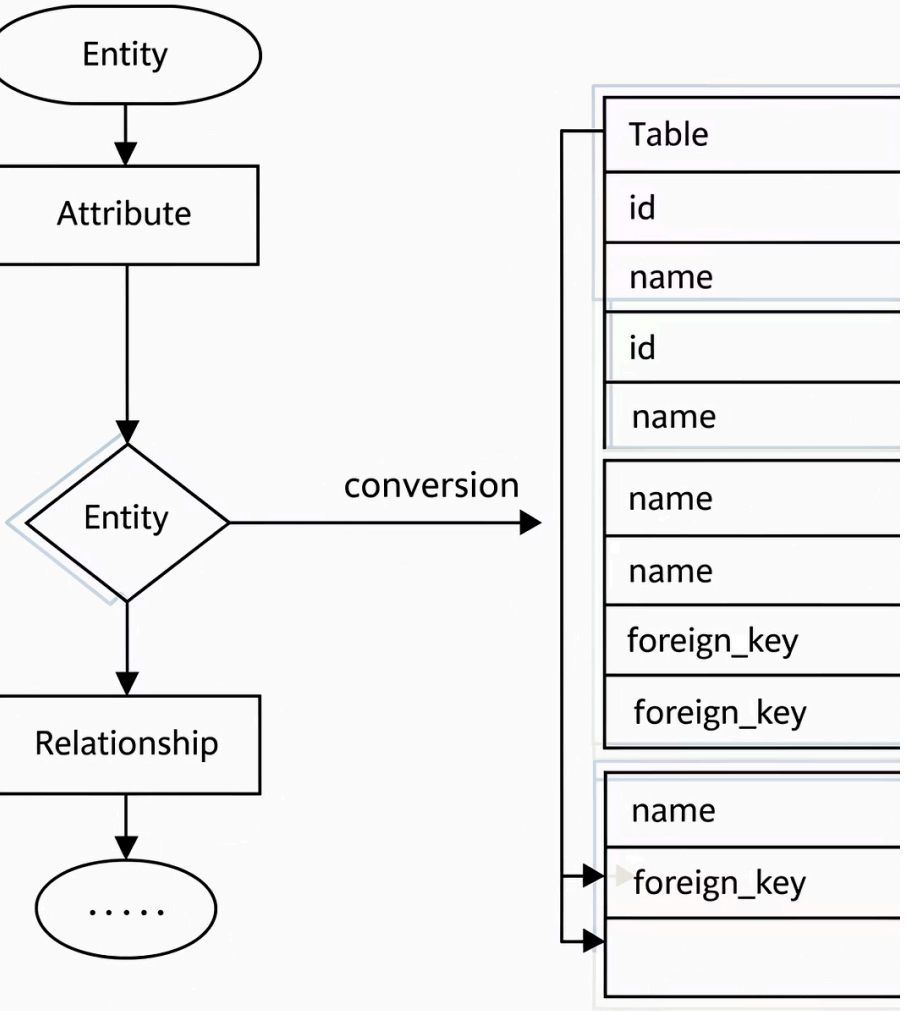
Manager

Supervises the assignment

Works_On Relationship

The assignment (aggregated)

Without aggregation, we would struggle to properly represent that a manager supervises specific employee-project combinations. Aggregation allows us to treat the "Works_On" relationship as an entity that can participate in the "Supervises" relationship.



SECTION 2.7

Converting ER Diagrams to Database Schema

After creating an ER diagram, the next step is converting it into a relational database schema. This transformation process follows systematic rules that map ER components to database tables, columns, and constraints.

The conversion process is crucial because it bridges the gap between conceptual design (ER model) and physical implementation (relational database). Following established conversion rules ensures that the database accurately reflects the designed structure and maintains all specified constraints.

