**What You’ll Learn:**

* **Entity-Relationship Diagrams (ERDs):** Explore the use of ERDs to visually represent data relationships and ensure clarity in your database design.
* **Normalization Techniques:** Learn how to apply normalization rules to eliminate data redundancy and improve data integrity.
* **Design Strategies:** Explore various design strategies that cater to different business requirements and use cases.
* **Case Studies and Real-World Examples:** Gain insights from real-world scenarios and examples, helping you apply theoretical knowledge to practical situations.

# Design Process Overview

The database process consists of several stages, each stage comprising a unique set of tasks and considerations. From understanding the requirements and gathering necessary data, to implementing the database schema and ensuring its smooth operation, each phase plays a critical role in creating an efficient database solution for an organization.

* **Requirement Analysis:** Understand the requirements of the database by gathering information from users/stakeholders. Identify the entities, attributes, and relationships that need to be represented in the database.
* **Conceptual Design:** Requirements are tranformed into conceptual schema using E-R models. Create an Entity-Relationship Diagram (ERD) that represents the high-level conceptual view of the database. The conceptual schema should indicate functional requirements or operations that needs to be performed by the database.
* **Logical Design:** also called data model mapping. Convert the conceptual design into a logical design by mapping the ERD from the conceptual design to relational schema tables. Identify primary keys, foreign keys, and relationships between tables. Normalize the tables to eliminate redundancy and ensure data integrity.
* **Physical Design:** Define the physical characteristics of the database, including storage structures, indexing, and data types, record placement or file organization.
* **Implementation:** the database in the physical design is implemented, tested and deployed as a database application.

# Entity - Relationship (ER) Model

**一、为什么需要数据库设计**

* 在数据驱动的世界里，数据库必须：
  + **高效存储**
  + **快速检索**
  + **便于管理**
* 如果数据库设计不好，就会“垃圾进，垃圾出”（Garbage in, garbage out）。
* 好的设计能保证数据一致性、避免冗余，提高系统效率。

数据库设计通常有 **5 个阶段**：

1. **需求收集与分析** → 用户需求、业务规则
2. **概念设计** → 用 **E-R 模型**（ERD 图）表示
3. **逻辑设计** → 把 ER 模型转换成关系模式（表结构）
4. **物理设计** → 考虑存储、索引、磁盘布局
5. **实现与部署** → 真正建库、测试、上线

**二、ER 模型的基本概念**

* **E-R 模型 = Entity（实体） + Relationship（关系）**
* **实体（Entity）**：现实世界中可区分的对象（通常是名词，如 Author、Book、Student）
* **关系（Relationship）**：实体之间的联系（通常是动词，如 Author *writes* Book）
* 关系是 **双向的**：
  + 作者写书 ↔ 一本书由作者写

**三、常见的 ER 图表示法**

1. **Chen’s notation**（陈氏表示法）
   * 实体 = 矩形
   * 属性 = 椭圆
   * 关系 = 菱形
   * 基数（cardinality）= 数字或 1:N
2. **UML notation**（统一建模语言，更常用）
   * 实体 = 矩形，内部列出属性
   * 关系 = 连线
   * 基数 = 0..1, 1..*, 0..*
3. **Crow’s foot notation**（乌鸦爪表示法）
   * 实体 = 矩形
   * 关系 = 线，末端有“乌鸦爪”表示多
   * 基数用 |、||、|< 之类符号

**四、关系类型（Cardinality & Participation）**

* **一对多 (1:N)**
  + 一个作者写多本书，但一本书只能有一个作者
  + 外键放在“多”的一方（Book 表中放 AuthorID）
* **多对多 (M:N)**
  + 一个演员出演多部电影，一部电影有多个演员
  + 需要建立 **中间表（关联实体）**
* **一对一 (1:1)**
  + 一个用户对应一个 Profile

**参与度（Participation）**：

* **Mandatory (必须)** → 1
* **Optional (可选)** → 0

**基数（Cardinality）**：

* 最大参与数（1 或 N/∗）
* 可以精确表达 “至少 / 至多 n” （例如 0..5，1..\*）

**五、属性类型（Attributes）**

* **单值属性（Single-valued）**：一个作者只有一个名字
* **复合属性（Composite）**：地址 = {街道, 城市, 邮编}
* **多值属性（Multivalued）**：联系方式 = {电话1, 电话2}
* **派生属性（Derived）**：年龄可由出生日期计算 → 不放进关系模式

**六、键（Keys）**

* **主键（Primary Key, PK）**
  + 唯一标识元组，不可为空
  + 例如：AuthorID, BookID
* **外键（Foreign Key, FK）**
  + 引用另一表的主键，保证 **参照完整性**
* **复合键（Composite Key）**
  + 由多个属性组合而成
* **代理键（Surrogate Key）**
  + 人工生成的唯一 ID，没有业务含义

**七、ER 模型到数据库的映射**

* **确定关系类型**（1:1、1:N、M:N）
* **外键放在多的一方**
* **多对多关系要拆成新表**
* **确定主键 / 外键**
* **复合 / 多值属性要拆解**

