考试时间:3 月 20 日 下午 1:00 1 个半小时
考试内容:
- **四个部分:**
  - **第一部分是 Normalization(规范化):**
    - **Relation + FD(关系 + 函数依赖)-> Analysis + Decomposition (分析 + 分解)**
  - **第二部分是 Conceptual(概念层):**
    - **True/False 题，判断 ER Diagram 的 Interpretation(解释)**
    - **Narrative -> ER Diagram，文字描述转换为 ER Diagram**
  - **第三部分是 Logical(逻辑层):**
    - **ER Diagram 转换为 Relation (关系模式)**
  - **第四部分是 Physical(物理层):**
    - **Description(给定描述)选择合适的索引(确定是否使用索引)，并解释原因**
  - **附加题**
    - **通过 FDs(函数依赖)找出候选键(key)**
    - **检查给定的 FDs(函数依赖)是否成立或者无效**

考试规则:
- **允许一张参考纸(正反面)**
- **老师会提供一个 ERD Reference Sheet 和 一张额外的空白纸**

第一部分 Normalization(规范化)

1. Normalization(规范化)的目标是什么?
   a) Spurious Tuples(虚假元组)
      i. Spurious Tuples(虚假元组)是什么?
         1. 当一个关系被错误地分解成多个子关系之后，在 Join(连接)回去时，产生了原关系中不存在的新元组，这些错误的数据称之为虚假元组，会影响数据库的正确性。
      ii. 分解关系时，如何导致 Spurious Tuples?
         1. 分解关系时，没有保持原始关系中的函数依赖(FDs)，导致 Join 不能唯一恢复数据
         2. 关系分解后，两个表的 Join 无法正确匹配原始数据，就会导致 Spurious Tuples
      iii. 如何避免 Spurious Tuples?
         1. 确保分解是无损连接(Lossless Join):
            a) 确保两个分解的表可以无误地 Join 回原始关系，而不会引入错误数据
         2. 使用 BCNF 或 3NF 进行合理分解:
            a) 不能随意分解关系，要确保所有的函数依赖(FD)仍然成立
         3. 分解验证
            a) R1 ∩ R2，如果这个交集包含了某个候选键，那么分解是无损的
   b) Additive Decomposition(可加性分解)
      i. 什么是 Additive Decomposition(可加性分解)?
         1. 如果我们把一个关系 R 分解成 R1 和 R2，那么分解后的关系可以通过 JOIN 无损的地恢复成原始关系 R，并且不会丢失信息。也可以理解为，分解后的表仍然可以保持原有的所有数据依赖关系，不会破坏原始数据的完整性
      ii. 为什么 Decomposition(分解)关系时，要保证 Additive(可加性)?
         1. 可能数据丢失(Loss of Information)
         2. 产生虚假元组(Spurious Tuples)
      iii. 如何判断一个 Decomposition(分解)是 Additive(可加)的?
         1. 看分解后的表的交集是否含有候选键(Candidate Key)
   c) Modification Anomalies(修改异常)
      i. 什么是 Modification Anomalies(修改异常)?
         1. 当数据库没有经过良好的 Normalization(规范化)，会产生以下三种异常修改:
            a) 插入异常(Insertion Anomaly)
               i. 当我们想要插入数据时，必须插入额外的，不必要的数据，否则无法插入
            b) 删除异常(Deletion Anomaly)
               i. 当我们删除某些数据时，可能会导致无意中丢失其他重要数据
            c) 更新异常(Update Anomaly)
               i. 当需要更新数据时，需要修改多个重复的地方，如果没有全部修改，数据库会出现不一致
2. Functional Dependencies (FDs, 函数依赖)
   a) 什么是 Functional Dependency(函数依赖)，定义?
      i. 在一个关系 R 中，如果某个属性 X 的值，可以唯一决定另一个属性 Y 的值，我们就说 X->Y，即 Y 依赖于 X
   b) Functional Dependency 和 Key 的关系
      i. 主键是一个特殊的 FD，因为它能唯一确定一行数据的所有属性
   c) 如何判断 Functional Dependency 是否成立?
      i. 看是否存在违反 FD 的情况，如果 X->Y，那么 X 相同时，Y 必须也想通
   d) 如果 Functional Dependency 不成立，如何找到违反的情况?
      i. 找出 X 相同但 Y 不同的行，这些行就违反了 FD
   e) 不同类型的 Functional Dependency
      i. Trivial Functional Dependency(平凡依赖)
         1. X -> Y 且 Y 是 X 的子集。比如 A->A (A, B) -> A 平凡依赖永远成立，没有实质性意义
      ii. Transitive Functional Dependency(传递依赖)
         1. 如果 A->B, B-> C 那么 A->C
      iii. Full Functional Dependency(完全依赖)
         1. 如果 X->Y 但 X 的任何子集都不能单独决定 Y，那么 X->Y 是完全依赖
         2. 比如(学号，课程号)->成绩，就是完全依赖。但是如果学号->成绩，那就是部分依赖，(2NF，需要消除部分依赖)
3. Normal Forms(规范化范式)
   a) 1NF(第一范式)
      i. 1NF(第一范式)要求所有的列都是原子值(Atomic Values)。