ER to Relational Mapping Overview

01

Map Strong Entities

Create a table for each strong entity with columns for all attributes

02

Map Weak Entities

Create tables including foreign keys from owner entities

03

Map Binary Relationships

Handle 1:1, 1:N, and M:N relationships appropriately

04

Map Multi-valued Attributes

Create separate tables for attributes with multiple values

05

Map Higher-Degree Relationships

Convert ternary and n-ary relationships to tables

06

Map Specialization Hierarchies

Choose appropriate strategy for superclass/subclass implementation

Mapping Strong Entity Sets

Conversion Rule

For each strong entity in the ER diagram:

1. Create a table with the same name
2. Include columns for all simple attributes
3. Expand composite attributes into simple components
4. Designate the primary key
5. Omit derived attributes (calculate when needed)

Example

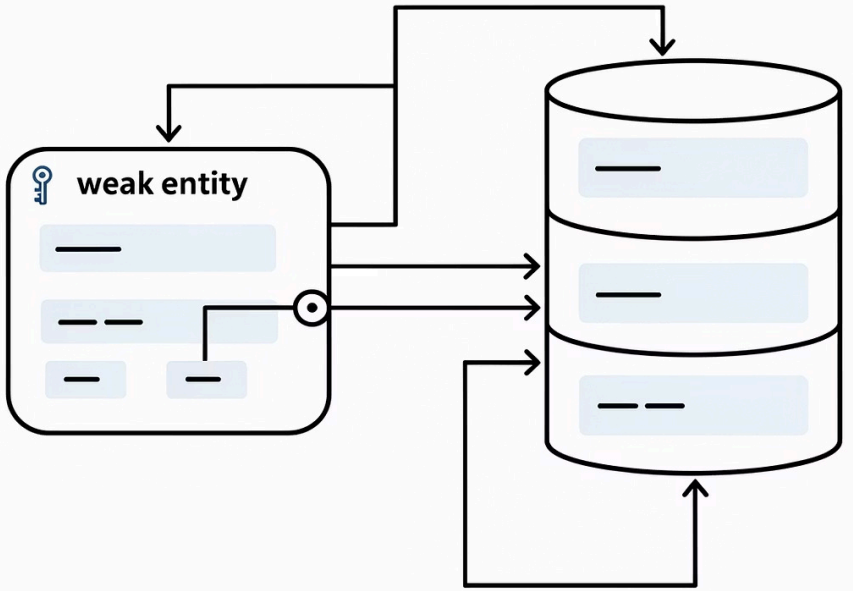
ER Entity: STUDENT

Attributes: Student_ID (PK), Name, DOB, Address (Street, City, ZIP), Age (derived)

Resulting Table: STUDENT

STUDENT
- Student_ID (PK)
- Name
- DOB
- Street
- City
- ZIP

Note: Age is derived from DOB, so not stored



Mapping Weak Entity Sets

Weak entities require special handling during conversion because they depend on their owner entity for identification.

Conversion Rule for Weak Entities

Create a table for the weak entity

Include all its attributes as columns

Include the owner's primary key as a foreign key

This establishes the dependency relationship

Form the primary key

Combine the owner's primary key (FK) + the weak entity's partial key

Add cascade constraints

Typically use ON DELETE CASCADE to maintain referential integrity

Weak Entity Mapping Example

ER Model

Strong Entity: EMPLOYEE

Employee_ID (PK), Name, Salary

Weak Entity: DEPENDENT

Dependent_Name (Partial Key), Age, Relationship

Identifying Relationship: Has_Dependent

Database Tables

EMPLOYEE
- Employee_ID (PK)
- Name
- Salary

DEPENDENT
- Employee_ID (FK, PK)
- Dependent_Name (PK)
- Age
- Relationship

Primary Key = {Employee_ID, Dependent_Name}

The dependent cannot exist without an employee. If Employee_ID 101 is deleted, all dependents with Employee_ID 101 are automatically removed (CASCADE).

Mapping Binary Relationships: 1:1

One-to-one relationships can be implemented using several approaches, depending on participation constraints.

1

Foreign Key Approach

Add the primary key of one entity as a foreign key in the other entity's table. Choose the entity with total participation to hold the foreign key to avoid NULL values.

Example: PERSON (1) - Has - (1) PASSPORT

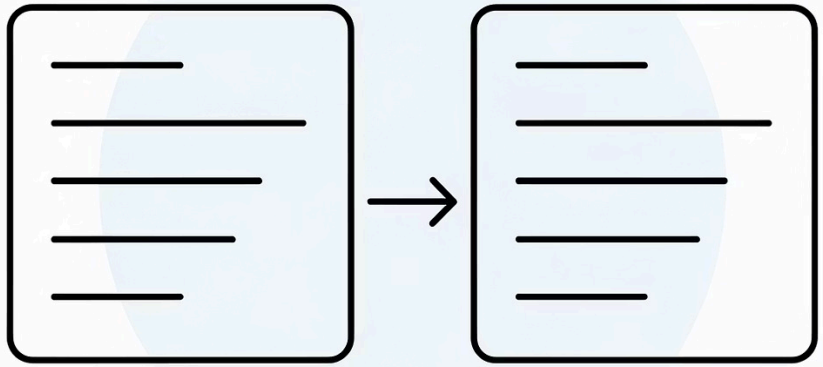
```
PERSON (Person_ID, Name, Passport_ID FK)
PASSPORT (Passport_ID, Number, Issue_Date)
```