**八、例子：大学数据库（UML ERD）**

* **Student – ClassSection**：
  + 学生必须注册至少 1 门课 (1..\*)
  + 一个课程班级有多个学生 (0..\*)
* **ClassSection – Room**：
  + 每个班级安排在一个房间 (1..1)
  + 一个房间可供多个班级使用 (0..\*)
* **Professor – ClassSection**：
  + 教授可能不教课 (0..\*)
  + 但一门课必须有一个教授 (1..1)
* **Course – ClassSection**：
  + 一个课程有多个班级 (1..\*)
  + 一个班级只属于一个课程 (1..1)

## Entity-relationship (ER) modeling

Entity-relationship (ER) modeling is a technique used in database design to visually represent the structure and relationships between entities (or objects) within a system. It provides a conceptual framework for understanding and organizing the data elements and their associations in a database. There are basically two important building blocks in ER modeling: entity and relationship.

**1.1 Entity**

Definition: "An entity represents a thing or object in the real world with an independent existence that can be differentiated from other objects, such as Author and Book" (Watt & Eng, 2014)

For example, in a book library database, entities could include authors, books, staff, and borrower. Each entity consists of attributes, which are the properties or characteristics that define an entity. In design practice, an entity would be a noun as represented in a given business rule.

Considering a book library database case: a borrower entity might have attributes such as borrower ID, first name, last name and street address.

**1.2 Relationship**

Definition: "A relationship represents associations or interactions between entities; for instance, an author writes books, but each book us written by one author. There is a relationship between author and book entities in both directions (bi-directional)" (Watt & Eng, 2014).

Relationships help us to describe how entities interact with each other in the system, and they can have different types, such as one-to-one, one-to-many, or many-to-many depending on the cardinality and participation constraints. In practice, a relationship would be a verb in a business rule, such as writes/written given in the business rule. A business rule is a brief, precise description of a policy, procedure or principle within a specific organization.

**1.3 Keys**

* **Primary Keys:** Every table typically has a primary key, which uniquely identifies each row or record in the table. It ensures the integrity and uniqueness of the data. Commonly, primary keys are denoted with underlined in a diagram. Primary key scan be simple keys with single attributes ot composite keys with two or more attributes. Primary keys enforce entity integrity. Entity Integrity ensures that there are no duplicate records within the table and that the field that identifies each record within the table is unique and never null.
* **Foreign Keys:** A foreign key is a column in a table that refers to the primary key of another table. It establishes a relationship between the two tables. Foreign keys are represented in diagrams as lines connecting the referencing table to the referenced table. A foreign key ensures referential integrity -- this means that a foreign key must have a matching primary key or it must be null. This rule is specified between two entities (parent and child). The rule maintains the correspondence between rows in these entities, and this reference must be valid.

**Modeling Constraints - Participation and Cardinality**

In a Unified Modeling language (UML) ERD, multiplicity is a combination of the participation and cardinality of the relationship.

Multiplicity = Participation + Cardinality

**Cardinality** refers to the maximum number of times an instance in one entity can relate to instances of another entity. Cardinality helps determine how many instances are allowed on each side of the relationship, and defines the type of relationship (either 1:M, M:M or 1:1)

**Participation** is about the minimum number of entities that are involved. Participation can be partial (also optional), if an entity does not have to participate in a relationship. For instance, not every borrower has membership to the library. Some borrowers do not wish to subscribe as members of a library. Participation is said to be total/mandatory, if entity must participate in a given relationship. Cardinality and Participation Constraints are indicated using notations such as "1" (one), "\* or N" (many), or "0..1" (zero or one).

Multiplicity of a relationship is formatted as min..max or x..y notation. The first number specifies the minimum number of instances allowed (participation), and the second number specifies the maximum number allowed (cardinality).

**References**

Watt, A., & Eng, N. (2014). *Database Design, 2nd edition*. BCcampus. https://openlibrary-repo.ecampusontario.ca/jspui/handle/123456789/247

The ERD below shows Customer, Product, and Order Information in a CustomerOrder database. Use the ERD to list all the business rules represented in the diagram. The answer is at the bottom of this page.

## ERD for CustomerOrder Database:

A diagram of a product

AI-generated content may be incorrect.

CustomerOrder Business Rules:

1. One customer place multiple (1 or more) orders. Each order is associated with only one (1 and only 1) customer.
2. Customer stores CustomerID, name of customer, Phone, and email as attributes.
3. Order has OrderID, date of order, and total amount as attributes.
4. Each order can include multiple products, and each product can be part of multiple orders.
5. Product has ProductID, name of product, and price.
6. Each product belongs to a specific category, and a category can have multiple products.
7. Category consists of CategoryID, name and description fields.