- ii. 不能有 Multivalued Atrributes(嵌套表)
- b) 2NF(第二范式)
  - i. 2NF(第二范式)等于 1NF 和消除部分依赖(Partial Dependency)
- c) 3NF(第三范式)
  - i. 3NF(第三范式)等于 2NF 和消除传递依赖(Transitive Dependency)
- d) Decomposition Algorithm(分解算法)
  - i. 找出所有的候选键(Candidate Key)
  - ii. 识别所有的函数依赖(Functional Dependency)
  - iii. 拆分关系，保证无损连接
  - iv. 确保所有的表都符合 3NF

4. Exercise
   - a) 第一题
     - i. Non Trivial FDs
       1. S -> T, TU -> W
     - ii. Candidate Keys
       1. SUV

第二部分 ER Diagram & Mapping

1. **Conceptual Design(概念设计)**
   - a) **Goals**
     - i. **Accurately Represent business requirements 准确表示业务需求:**
       1. **Identify real-world entities，attributes，and relationships to ensure the database accurately reflects business requirements 识别真是世界中的实体，属性和关系，确保数据库能够真是反应业务需求**
     - ii. **Improve Database Maintainability 提高数据库的可维护性:**
       1. **Make the data model intuitive and clear facilitating future maintenance and expansion while avoiding data redundancy and inconsistencies 让数据模型更直观，清晰，方便后续维护和扩展，避免数据冗余和数据不一致**
     - iii. **Avoid Data Anomalies 避免数据异常:**
       1. **Use Normalization techniques to minimize insertion, deletion and update anomalies. 通过规范化设计，检查插入异常，删除异常和更新异常。**
     - iv. **Enhance Database Performance 提高数据库的性能:**
       1. **Design optimized relational structures to reduce unnecessary JOIN operations and improve query efficiency; use appropriate indexing and key design for faster data retrieval 设计合理的关系结构，减少不必要的 JOIN 操作，提高查询效率，合理的索引和键设计优化数据检索速度**
     - v. **Support Physical Database Implementation 支持数据库的物理实现:**
       1. **Ensure that the ER diagram can be mapped to relational database tables, supporting efficient storage and querying. 确保 ER 图可以转换成关系数据库表结构，并支持搞笑村粗和查询**
   - b) **Approach**
     - i. **Top - Down 自顶向下**
       1. **Define High-level abstract entities first, then refine specific attributes and relationships. Suitable for object-oriented modeling 先定义抽象的高层实体，再细化具体的属性和关系。适用于面向对象建模。**
     - ii. **Bottom - Up 自底向上**
       1. **Start with specific data points and generalize higher-level entities and relationships. Suitable for scenarios where data already exists. 从具体的数据点出发，归纳出更高级的实体和关系。适用于已有的数据的情况下推导数据库结构**
     - iii. **Inside – Out/ Mixed Approach**
       1. **Identify key entities first, then expand relationships and attributes gradually. Suitable for complex database designs 先识别关键实体，再逐步扩展关系和属性，适用于复杂数据库设计。**
     - iv. **Summary 结论**
       1. **Represent Business Requirement 表示业务需求**
          - a) **Top-Down From abstract to specific 从抽象到具体**
       2. **Improve Maintainability 提高可维护性**
          - a) **Bottom-Up From specific to general 从具体到抽象**
       3. **Avoid Data Anomalies 避免数据异常**
          - a) **Inside-Out， Identify core entities first，then expand 先确定核心再扩展**
       4. **Optimize Performance 优化性能**
          - a) **Select appropriate relational structures and indexing 选择合适的关系结构和索引**
       5. **Support Database Implementation 支持数据库实现**
          - a) **Design to be mapped to a relational model 设计可以映射到关系模型**
2. **All the notation we covered(ER 图的所有符号和概念)**
   - a) **Entities 实体**
     - i. **Strong 强实体**
       1. **可以独立存在，拥有唯一标识(主键 Primary Key)，表示方法为单矩形框，内部写上实体名**
       2. **Student(SID, Name, Major)，SID 是主键**
     - ii. **Weak 弱实体**
       1. **不独立存在，必须依赖于强实体，没有完整的主键，依赖外键 FK 和部分键（Partial Key)，表示方法为双矩形框，双菱形表示识别关系**
       2. **Dorm(RoomID, Capacity) 依赖于 Student(SID)**
   - b) **Attributes 属性**