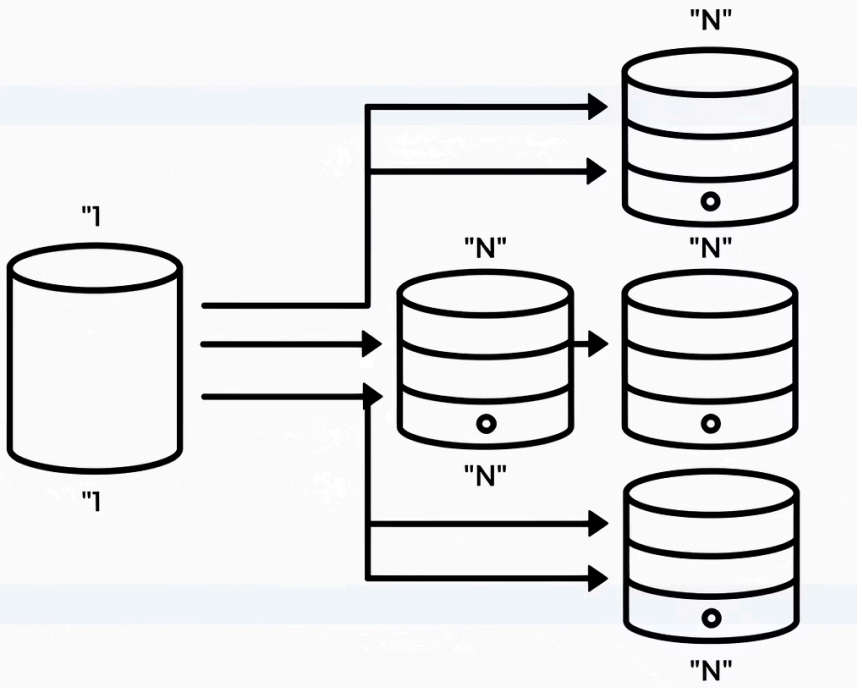
2

Merged Table Approach

If both entities have total participation, merge them into a single table.

Example: Combine PERSON and PASSPORT attributes into one CITIZEN table





Mapping Binary Relationships: 1:N

One-to-many relationships are the most common and straightforward to implement in relational databases.

Conversion Rule

For each 1:N relationship:

1. Identify the "many" side entity (N)
2. Add the primary key of the "one" side (1) as a foreign key in the "many" side table
3. The foreign key references the primary key of the "one" side
4. No additional table is needed

Why This Works

The "many" side table can reference the same "one" side record multiple times, but each "many" side record references only one "one" side record.