## Relation Schemas from Banking ERD

**Translate ERD for banking database in relation schemas:**

Customer (CustomerID, CustomerName, Address, Phone)

Account (AccountNo, AccountType, Balance, CustomerID)

Transaction (TransID, Date, Amount, AccountNo, BranchID)

Branch (BranchID, BranchName, Location)

Employee (EmployeeID, EmployeeName, Position, BranchID)

## **Relation Schemas from CustomerOrder ERD**

**Translate ERD for CustomerOrder database in relation schemas:**

Customer (CustomerID, CustomerName, Email, Phone)

Order (OrderID, OrderDate, TotalAmount, CustomerID)

Product (ProductID, ProductName, Price, CategoryID)

Category (CategoryID, CategoryName, Description)

ProductOrder (ProductID, OrderID, Quantity)

# Extended ER Model/Diagram

**一、什么是扩展 ER 模型 (EER)**

* EER 是 **ER 模型的增强版**。
* 在传统实体–关系模型 (ER) 的基础上，EER 增加了更多特性，使我们能更好地表示 **复杂的关系和约束**。
* 这些扩展特性包括：
  + **Supertype / Subtype（超类型与子类型）**
  + **Specialization / Generalization（特化 / 泛化）**
  + **Overlapping vs. Disjoint Subtypes（重叠子类 vs. 互斥子类）**
  + **Completeness Constraint（完全性约束：全体 vs. 部分）**
  + **Weak vs. Strong Entities（弱实体与强实体）**

**二、超类型 (Supertype) 与子类型 (Subtype)**

* **Supertype（超类型）**：抽象的上层实体，表示共性。
  + 例如：Person（人）
* **Subtype（子类型）**：下层实体，表示特化。
  + 例如：Student（学生）、Staff（职员）、TA（助教）、RA（科研助理）
* 子类型 **继承** 超类型的属性和关系。
  + Person 有 Name、BirthDate 等属性
  + Student 继承这些属性，同时增加 Major、GPA
  + Staff 增加 Department

**三、特化与泛化**

* **Specialization（特化，Top-Down）**
  + 从 **超类型 → 子类型** 的过程
  + 根据实体的 **独特特征** 划分
  + 例如：Person → Student / Staff
* **Generalization（泛化，Bottom-Up）**
  + 从 **子类型 → 超类型** 的过程
  + 根据 **共同特征** 归纳
  + 例如：TA 和 RA 共同泛化为 Student

**四、子类型约束**

1. **Overlapping（重叠子类）**
   * 一个超类型实例可以属于多个子类
   * 例如：某人既是 Student 又是 Staff
2. **Disjoint（互斥子类）**
   * 一个超类型实例 **只能属于一个子类**
   * 例如：学生要么是 TA，要么是 RA，不可能同时是两者

**五、完全性约束 (Completeness Constraint)**

* **Total Completeness（全体约束）**
  + 超类型的所有实例必须属于某个子类
  + 例如：所有 Person 必须是 Student 或 Staff
* **Partial Completeness（部分约束）**
  + 超类型的实例可以不属于任何子类
  + 例如：某些 Person 既不是 Student 也不是 Staff

**六、弱实体与强实体**

* **强实体 (Strong Entity)**
  + 自己的属性能唯一标识实例
  + 例如：Person(PersonID)
* **弱实体 (Weak Entity)**
  + 自身属性无法唯一标识，必须依赖强实体的主键
  + 例如：Dependent(DependentID, PersonID)
  + 其主键通常是 **组合键 (Composite Key)**：强实体主键 + 自身部分键

**七、关系模式 (Relational Schema) 表示**

* **Supertype** → 单独一张表（Person）
* **Subtype** → 单独一张表，包含：
  + 父类主键（同时是外键）
  + 继承的属性
  + 子类自己的属性

例如：

Person(PersonID [PK], FirstName, LastName, BirthDate)

Student(PersonID [PK, FK], Major, GPA)

Staff(PersonID [PK, FK], Department)

TA(PersonID [PK, FK], Hours, Credit)

RA(PersonID [PK, FK], Date, Topic)

Dependent(PersonID [PK, FK], DependentID [PK], Name)

**八、复合属性与多值属性**

* **复合属性 (Composite Attribute)**
  + 需拆分为单一属性
  + 例如：AuthorName → AuthorName\_FirstName, AuthorName\_LastName
* **派生属性 (Derived Attribute)**
  + 由其他属性计算得出（如 Age = Today – BirthDate）
  + **不放入关系模式**
* **多值属性 (Multivalued Attribute)**
  + 需要新建一个关系表
  + 例如：Author 有多个联系方式
* Author(AuthorID [PK], Name)
* Author\_ContactInfo(AuthorID [PK, FK], ContactInfo [PK])

**九、ER 设计中的常见问题**

1. **实体 vs. 属性**
   * 如果需要对“电话”记录更多信息（类型、服务商），则应该建实体而不是属性。
2. **关系 vs. 实体**
   * 如果“贷款”有自己的属性（金额、利率），应建为实体而不是单纯的关系。
3. **二元 vs. 多元关系**
   * 能拆成二元关系的，尽量不要保留多元关系。
4. **关系属性的放置**
   * 对于 1:N 关系，把属性放在 **N 的一方**
   * 对于 M:N 关系，新建 **关联实体**

✅ **总结**  
扩展 ER 模型 (EER) 通过引入 **超类 / 子类、继承、重叠 / 互斥、全体 / 部分约束** 等机制，更好地表达复杂业务规则。最终在转换为关系模式时，仍然要落实到：

* **主键 / 外键设计**
* **多值 / 复合属性的拆分**
* **弱实体依赖强实体**

## Extended Entity-Relationship Modeling

Extended Entity-Relationship (EER) modeling is an enhanced version of traditional Entity-Relationship (ER) modeling that incorporates additional concepts and constructs to represent complex relationships, constraints, and features more effectively. EER modeling extends the capabilities of ER modeling by introducing additional components. In addition, EER modeling extends the basic components of an Entity-Relationship Diagram (ERD) to include components beyond entities, attributes, and relationships. Below are some of the extended components commonly used in an Extended Entity-Relationship Model/Diagram (EERM/EERD).