i. Composite Attribute 复合属性
   1. 可以拆分成多个子属性，比如 name 可以拆分成 firstname lastname 等。表示方法为椭圆形+子属性分支。
   2. Name(FirstName, LastName)銆
ii. Multivalued Attribute 多值属性
   1. 一个实体可能有多个值属性，如 phonenumbers，双椭圆形，
   2. Employee(Name, {PhoneNumbers})
iii. Derived Attribute 派生属性
   1. 由其他数据计算得出的属性，不直接存储，如 Age 由 Birthdate 计算，表示方法为虚线椭圆形
   2. Age 由 Birthdate 计算
iv. Key 键
   1. 用于唯一表示一个实体的属性，椭圆形+下划线

c) Relationships
i. Cardinality 基数性
   1. 实体之间的关系数量，表示一个实体可以与多少个其他实体关联，常见类型 1:1(在其中一个表添加外键),1:M(在多端添加外键)，M:N(创建中间表，桥接表)
   2. Student 1: M Enrolls M:1 Course
ii. Structure 结构
   1. 决定关系的强制性，确保数据一致性，有全参与(Total Participation，双线连接关系和实体)和部分参与(Partial Participation，单线连接关系和实体)两种类型
   2. Employee 必须属于一个 Department(全参与)，但 Department 可能没有员工(部分参与)
iii. Attributes 关系属性
   1. 关系本身可以拥有属性，存储关于实体之间交互的信息
   2. WorksFor(Employee, Department, {StartDate})

d) Specialization 专门化 /Generalization 泛化
i. Specialization 专门化
   1. 一个父实体拆分为多个子实体
   2. Employee 被细分为 Professor 和 Admin
ii. Generalization 泛化
   1. 多个子实体合并成一个父实体
   2. Car 和 Truck 泛化为 Vehicle

e) When to use? 何时使用这些概念

3. Mapping to Tables (ER 图如何映射到表)
a) General Mapping Steps 映射 ER 图到表
i. 基本规则
   1. 每个强实体变成一个表
      a) 主键直接映射到表的主键字段
   2. 每个弱实体变成一个表
      a) 需要外键连接到其他强实体
      b) 组合部分建形成联合主键
   3. 1:1 关系
      a) 选择一方作为外键存在另一张表中，避免冗余
   4. 1:M 关系
      a) 多端的表添加外键，只想 1 端的主键
   5. M:N 关系
      a) 创建一个新表，存储两张表的主键，作为外键
   6. 多值属性
      a) 创建一个新表，包含实体和主键和多值属性字段
   7. 专门化/泛化

b) Multiple methods for specialization or generalization 多种方法处理专门化/泛化
i. Single Table Inheritance 单表继承
   1. 所有子类的数据存储在同一个表，使用类型字段(Discriminator Column)区分子类
   2. 适用于子类属性较少，查询时经常需要访问所有子类数据
   3. 优点是可以减少表的数量，查询所有子类更简单，插入新子类更方便，适用于数据量不大，继承层次简单的情况
   4. 缺点是，字段冗余因为某些子类的列为空，查询特定子类数据时可能低效，违反 3NF 因为某些字段可能是 NULL

ii. Table per Subclass 子类单表
    1. 为每个子类创建单独的表，主键继承父类
    2. 适用于子类属性比较多，各子类型查询需求较为独立
    3. 优点是避免数据冗余因为每个子类只存自己需要的字段，数据存储更紧凑，适用于继承层次深，子类数据较为独立的情况
    4. 缺点是查询所有子类时需要 JOIN，插入新子类更复杂因为要插入多个表，无法直接查询所有 Employee 数据
iii. Table per Hierarchy 父类子类拆分
    1. 父类存储通用属性，子类存储特定属性，子类表的主键是外键指向父类
    2. 适用于需要同时支持子类查询和所有实例查询
    3. 优点是查询所有子类数据更高效，可以灵活拓展，避免数据冗余
    4. 缺点是子类查询仍然需要 JOIN，可能导致多表 JOIN 影响性能，插入数据稍复杂