Example

DEPARTMENT (1) - Has - (N) EMPLOYEE

DEPARTMENT
- Dept_ID (PK)
- Dept_Name
- Location

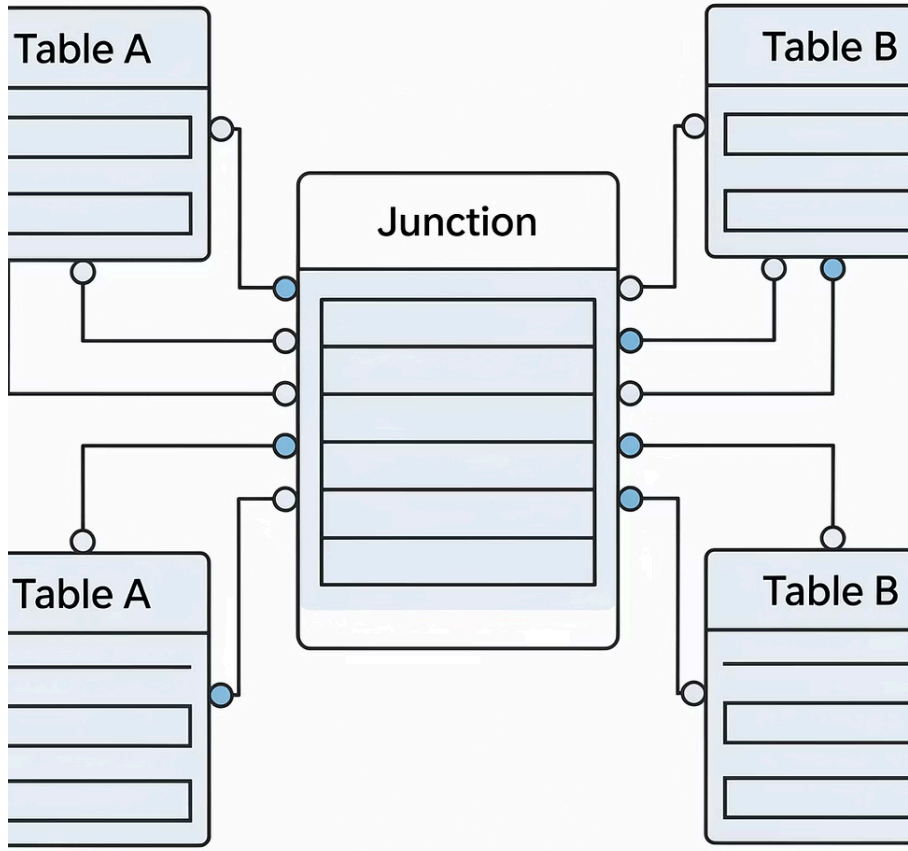
EMPLOYEE
- Emp_ID (PK)
- Name
- Salary
- Dept_ID (FK)

Mapping Binary Relationships: M:N

Many-to-many relationships require a junction table (also called bridge table, associative table, or linking table) because neither entity can hold a simple foreign key reference.

Conversion Rule

1. **Create Junction Table**
   New table with descriptive name
2. **Add Foreign Keys**
   Include PKs from both entities as FKs
3. **Define Primary Key**
   Combine both FKs as composite PK
4. **Add Attributes**
   Include relationship-specific attributes



M:N Relationship Mapping Example

ER Model

STUDENT (M) - Enrolls_In - (N) COURSE

Relationship attributes: Enrollment_Date, Grade

Many students can enroll in many courses, and each course can have many students.

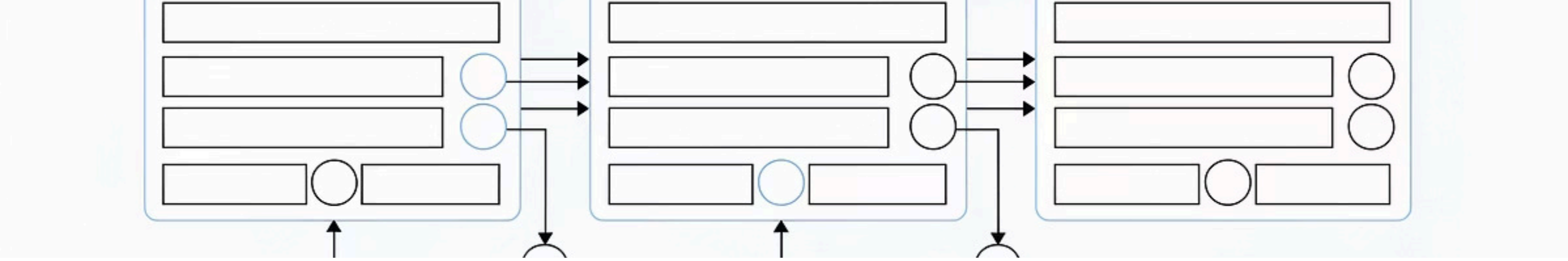
Database Tables

STUDENT
- Student_ID (PK)
- Name
- Major

COURSE
- Course_ID (PK)
- Course_Name
- Credits

ENROLLMENT (Junction Table)
- Student_ID (FK, PK)
- Course_ID (FK, PK)
- Enrollment_Date
- Grade

The ENROLLMENT junction table allows us to represent the many-to-many relationship and store attributes specific to each enrollment. Each row represents one student enrolled in one course.



Mapping Multi-valued Attributes

Multi-valued attributes (attributes that can have multiple values for a single entity instance) cannot be directly represented in relational databases and must be converted to separate tables.