**Extended - ER components:**

* **Entity Subtype and Supertype:** both are allow for modeling hierarchical relationships between entities. A supertype is also a high-level entity that represents a general aspect, while subtypes are low-level entities that represent specialized aspect of an entity in the hierarchy. Each subtype inherits attributes and relationships from the supertype and related to specific characteristics of an entity. The relationship between the supertypes and subtypes is known as **"ISA"** This allows for capturing variations and specialized characteristics within a group of entities.
* **Specialization and Generalization:** Specialization and generalization are used to represent relationships between entities in the ISA relationships. Specialization is the is a top-down design process of creating subtypes from a supertype by grouping each subtype basing on specific or unique characteristics, while generalization is the reverse process of combining subtypes into a supertype.
* **Inheritance:** Inheritance is the mechanism by which subtypes inherit attributes and relationships from their supertype. It allows for reusing and extending the characteristics of entities, reducing redundancy and improving data consistency.
* **Weak Entities:** A weak entity depends on another entity, called its identifying or owner/strong entity, to have a meaningful existence. They are existence-dependent -- implies that their existence depends on the exisitence of other entities. A weak entity is identified by a partial key, in addition to its relationship with the owner entity.
* **Multi-valued Attributes:** Multi-valued attributes represent an attribute that can have multiple values for a single instance of an entity. They are indicated by a double oval in the EER diagram.
* **Derived Attributes:** Derived attributes are attributes whose values can be calculated or derived from other attributes. They are represented by dashed ovals in the EER diagram.

Detailed Narrative Including Extend-ER features.

**Faculty database example**

A Person may have several dependents, but each dependent depends on one person. A Person is a Student and a Staff (specialization). A student is a Person (generalization), and Staff is a Person with Dept as attribute. A Person has PersonID, FirstName, LastName and BirthDate that needs to be inherited by all subtype entities. Student can be TA or RA and has Major, and GPA attributes. TA has Hours and Credit, while RA stores Date and Topic. Lastly, the dependent has DepID, DepName, and DepRelation as attributes.

A diagram of a data flow

AI-generated content may be incorrect.

\*TA stands for Teaching Assistant

\*RA stands for Research Assistant

# Functional Dependency (FD)

**📌 什么是函数依赖？**

在关系数据库中，如果 **属性集 X 的值能唯一决定属性集 Y 的值**，我们就说 **X → Y**。

* X：决定因素（determinant，箭头左边）
* Y：被依赖属性（dependent，箭头右边）

⚠️ 重要条件：  
只要两个元组（行）在 X 上取值相同，那么它们在 Y 上的取值也必须相同。

**📌 函数依赖的类型**

1. **完全函数依赖（Full FD）**
   * 被依赖属性必须依赖于全部的决定因素。
   * 例子：{A,B} → C，C 依赖 A 和 B 两个一起，如果去掉 A 或 B，依赖关系就不成立。
2. **部分函数依赖（Partial FD）**
   * 被依赖属性只依赖于复合键的一部分。
   * 例子：{A,B} → C，但其实 A → C 就成立。
   * 🚨 在数据库范式中，这种情况会导致 **不满足第二范式 (2NF)**。
3. **传递依赖（Transitive FD）**
   * 依赖关系通过中间属性间接传递。
   * 例子：A → B 且 B → C，那么有 A → C（传递依赖）。
   * 🚨 在数据库范式中，这种情况会导致 **不满足第三范式 (3NF)**。

**📌 键的概念**

* **候选键 (Candidate Key)**：最小的属性集，可以唯一标识一行数据。
* **主键 (Primary Key)**：从候选键中选定的一个作为主要标识符。
* **主属性 (Prime Attribute)**：属于某个候选键的属性。
* **非主属性 (Non-prime Attribute)**：不属于任何候选键的属性。

**📌 要点总结**

* 函数依赖必须是一种 **一对一** 的映射。
* 如果决定因素的某个值对应多个不同的被依赖值（即一对多），那就 **不是函数依赖**。
* 如果某个属性列的取值全都唯一（比如学号、身份证号），那它就是候选键。