4. Exercise
a) 用文字描述下面的 ERD 图
    i. How can you identify an instance of E? 如何标识 E 的一个实例
        1. 主键
            a) Id1 and id2 可能是候选键(FK)，因为他们都是单独连接到 E 的，通常表示唯一标识符
        2. 其他属性
            a) A1，a2，a3，是普通属性，直接存储在 E 表中
            b) D(虚线椭圆)是派生属性(Derived Attribute)，意味着它可以通过其他数据计算得出，不会直接存储
            c) Mv(双圆椭圆)是多值属性(Multivalued Attribute)，需要单独存储在另一个表中
        3. 结论
            a) Id1 和 id2 作为主键来唯一标识 E 的实例
b) 将 E 映射到关系表(Mapping E to Relations)
    i. 主键(Primary Key)是什么?
        1. Id1 是主键，id2 作为唯一
        2. A1,a2,a3 作为普通属性存储
        3. D 作为派生属性不会存储
    ii. 其他键会发生什么变化?
        1. 多值属性 mv 需要单独存储在一个新表中
        2. Id1 作为外键，连接 E 表
        3. Mv_value 存储 mv 的多个值
        4. 复合主键 id1，mv_value 可以确保每个 id1 可以有多个 mv 值

第三部分 Indexing(索引)

1. **What is an Index? What are the potential costs & benefits of using one? 什么是索引? 使用索引潜在的花费和好处是什么?**
   a) **索引是数据库中用于加速查询的数据结构，类似于书籍的目录，可以帮助数据库更快地查询数据，而不是扫描整个表**
   b) **好处是，Faster Queries 加速查询，Lower Disk I/O 减少磁盘 I/O 避免全表扫描，优化排序(Order By)因为索引能加速排序和分组操作，提高查询范围(Between, <, >)的性能**
   c) **缺点是，Storage Overhead，索引占用存储空间，影响插入/更新/删除(Slower write)因为索引也需要维护，并非所有的查询都能收益(Not always useful)，过多的索引可能会影响性能(Too many indexes can slow down write)**
2. **What factors should be considered when choosing whether or not to use an index? 在选择是否使用索引的时候应该考虑哪些因素?**
   a) **在决定是否创建索引时，需要考虑索引的作用，查询需求，数据规模，以及对写入性能的影响**
   b) **适合使用索引的情况，**
      i. **Frequent Queries 查询频繁**
         1. **如果一个列经常船现在 where，orderby groupby 语句中，索引可以显著提升性能**
      ii. **Large Tables**
         1. **小表可以直接遍历，大表需要索引来提升查询速度**
      iii. **Where Clauses 有 Where 过滤条件的查询**
      iv. **Order By/ Group By 排序和分组**
         1. **如果 Order By/Group By 不是索引字段，数据库可能创建临时表进行排序，开销大**
      v. **Foreign Key 外键**
         1. **例如 orders 表的 customer_id 是 customers 表的外键**
   c) **不适合使用索引的情况**
      i. **Small Table 表很小**
         1. **如果表只有几十行，数据库直接扫描全表可能比索引查找更快**
      ii. **High write workload 频繁写入**
         1. **索引需要维护，如果 Insert/Update/Delete 很多，索引会降低写入性能**
      iii. **Low Cardinality Columns 低基数列**
         1. **索引在唯一值多的列上更有效，如果一个列只有几个不同的值，索引帮助不大**
      iv. **使用 Like %...%模式匹配**
         1. **索引无法用于前缀是%的 LIKE 查询**
3. What are clustered indexes? Covering? 什么是聚类索引, 和覆盖索引?
   a) Cluster index 聚簇索引索引
      i. 聚簇索引定义
         1. 数据物理存储按照索引顺序排列(不像是普通索引只是额外的索引结构)
         2. 每个表只能有一个聚簇索引, 因为数据物理上只能有一个顺序
         3. 主键默认是聚簇索引(如果未指定其他索引)
      ii. 什么时候使用聚簇索引
         1. 查询 Where 主要基于 Primary Key
         2. 需要优化查询的范围 (Between,<, >)
         3. 需要避免 Order By 额外排序开销
      iii. 聚簇索引的缺点
         1. 插入新数据可能会导致磁盘页重新排序, 影响写入性能
         2. 更新聚簇索引字段的值代价大, 数据需要移动
         3. 非主键查询可能会变慢, 需要额外的索引来优化
   b) Covering index 覆盖索引
      i. 定义
         1. 索引本身包含了查询所需的所有列, 避免回表查询, 避免额外磁盘 I/O
         2. 适用于 SELECT 只查询部分字段的情况
      ii. 什么时候覆盖索引?
         1. 查询只涉及到少数列, 而不是 SELECT *
         2. 减少回表查询(避免访问主表数据, 提高性能)
         3. 适用于只读查询较多的场景(分析型查询)
      iii. 覆盖索引的缺点
         1. 需要额外的存储空间, 索引变大
         2. 如果表结构频繁变更, 索引维护成本高
4. Why use a hash table vs b+-tree? 什么是哈希索引和 B+树索引?
   a) Hash Index

i. 定义
1. 使用哈希函数将索引键映射到特定位置
2. 适用于精确匹配查询(Exact Match Queries)，但不支持范围查询(Between, <, >)
3. 时间复杂度(1)，查询速度鸡块