Conversion Rule

01

Create New Table

Create a separate table for the multi-valued attribute

02

Include Foreign Key

Add the primary key of the parent entity as a foreign key

03

Add Attribute Column

Include a column for the multi-valued attribute

04

Define Primary Key

Primary key = {Parent PK, Attribute Value} or add surrogate key

Multi-valued Attribute Example

ER Model

Entity: EMPLOYEE

Attributes:

- Emp_ID (PK)
- Name
- Department
- Phone_Numbers (multi-valued)
- Skills (multi-valued)

An employee can have multiple phone numbers and multiple skills.

Database Tables

EMPLOYEE
- Emp_ID (PK)
- Name
- Department

EMPLOYEE_PHONE
- Emp_ID (FK, PK)
- Phone_Number (PK)

EMPLOYEE_SKILL
- Emp_ID (FK, PK)
- Skill_Name (PK)
- Proficiency_Level

This approach allows flexible storage of multiple values while maintaining normalization. An employee with three phone numbers will have three rows in EMPLOYEE_PHONE.

Mapping Specialization Hierarchies

Converting superclass/subclass hierarchies to relational tables can be done using multiple strategies. The choice depends on constraints (total/partial, disjoint/overlapping) and query patterns.

Strategy 1: Single Table

Create one table with all attributes from superclass and all subclasses. Add a discriminator column to identify the subclass type.

Advantages: Simple, no joins needed

Disadvantages: Many NULL values if subclasses have many unique attributes

Strategy 2: Table per Subclass

Create separate tables for each subclass containing both inherited and specific attributes.

Advantages: No NULL values, clear separation

Disadvantages: Data redundancy for shared attributes

Strategy 3: Table per Hierarchy Level

Create one table for superclass and separate tables for each subclass. Subclass tables only contain specific attributes + foreign key to superclass.

Advantages: No redundancy, no excessive NULLs

Disadvantages: Requires joins for queries

Specialization Mapping Example

Scenario: PERSON superclass with STUDENT and EMPLOYEE subclasses (overlapping, partial)

Strategy 1: Single Table

PERSON
- Person_ID (PK)
- Name
- Address
- Type
- Student_ID
- GPA
- Major
- Employee_ID
- Dept
- Salary

Many NULLs

Strategy 2: Per Subclass

STUDENT
- Person_ID (PK)
- Name
- Address
- Student_ID
- GPA
- Major

EMPLOYEE
- Person_ID (PK)
- Name
- Address
- Employee_ID
- Dept
- Salary

Redundant attributes

Strategy 3: Per Level

PERSON
- Person_ID (PK)
- Name
- Address

STUDENT
- Person_ID (FK,PK)
- Student_ID
- GPA
- Major

EMPLOYEE
- Person_ID (FK,PK)
- Employee_ID
- Dept
- Salary

Best normalized

Mapping Ternary Relationships

Ternary relationships (involving three entities) require junction tables similar to many-to-many binary relationships, but with three foreign keys.

Conversion Rule

Create Junction Table

Create a new table for the ternary relationship

Add Three Foreign Keys

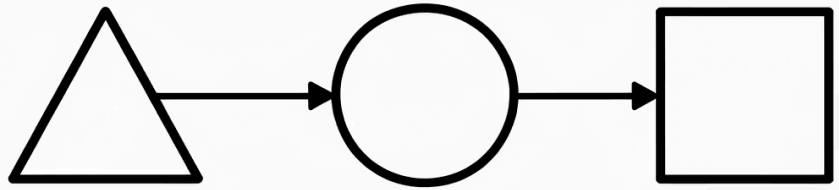
Include primary keys from all three participating entities as foreign keys

Define Primary Key

Typically use all three foreign keys as a composite primary key

Include Relationship Attributes

Add any descriptive attributes of the relationship itself



Ternary Relationship Example

ER Model

Relationship: SUPPLIES

SUPPLIER supplies PART to PROJECT

Attributes: Quantity, Date

This captures which supplier provides which parts for which projects, along with quantity and date information.

Database Tables

SUPPLIER
- Supplier_ID (PK)
- Name

PART
- Part_ID (PK)
- Part_Name

PROJECT
- Project_ID (PK)
- Project_Name

SUPPLIES
- Supplier_ID (FK, PK)
- Part_ID (FK, PK)
- Project_ID (FK, PK)
- Quantity
- Date

Each row in SUPPLIES represents a specific transaction where one supplier provides one type of part to one project.