# Attribute Closure and Candidate Keys

**📌 属性闭包 (Attribute Closure, X⁺)**

**定义**：  
给定一个属性集 X 和一组函数依赖 F，闭包 X⁺ 表示从 X 出发，利用 F 能推导出的所有属性的集合。

👉 通俗说：从 X 开始，根据函数依赖能一路“推”出来的所有属性。

**算法步骤（Armstrong Axioms 方法）**：

1. 初始化：X⁺ = X
2. 对于每一个函数依赖 Y → Z：
   * 如果 Y ⊆ X⁺，则将 Z 加入 X⁺。
3. 重复第 2 步，直到不能再加入新属性。
4. 返回最终的 X⁺。

**📌 Armstrong 公理（三条推理规则）**

1. **自反律 (Reflexivity)**  
   如果 Y ⊆ X，那么 X → Y 一定成立。  
   （大的集合总能决定它的子集）
2. **增广律 (Augmentation)**  
   如果 X → Y，那么 XZ → YZ 也成立。  
   （在左右两边同时加相同的属性，依赖仍然成立）
3. **传递律 (Transitivity)**  
   如果 X → Y 且 Y → Z，那么 X → Z。  
   （间接决定关系）

**📌 属性闭包的应用**

1. **判断候选键**：  
   如果某个属性集 X 的闭包 X⁺ 包含了关系 R 的所有属性，那么 X 就是 **超键 (Superkey)**。  
   如果 X 是最小的（没有子集也能当超键），那么 X 就是 **候选键 (Candidate Key)**。
2. **验证函数依赖**：  
   如果 Y ⊆ X⁺，则可以说 X → Y 成立。

**📌 示例**

关系：R(A, B, C, D, E)  
函数依赖集：

1. CE → D
2. D → B
3. C → A

问题：求 (CE)⁺

**计算过程**

* 初始：(CE)⁺ = {C, E}
* 因为 CE → D，所以加上 D → (CE)⁺ = {C, E, D}
* 因为 D → B，所以加上 B → (CE)⁺ = {C, E, D, B}
* 因为 C → A，所以加上 A → (CE)⁺ = {C, E, D, B, A}
* 最终得到 (CE)⁺ = {A, B, C, D, E}

结论：

* CE 的闭包包含所有属性，因此 CE 是一个 **超键**。
* 检查它的子集：
  + C⁺ = {C, A}
  + E⁺ = {E}
  + 都不包含所有属性
* 所以 CE 是 **候选键 (Candidate Key)**。
* 如果只有这一个候选键，设计者就会把 CE 定为 **主键 (Primary Key)**。

✅ 总结：

* **属性闭包** 是通过函数依赖把能推出的属性都列出来。
* **候选键** 是最小的属性集，它的闭包包含关系的所有属性。
* **主键** 是候选键中的一个。

## Armstrong's Axioms Closure Algorithm

To determine the functional dependencies in a given set of attributes, you can use an algorithm called the Armstrong's axioms closure algorithm. This algorithm calculates the closure of a set of attributes under a given set of functional dependencies. Here's a step-by-step explanation of the algorithm:

* Initialize the closure set as the given set of attributes.
* Repeat the following steps until no further attributes can be added to the closure:
* For each functional dependency X →→right arrow Y in the given set of functional dependencies:
* If X is a subset of the closure set, add Y to the closure set.

Return the closure set.

The closure set obtained from the algorithm represents all the attributes that are functionally dependent on the given set of attributes. This algorithm applies Armstrong's axioms, which are a set of rules that define the properties and inference rules of functional dependencies.

**Armstrong's Axioms Include:**

Reflexivity: If Y is a subset of X, then X →→right arrow Y.

Augmentation: If X →→right arrow Y, then XZ →→right arrow YZ for any set of attributes Z.

Transitivity: If X →→right arrow Y and Y →→right arrow Z, then X →→right arrow Z.

The closure algorithm iteratively applies these axioms to determine the closure of a set of attributes. It checks if the closure set already contains the attributes on the left-hand side (X) of each functional dependency and adds the attributes on the right-hand side (Y) if necessary.

## Attribute Closure

Given the relation R = (A, B, C, D, E) with functional dependencies FDs = {CE →→right arrowD, D →→right arrow B, C →→right arrow A}, what is the attribute closure?

**Computing CE closure (CE)+ =**

1st iteration, answer = {CE} by reflexivity rule

2nd iteration, answer = {CED} from CE →→right arrow D

3rd iteration, answer = {CEDA } from C →→right arrow A

4th iteration, answer = {CEDAB} from D →→right arrow B

5th iteration, no change to the 4th iteration answer, therefore

(CE)+ = {CEDAB } OR {ABCDE}, which is equivalent to R ={ABCDE} i.e., all the attributes in (CE)+ are included in R. We conclude that CE is a super key of relation R.

**Computing D closure (D)+ =**

1st iteration, answer = {D} by reflexivity rule

2nd iteration, answer = {DB} from D →→right arrow B

3rd iteration, no change to the 2nd iteration answer, therefore

(D)+ = {DB} is not equivalent to R ={ABCDE} i.e., (D)+ does not include all the attributes of R. We conclude that D is NOT a key (super key) of relation R.

**Computing C closure (C)+ =**

1st iteration, answer = {C} by reflexivity rule

2nd iteration, answer = {CA} from C→→right arrow A

3rd iteration, no change to the 2nd iteration answer, therefore

(C)+ = {CA} is not equivalent to R ={ABCDE} i.e., (C)+ does not include all the attributes of R. We conclude that D is NOT a key (super key) of relation R.