ii. 哈希索引的使用场景
1. Exact Match Queries 等值查询
2. 唯一性检查
a) 哈希索引适用于 UNIQUE 约束的字段

iii. 哈希索引的缺点
1. 不支持范围查询
2. 不支持排序
3. 不支持部分匹配查询

b) B+ Tree Index
i. 定义
1. B+树是一种自平衡树结构，用于加速范围查询和排序
2. 适用于大多数数据库查询
3. 时间复杂度 O(log n) 比哈希索引稍慢，但适用范围更广

ii. B+树索引的使用场景
1. Range Queries 支持范围查询
a) B+树可以高效地遍历有序数据，处理范围查询
2. Order By 支持排序
a) B+树索引中的数据是有序的，可以直接用于排序
3. 支持部分匹配查询

iii. B+树索引的缺点
1. 查询 = 可能比哈希索引慢 O(logn)vsO(log1)
2. 占用更多的存储空间
3. 插入/删除时可能需要调整树结构，但比哈希索引稳定

第四部分 Bonus

1. **Function Dependencies(函数依赖)如何确定候选键 Keys**
   a) **如何通过 Function Dependencies(函数依赖)找到候选键**
      i. **列出所有 FD 依赖关系**
      ii. **计算闭包(Closure):找出一个属性(或属性组合)能确定整个关系的最小集合**
      iii. **如果某个属性集的闭包能涵盖所有属性，它就是候选键**
      iv. **例子**
         1. **R(A,B,C,D)**
         2. **FDs A→B，B→C，A→D**
         3. **A 就是候选键?**
2. **State(状态)和 FD(函数依赖)如果判断(Holds)成立或者(Invalidation 失效)?**
   a) **给定一个关系和状态，FD 是否仍然 Holds(成立)?**
   b) **如果某个 FD 失效了，发生了什么?**
   c) **例子**
      i. **R(A,B,C)**
      ii. **FDs A→B**
      iii. **初始数据**
         1. **A B C**
         2. **1 x x**
         3. **1 x y**
         4. **2 y z**
      iv. **此时 A→B 是成立(Holds)的，因为相同的 A 对应相同的 B**
      v. **如果我们插入了一条数据，使 A→B 失效(Invalidation)**
         1. **A B C**
         2. **1 x x**
         3. **1 y y**
         4. **2 y z**
      vi. **由于 A=1 对应 B=x 和 B=y 违反了 A→B 规则，FD 失效了！！！！**

Normalization 规范化

好的数据库设计:minimizes redundancy、improves query efficiency、supports business logic
坏的数据库设计:leads to data inconsistency、redundancy、and maintenance difficulty.
问题领域(Problem Domain): Refers to the business logic and data requirements that the database supports.
正确设计数据库，需要明确: Main Entities、Their Relationships、Business Constraints
如何评估一个 Schema: Data Redundancy、Violate Normalization Rules、Queries Efficient
改进的方法: Apply normalization to eliminate redundancy. Use indexing to improve query performance. Redesign table structures to better align with business logic.

规范化的目标(Objectives of Normalization):下面四个
使数据库模式更具信息性(Make the schema informative): 设计易于理解的 relational schema。实体和关系应分离，避免所有数据塞进一个大表中。表的结构应尽量反映真实世界的概念，这样数据库才能有效管理数据。
最小化信息重复(Minimize info duplication):重复存储数据易导致数据冗余、更新异常和数据不一致问题。
避免修改异常(Avoid modification anomalies):修改异常(Modification Anomalies)是指在修改数据库表时，由于表结构未经过良好的规范化，导致出现意外的副作用(undesired side-effects)。
禁止虚假元组(Disallow spurious tuples):在数据库设计中，如果在不同表之间匹配的属性(如外键和主键)不正确，就可能导致虚假元组(Spurious Tuples)。错误的关系设计 可能会在连接(JOIN)操作时引入无效的数据行。解决方案是确保表与表之间的关系遵循数据库的规范化(Normalization)，正确使用外键和主键的约束。

主要的修改异常包括:下面三个
插入异常(Insertion Anomaly): Difficult or impossible to insert a new row
删除异常(Deletion Anomaly): Updates may result in logical inconsistencies
更新异常(Update Anomaly): Deletion of data representing certain facts necessitates deletion of data representing completely different facts
功能依赖 Functional Dependency (FD): 如果 X 值相同，那么 Y 的值也必须相同。
函数依赖与键(FDs & Keys):1. 不能单纯根据数据表推断 FDs，需要结合实际业务规则，业务规则的不同可能会改变 FDs 的适用性，因此需要谨慎验证。2. 不能仅通过数据实例确定 FDs 是否对所有关系状态都成立，除非知道属性的含义和相互关系。3. 可以通过提供违反 FD 的元组(tuples)来证明某个 FD 不成立