Best Practices for ER to Database Conversion

- **Maintain naming conventions**

  Use consistent, descriptive names for tables and columns that match your ER diagram entities and attributes

- **Document foreign key relationships**

  Clearly specify all foreign key constraints, including ON DELETE and ON UPDATE rules

- **Choose appropriate data types**

  Select data types that efficiently store your data while ensuring accuracy and preventing invalid values

- **Consider indexing**

  Create indexes on foreign keys and frequently queried columns to improve performance

- **Normalize appropriately**

  Ensure tables are normalized to reduce redundancy, but consider denormalization for performance-critical queries

- **Validate with sample data**

  Test your schema with realistic data to identify potential issues before full implementation

Unit II Key Takeaways

ER Model Fundamentals

Entities, attributes, and relationships form the building blocks of database design

Cardinality Constraints

1:1, 1:N, and M:N relationships define how entities associate with each other

Keys Are Critical

Primary and foreign keys maintain uniqueness and referential integrity

Visual Communication

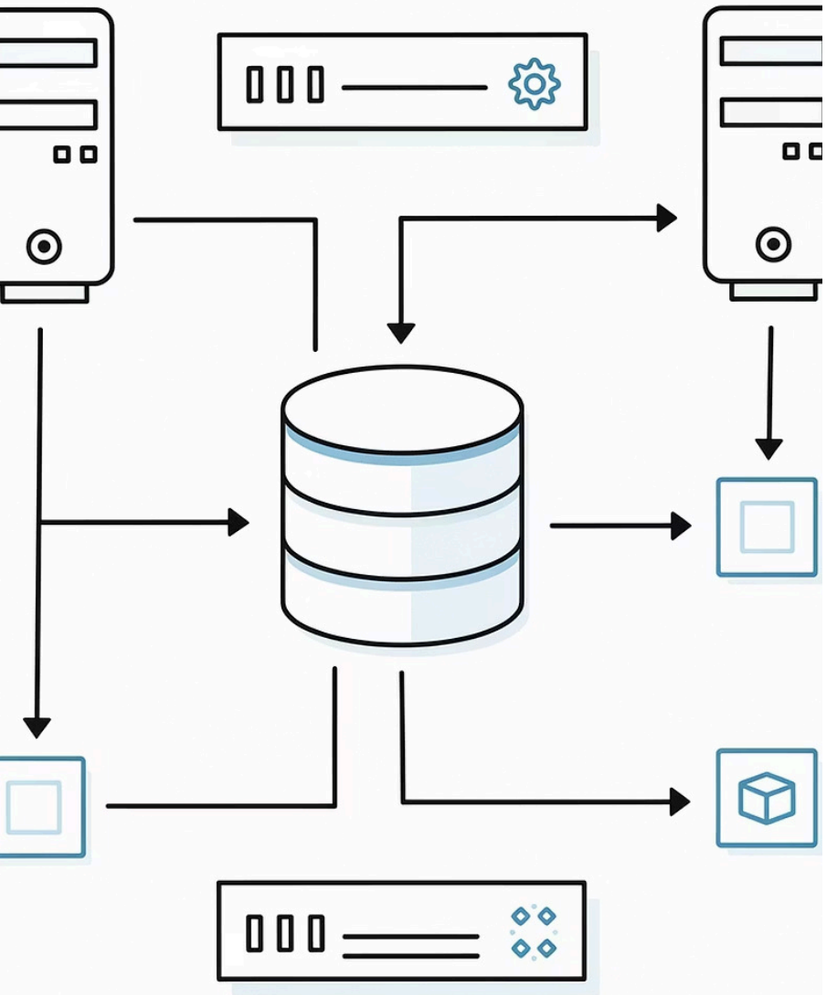
ER diagrams provide clear, standardized representations of database structure

Enhanced Features

Specialization, generalization, and aggregation handle complex scenarios

Systematic Conversion

Following mapping rules ensures accurate translation from ER to relational schema



Mastering Database Design

The Entity-Relationship model provides a powerful framework for conceptualizing and designing databases that accurately reflect business requirements. By mastering entities, attributes, relationships, and the conversion process, you build a strong foundation for creating efficient, scalable database systems.

As you continue your journey in database management, remember that good design is iterative. Start with a clear understanding of requirements, create thorough ER diagrams, validate with stakeholders, and systematically convert to relational schemas. These skills will serve you throughout your career in information technology.

- **Practice with real scenarios**
- **Review and refine designs**
- **Study existing databases**
- **Collaborate with peers**

Continue exploring database management concepts, and apply these principles to build robust, well-designed database systems.