**Computing CD closure (CD)+ =**

1st iteration answer = {CD} by reflexivity rule

2nd iteration answer = {CDB} from D →→right arrow B

3rd iteration answer = {CDBA} from C →→right arrow A

4th iteration answer = no change to the 4th iteration answer, therefore

(CD)+ = {CDBA} is not equivalent to R ={ABCDE} i.e., (CD)+ does not include all the attributes of R. We conclude that CD is NOT a key (super key) of relation R.

# Canonical / Minimal Cover

**什么是最小覆盖（Canonical / Minimal Cover）**

**最小覆盖（记作 Fc）** 是对一组函数依赖 F 的“等价精简版”：

* **等价**：Fc 与 F 的**闭包相同**，即 F⁺ = Fc⁺（两者能推导出的依赖集合完全一致）。
* **最小**（常见的“规范”要求）：
  1. **右部（RHS）都是单属性**：每条 FD 的右侧只有 1 个属性。
  2. **左部（LHS）没有冗余属性**：任意一条 FD 的左侧，删掉任何一个属性都会改变闭包（也就是删不得）。
  3. **没有冗余 FD**：去掉任何一条 FD，闭包就变了（也就是每条都必不可少）。  
     4)（常见附加）**左部唯一**：同一左部的多条 FD 可按需**合并**（Union/合并律）。

⚠️ 注：**最小覆盖一般并不唯一**。因为检验/删除的顺序不同，最终形态可能略有差异，但都与原集合等价。

**关键概念**

**1) 冗余（Extraneous / Redundant）**

* **冗余属性（左侧）**：若在某条 FD X → A 中，左侧 X 的某个属性 x 可以被去掉且**不改变**整个集合的闭包（仍能推出相同依赖），则 x 在该 FD 的左侧是**冗余**的。
* **冗余属性（右侧）**：若 X → YZ 中，右侧属性 Z 可以被移走而不影响闭包（因为 X → Z 能从其它 FD 推出），则 Z 对这条 FD 的右侧是**冗余**的。  
  但在**规范最小覆盖**里，我们先把 RHS 拆成单属性（见算法步骤），所以右侧“冗余属性”的检查通常转化为“冗余 FD”的检查。
* **冗余 FD**：如果把某条 f 从集合 F 中移除，剩余 F − {f} 的闭包**仍然**等于原闭包（(F − {f})⁺ = F⁺），则 f 是冗余 FD，可删。

**Armstrong 公理（推理规则）**

基础三条（之前课里讲过）：

1. **自反性**：Y ⊆ X ⇒ X → Y
2. **增广性**：X → Y ⇒ XZ → YZ
3. **传递性**：X → Y 且 Y → Z ⇒ X → Z

常用派生规则（这节课新增）：

* **合并（Union）**：X → Y 且 X → Z ⇒ X → YZ
* **分解（Decomposition）**：X → YZ ⇒ X → Y 且 X → Z
* **伪传递（Pseudo-transitivity）**：X → Y 且 Y Z → W ⇒ X Z → W

我们在做“最小覆盖”时，会大量用到 **分解**（把 RHS 拆成单属性）、**合并**（最后把相同 LHS 的多条 FD 合成一条）。

**判定“冗余”的实操方法**

**A. 检查左侧是否有冗余属性（对单条 FD）**

给定集合 F，单条 FD 为 X → A，想检测 x ∈ X 是否冗余：

1. 令 X' = X − {x}。
2. 在 **F 的闭包框架下** 计算 X'⁺（就是用 F 反复推演，看能得到什么属性）。
3. **若 A ∈ X'⁺**，说明即便不依靠 x，X' 也能推出 A，因此 x 是冗余的，可从这条 FD 的左侧删掉。  
   否则 x 不冗余，保留。

直观：如果去掉 x 后，这条依赖能由“其它路径”补上，则 x 多余。

**B. 检查右侧属性是否冗余（较少用，因我们先拆 RHS）**

给定 X → YZ，想检测 RHS 上的 Z 是否冗余：

1. 构造 F' = (F − {X → YZ}) ∪ {X → Y}（暂时把 Z 从该条 FD 里移走）。
2. 在 F' 下计算 X⁺。
3. 若 Z ∈ X⁺，则 Z 对该条 FD 冗余（因为它能被其它 FD 推出），删除即可。

但通常我们先把 RHS **分解为单属性**，此时“右侧冗余”就倒退为“整条 FD 是否冗余”的检查。

**C. 检查整条 FD是否冗余**

给定某条 f: X → A：

1. 构造 F' = F − {f}（把这条先拿掉）。
2. 在 F' 下计算 X⁺。
3. 若 A ∈ X⁺，说明 f 可由其余 FD 推出，是冗余 FD，可删。