规范化过程(Normalization Process): 1. Submit a relational schema to a set of tests (related to FDs) to certify whether it satisfies a normal form 提交一个关系模式，进行一系列测试(与函数依赖 FDs 相关)，以验证它是否满足某个范式。 2. If it does not pass, decompose into smaller relations that satisfy the normal form 如果不符合该范式，则需要将其分解为较小的关系，以满足该范式。3. The normal form of a relation refers to the highest normal form that it meets 一个关系的范式是指它满足的最高级范式。4. The normal form of a database refers to the lowest normal form that any relation meets 数据库的范式指的是其所有关系中最低的范式。
第一范式(1NF): 1. 属性的域必须只包含原子值，即属性值必须是不可再分的单个值。2. 不能有关系嵌套(不能在关系中再包含关系)。3. 作为扁平化关系模型(flat relational model)的一部分，这一约束是隐式的
第二范式(2NF): 符合 1NF 每个非主属性(Non-Prime Attribute)必须完全依赖于主键(Fully Functional Dependency on Primary Key。不能有部分依赖，即某些非主属性仅依赖于主键的一部分，而不是整个主键。
第三范式(3NF): 3NF 需要满足 2NF，并且 每个非主属性都不能传递依赖于候选键。
分解算法(Decomposition Algorithm):找出所有的候选键(Candidate Key)。识别所有的函数依赖(Functional Dependency)。拆分关系，保证无损连接。确保所有的表都符合 3NF

不同类型的 Functional Dependency:下面四个
平凡依赖(Trivial Functional Dependency):X -> Y 且 Y 是 X 的子集。比如 A->A (A, B) -> A 平凡依赖(implicit constraint)。
非主属性(Non-Prime): 不属于任何候选键的属性
传递依赖(Transitive Functional Dependency):如果 A->B, B-> C 那么 A->C 永远成立，没有实质性意义
完全依赖(Full Functional Dependency):如果 X->Y 但 X 的任何子集都不能单独决定 Y，那么 X->Y 是完全依赖 比如(学号，课程号)->成绩，就是完全依赖。但是如果学号->成绩，那就是部分依赖，(2NF，需要消除部分依赖)

实体(Entity)单矩形: 实体是现实世界中可被区分的对象，可以是一个人，一个地方，一个事件或一个概念。
属性(Attribute)单椭圆形: 属性是描述实体的特性，也是数据库中存储的大部分信息。
关系(Relationship)菱形:关系用于表示不同实体之间的关联，可以携带属性。“学生” 和 “课程” 之间的 “选课” 关系。
关系本身也可以有属性:例如，“选课” 关系可能有 “成绩” 作为属性。(1:1, 1:N, N:M) 1:1 关系的属性可以放到任意一方。1:N 关系的属性只能放到 1 的这一方的实体。N:M 需要用关联表。
实体集(Entity Sets): 指一组具有相同属性的实体。在数据库中，我们用 ER 图 来描述这些实体及其属性。
复合属性(Composite Attributes): 是一种可以进一步细分的属性，即它由多个子属性(sub-attributes)组成。
多值属性(Multivalued Attributes)双椭圆形: 指一个实体的某个属性可以有多个不同的值。
主键属性(Key Attributes)下划线:能够唯一标识一个实体实例的属性，即该属性的值可以区分所有实体对象。
关系模式(Relational Schema)中，如果多个属性被加下划线，它表示这些属性的组合是主键，即:组合键**(Composite Key)**:这些属性一起才能唯一标识一行数据，单独的某个属性不能唯一标识记录。
ER 图(ERDs)中，如果多个属性被加下划线，表示它们是**候选键(Candidate Keys)**，即:每个单独的属性都能唯一标识实体。但最终只能选一个作为**主键(Primary Key)**。
派生属性(Derived Attributes)虚线椭圆:可以通过计算得到的属性，而不是直接存储在数据库中的值
弱实体(Weak Entity)双线矩形:没有主键的实体，不能独立标识自己。它的唯一性必须依赖于某个强实体(Strong Entity)。必须通过关系(Identifying Relationship 双线菱形)和强实体关联才能唯一确定。
专门化(Specialization):一个父实体拆分为多个子实体，子实体拥有额外的属性或者特殊的关系。
泛化(Generalization):多个子实体合并成一个父实体，即从多个具体的实体 抽象出一个更通用的实体。
多重子类型的互斥性(Multiple Subtypes: Disjointedness): Overlap(重叠)一个实体可以属于多个子类型(员工、校友)、Disjoint(互斥)一个实体只能属于一个子类型，不能同时属于多个子类型。(学生，校友)双线继承(Total Completeness)表示 PERSON 必须属于一个子类型，不能缺失。