**最小覆盖的标准算法（手工版）**

**输入**：一组 FD，F  
**输出**：一个最小覆盖 Fc，满足 Fc⁺ = F⁺ 且符合“最小/规范”的要求

**步骤**：

**Step 0.（可选）预处理**

* 去掉**语义上明显重复**的 FD（完全相同的两条）。

**Step 1. 右部单属性化（分解律）**

* 把每条 X → Y1Y2…Yk 拆成 k 条：X → Y1、X → Y2、…、X → Yk。
* 这样做的目的是：后续“冗余”判断都在单属性 RHS 的粒度下进行。

**Step 2. 清理左侧冗余属性**

* 对集合中每条 X → A，对 X 里的每个属性 x，按上文 A 的方法测试是否冗余；冗余则移除。

**Step 3. 删除冗余 FD**

* 对集合中每条 f: X → A，按上文 C 的方法测试是否冗余；冗余则删除。

**Step 4.（可选但常用）合并相同左侧**

* 若存在同一左侧 X 指向多个右侧单属性（如 X → A、X → B…），可按合并律合成 X → AB…。
* 若你想保持“RHS 全为单属性”的形式，这一步可以不做；教材/平台有时要求**保持单属性 RHS**。

✅ 执行完 Step 1–3，就已经是**最小覆盖**。Step 4 仅是“美化/规整”。

**常见陷阱 & 小技巧**

* **顺序会影响中间形态**：你先删谁、后删谁，得到的中间集合可能不同，但最终都是与 F 等价的最小覆盖。
* **RHS 先单属性化**：不这么做，后面判断会绕。
* **做闭包时别把“正在测试的那条”用进去**：
  + 查“冗余 FD”时，要在 F − {f} 下算闭包；
  + 查“LHS 冗余属性”时，用**完整 F**计算 去掉该属性后的左侧 的闭包。
* **最小覆盖 ≠ 唯一**：多条等价的解是正常的。
* **工具思维**：大量闭包计算可以半自动（纸笔表格/脚本），避免手算出错。
* **设计意义**：最小覆盖是后续 **分解、规范化（2NF/3NF/BCNF）** 的基础材料；它让依赖表达更“紧凑、无重复”。

# Database Normalization

Database normalization is a fundamental process in database design as it ensures data integrity, minimizes redundancy, and improves overall database performance and reliability. It provides a solid foundation for efficient and effective data management in relational databases. Database normalization applies a systematic process to the relations, reducing the degree of redundancy and guaranteeing the following properties:

* Lossless decomposition
* Dependency preservation

**Lossless decomposition:**

Assume a relation R is decomposed into set of relations R1, R2, and R3. If R1, R2, and R3 gets joined and all the data of original relation R can be derived, then such a decomposition is defined as Lossless decomposition.

**Dependency preservation:**

Assume a relation R is decomposed into R1, R2, and R3 with a set of functional dependencies of the relations R1, R2, and R3. If we can derive all the functional dependencies of the original relation R, using a given set of FDs, then such a decomposition is defined as dependency preservation.

**Need for DB Normalization**

* **Eliminating Data Redundancy:** One of the primary goals of normalization is to eliminate data redundancy. Redundant data increases storage requirements, wastes disk space, and introduces the risk of inconsistencies and anomalies when data is updated. By organizing data into normalized relations, redundant data is minimized, and each piece of information is stored in only one place, promoting data consistency.
* **Ensuring Data Integrity:** Normalization helps maintain data integrity by eliminating update anomalies. Update anomalies occur when modifying data in one place leads to inconsistencies or errors in other parts of the database. Normalization reduces these anomalies by ensuring that data dependencies are properly represented and maintained.
* **Facilitating Data Consistency:** Normalized relations make it easier to maintain data consistency across the database. By accurately representing relationships between entities and dependencies between attributes, normalization helps enforce referential integrity and consistency constraints.
* **Reducing Data Anomalies:** Normalization helps eliminate various types of data anomalies, such as insertion anomalies (difficulty in inserting data due to missing dependencies), deletion anomalies (unexpected loss of data due to dependencies), and modification anomalies (inconsistencies caused by partial updates). By addressing these anomalies, normalization promotes data integrity and reliability.

## Introduction of Database Normalization

**1. 定义**

**数据库规范化**是一种系统化的过程，用来 **评估并修正** 数据表结构，以减少 **数据冗余（redundancy）** 和 **数据异常（anomalies）**，从而保证：

* **无损分解（Lossless decomposition）**  
  把一个关系 R 分解成多个子关系 R1, R2, … 后，再做自然连接（JOIN）能还原原始数据，不丢失信息。
* **依赖保持（Dependency preservation）**  
  分解后，子关系的函数依赖集合合起来，仍然能推出原始关系的所有依赖。

**2. 为什么需要规范化**

* **消除冗余**：相同信息不应存放在多个地方（避免浪费、减少存储错误）。
* **保证数据完整性**：更新时保持一致，否则可能一处改了另一处没改。
* **保证一致性**：命名、属性关系一致（比如 StudentID 和 StudentNumber 不要混用）。
* **减少异常（anomalies）**：
  + **更新异常（Update anomaly）**：只更新部分位置，导致不一致。
  + **插入异常（Insertion anomaly）**：插入一条记录时被迫加入无关或“虚拟”信息（phantom values）。
  + **删除异常（Deletion anomaly）**：删除一条数据时，意外删掉与之无关的重要信息。

**3. 函数依赖中的问题**

* **完全函数依赖（Full FD）**  
  一个属性集完整依赖于候选键。
* **部分函数依赖（Partial FD）**  
  非主属性依赖于复合主键的一部分。
* **传递函数依赖（Transitive FD）**  
  非主属性依赖于另一个非主属性，而该非主属性又依赖于主键。

✅ 部分依赖和传递依赖都是不希望存在的。

**🔹各范式（Normal Forms）**

**0. 非规范化（UNF, Unnormalized Form）**

* 有重复组（repeating groups）或多值属性（multivalued attributes）。
* 单元格里可能有多个值，甚至有空值（NULL）。

**1NF（第一范式）**

**条件：**

* 所有属性值必须是**原子值（atomic）**（不能再分割）。
* 不能有重复组。
* 不能有多值属性。

**转换方法：**

1. 消除重复组（复制主键填充）。
2. 拆开多值属性 → 每个值占一行。
3. 确定主键。

**2NF（第二范式）**

**条件：**

* 已在 1NF。
* 每个非主属性**完全依赖**于主键（即消除部分依赖）。

**关键点：**

* 如果主键是单属性 → 自动满足 2NF。
* 如果主键是复合键 → 要保证非主属性依赖于整个复合键。

**处理方法：**

* 针对**部分依赖**，单独生成新表。
* 保留主键拷贝作为外键，保持参照完整性。

**3NF（第三范式）**

**条件：**

* 已在 2NF。
* 不存在**传递依赖**（non-prime 属性不能依赖于其他非主属性）。

**处理方法：**

* 针对**传递依赖**，再拆出一个新表。
* 在原表保留该依赖的“键”作为外键。

**🔹总结口诀**

1. **1NF**：消除重复组 & 原子值。
2. **2NF**：消除部分依赖（非主属性必须依赖于整个键）。
3. **3NF**：消除传递依赖（非主属性不能依赖非主属性）。
4. **BCNF**：决定因素必须是候选键。

## Normal Forms (NFs)

Normal forms are a set of rules that help ensure the efficiency, integrity, and "good" or normal structure of a database schema. The normalization process utilizes these normal forms to create normalized relations by eliminating data redundancy and dependency anomalies. There are several normal forms, each with specific criteria and requirements:

* **First Normal Form (1NF):** In 1NF, the data is organized into tables, and each column contains atomic values (indivisible and cannot be further divided). It eliminates repeating groups of data and ensures that each attribute contains only a single value.
* **Second Normal Form (2NF):** In 2NF, the table is in 1NF, and all non-key attributes are functionally dependent on the entire primary key. It eliminates partial dependencies by separating data into multiple tables, with each table having a primary key that identifies its records uniquely.
* **Third Normal Form (3NF):** In 3NF, the table is in 2NF, and no non-key attribute is transitively dependent on the primary key. It eliminates transitive dependencies by decomposing the table into multiple tables based on functional dependencies.
* **Boyce-Codd Normal Form (BCNF):** BCNF is a stricter version of 3NF that eliminates all non-trivial functional dependencies. It ensures that every determinant is a candidate key, meaning no non-key attribute is functionally dependent on another non-key attribute.
* **Fourth Normal Form (4NF):** In 4NF, the table is in BCNF, and it further eliminates multi-valued dependencies. It decomposes the table into multiple tables to handle independent multi-valued attributes.
* **Fifth Normal Form (5NF):** 5NF is designed to handle cases where there are join dependencies between independent multi-valued relationships. It decomposes the tables into smaller tables to eliminate join dependencies.

## Boyce-Codd Normal Form

**1. 定义**

一个关系模式 **在 BCNF 中** 当且仅当：  
**每一个决定因素（determinant）都是候选键**。

* 决定因素：函数依赖 X → Y 中的左边 X。
* 候选键：能唯一标识关系中元组的最小属性集。

👉 换句话说，**不能存在“非候选键 → 其他属性”** 的函数依赖。

**2. 与 3NF 的关系**

* BCNF 是 **3NF 的更严格形式**。
* 如果一个关系只有一个候选键，那么它自动满足 3NF 和 BCNF。
* **只有当存在多个候选键时，BCNF 可能被违反**。

**3. 违反 BCNF 的典型情况**

* 出现了 **非候选键 → 候选键的组成部分** 的依赖。  
  比如：候选键是 (B, C)，但我们有 D → B。
  + D 不是候选键，却决定了主键的一部分 → 违反 BCNF。

**4. 转换方法（分解步骤）**

当发现依赖 X → A 违反 BCNF 时：

1. **建立新关系 R1**：包含 X 和 A（即决定因素 + 被决定属性）。
2. **建立新关系 R2**：原始关系减去 A（去掉依赖右边）。
3. 确认 R1、R2 的候选键，若仍有 BCNF 违规，继续分解。

这种分解能保证：

* **无损分解**（Lossless decomposition）。
* **依赖保持**（尽量保持依赖）。