需求获取(Requirements Elicitation): 下面四个
数据库需要存储哪些主要对象？(What are the main kinds of objects to be stored?) 实体 弱实体，专门化，泛化
对于每个对象，需要存储哪些信息？(What information should be stored for each object?) 属性，关系，键，
对于每条信息，如何定义有效值？(What characterizes a valid value?) 复合属性，多值属性，结构化数据
对象之间的关系如何影响其存在？(Can x exist without y?) 参与度(Total, Partial)，基数(Cardinality 1:1 1:N N:M)

第一步 普通的强实体 Regular (Strong) Entity:1. 对于每个强实体类型 (Strong Entity Type)，创建一个对应的关系(Relation) 该关系包括所有简单属性 (Simple Attributes)。复杂属性 (Composite Attributes) 需要拆分，

仅存储其简单组成部分。2. 如果实体有多个候选键 (Candidate Keys)，需要选择一个作为主键 (Primary Key)。若主键是复合键 (Composite Key)，则该主键由多个属性组合而成。3. 未被选为主键的候选键可以被保留为唯一键 (Unique Keys)。这些次要键在物理存储上对索引 (Indexing) 和查询优化有帮助。

第二步 弱实体 Weak Entity Types: 1. 创建关系表，直接把弱实体的所有简单属性(simple attributes) 作为列加进去。2. 添加外键，在弱实体对应的表中，增加一个外键，指向它的拥有者实体(Owner Entity)。这个外键就是拥有者实体的 主键(Primary Key)。3. 确定主键:主键 = 拥有者实体的主键 + 弱实体的部分关键字(partial key)。这样确保每个弱实体实例都能被唯一标识。

第三步 二元一对一关系映射 Mapping Binary 1-to-1: 方法 1:使用外键(Foreign Key)适用于大多数情况，选择其中一个表(称为 S)，另一个表(称为 T)。如果 S 是"总参与"关系(即每个 S 必须有一个 T)，那么存储更高效，NULL 值更少。在 S 表中:添加这个关系的所有属性。把 T 的主键 添加为 S 的外键。方法 2:合并关系(Merged Relation)适用于两个实体都必须参与关系(Total Participation，即直接把 两个实体合并成一个表。只有当 S 和 T 都是 total participation(即每个 S 都必须有 T，每个 T 也必须有 S)时才适用。这样可以避免外键，提高查询效率，但灵活性较低。

第四步 二元一对多关系映射 Mapping Binary 1-to-N: 方法 1:选择 S 作为 N 端(多的一方)，T 作为 1 端(单的一方)。在 S 中添加 T 的主键作为外键，确保 S 记录能够正确关联到 T。方法 2:除了直接在 S 中添加外键，也可以 创建一个独立的关系表(Relationship Relation)来存储关联信息，但一般不这么做，除非有特殊需求(如关系本身需要额外的属性)。

第五步 二元多对多关系映射 Mapping Binary M-to-N: 1:创建一个新的关系表 S(relationship relation)，由于多对多(M:N)关系不能直接转换，所以 必须创建一个新表 S 来表示这种关系。这个表 S 专门用于存储两个实体之间的 M:N 关系。在某些 ERD 画法中，M:N 关系本身可以被视为一个实体。2:取出 M 端和 N 端实体的主键，并 作为外键(Foreign Key) 添加到 S 表。这两个主键的组合构成 S 的主键(因为一对 M:N 关系中的每一条记录都唯一对应这两个主键)。3:如果 M:N 关系有额外的属性，比如雇佣日期(Hire Date)，这些属性也需要存入 S 表。

第六步 多值属性的映射 Multivalued Attributes: 1:创建一个新关系 S，这个新表专门用于存储多值属性的值。2:在 S 中添加原实体的主键作为外键。这个外键用于指向原实体，保持关系。3:在 S 中添加多值属性，如果这个属性是复合属性(Composite Attribute)，需要拆分成多个简单属性存储。4. S 表的主键是原实体的主键 + 多值属性(组合键)这样可以确保一个实体的每个属性值唯一，避免重复。

第七步 继承关系 Specialization/Generalization: 方法 1:Multiple Relations – Subclass and Superclass(父类和子类分别建表)适用于大多数情况。父类表存储通用属性，每个子类表存储特定属性，并且包含指向父类的外键(假设父类中的 ID 是唯一的)。方法 2:Multiple Relations – Subclass Only(只有子类建表)适用于互斥(disjoint)的情况。不创建父类表，每个子类独立存储所有属性(包括父类属性)。方法 3:Single Relation with One Type Attribute(单表+类型属性)适用于互斥(disjoint) 的情况。所有子类共享一个表，增加一个 type 字段标识对象属于哪个子类。方法 4:Single Relation with Multiple Type Attributes(单表+多个类型属性)适用于可以重叠(overlapping) 的情况。在一个表中使用多个布尔字段(或枚举类型)，表示对象属于哪些子类。**总结:**如果子类之间互斥(disjoint):可以用独立的子类表或单表+类型字段。如果子类可以重叠(overlapping):建议使用单表+多个类型字段。如果查询子类时不想 JOIN，可以用只有子类的表方法。

物理设计与调优 Physical Design & Tuning:

影响数据库性能的因素(Factors that Influence Performance):下面四个

查询中的属性(Attributes in Queries): Queried 多→适合索引(Indexed)。Updated 多→不适合索引(因为索引更新会增加数据库负担)。Unique(唯一值属性) →可以作为索引，如主键(Primary Key)唯一约束(Unique Constraint)

查询/事务的相对频率(Relative Frequency of Queries/Transactions): 80/20 规则(Pareto Principle)→80%的查询可能只涉及 20%的数据。更新操作(Updates)→频繁更新的表可能需要避免索引过多，以减少维护成本

查询/事务的性能约束(Performance Constraints xx): 例查询必须在 X 秒内完成，需要优化执行计划和索引

数据库性能分析(Profiling): 存储分配(Storage allocation) →数据如何在磁盘上存储。 I/O 性能(I/O performance) →读取和写入磁盘的效率。查询执行时间(Query execution time) →主要关注 SQL 语句的执行效率

数据库优化的一般原则(As a general rule with RDBMS's): 先保证正确性，再收集数据，再进行性能优化。

索引（Index）是什么？： 持久化的数据结构 (Persistent Data Structure) 。查询性能的主要优化机制 (Primary mechanism for improved query performance) 。索引的权衡 (Trade-offs of Indexing)

索引的好处和坏处： 好处是，Faster Queries 加速查询，Lower Disk I/O 减少磁盘 I/O 避免全表扫描，优化排序(Order By)因为索引能加速排序和分组操作，提高查询范围(Between、<, >)的性能。缺点是，Storage Overhead，索引占用存储空间，影响插入/更新/删除(Slower write)因为索引也需要维护，并非所有的查询都能收益(Not always useful)，过多的索引可能会影响性能(Too many indexes can slow down write)

如何选择索引来创建： 表的大小，全局扫描成本高。数据分布（选择性 Selectivity），选择性高（distinct value 多），索引更有效，可以快速定位特定的行，反之即是有索引，查询可能仍涉及大量行。确定查询更新负载，查询多适合索引，更新多减少索引，分析查询模式然后优先对那些查询最频繁的列建立索引。

基数 Cardinality， 指某一列中不同值的数量，基数越高，索引的潜在价值越大。

选择性 Selectivity = 100% * Cardinality / # rows，选择性越高索引查询性能越好，每个索引键匹配行数少。选择性越低，如果 Yes/No，索引优化效果不明显。

聚簇索引(Clustered Index): 影响磁盘上物理存储顺序，数据按照索引列顺序存储在磁盘上。一个表最多只能有一个聚簇索引，通常默认在主键上创建。适用于范围查询(range queries)和排序查询(O 正，R 反)，因为数据物理上是有序的。查询效率高，读取一段连续数据块时候磁盘 I/O 较小，插入删除时需要重新排序，+IO

非聚簇索引（Non-clustered Index）： 不会改变数据的物理存储顺序，仅维护逻辑顺序。一个表可以有多个非聚簇索引。查询时需要额外的磁盘读取：索引存储的是指向数据的指针，查询时需要先在索引中查找，再跳转到实际数据位置。适用于频繁的查找操作，但不适用于大量范围查询。

覆盖索引（Covering Index）： 包含查询所需数据的索引。优点查询不再需要回表，减少 IO 操作，适用于只涉及部分列。缺点是占更多存储空间，索引更新成本高，如查询设计费索引列，仍需回表。

B+ 树（B+-Trees）： B+ 树是一种自平衡搜索树，用于数据库索引和文件系统。所有数据存储在叶子节点，内部节点仅存索引，减少磁盘 I/O。叶子节点有序排列，并通过链表连接，适用于范围查询

哈希表（Hash Table）： 哈希索引适用于等值查询，查询速度极快（O(1) 平均复杂度。不支持范围查询，无法用于 >、<、BETWEEN 等查询。可能会导致哈希冲突，需要额外处理。常用于缓存、高速查询。

FD->Key 列出所有 FD，找出 1 个属性(组合)能确定整个关系的集合，R(A,B,C,D) FDs A→B，B→C，A→D A 就是候选键