数据类型

结构化数据（Structured Data）：具备严格的模式，可被组织进关系型数据库中。

非结构化数据（Unstructured Data）：无需预先定义，数据格式不固定。

文件系统与数据库系统的对比（File Systems vs Database Systems）

文件系统缺点包括：数据冗余、不一致性、数据隔离、完整性问题、并发访问问题。

1. 数据冗余 (Data Redundancy)

- 专业定义：同样的数据在多个文件中重复存储

- 造成的问题：存储空间浪费、数据更新困难、可能导致数据不一致

2. 数据不一致性(Data Inconsistency)

- 专业定义：同一数据在不同位置的多个副本之间存在矛盾

- 例如：同一客户信息在不同文件中的地址不同

3. 数据隔离(Data Isolation)

- 专业定义：数据分散在不同文件中，难以访问和组合查询

- 影响：难以生成综合报表或进行复杂查询

4. 完整性问题(Data Integrity Problems)

- 专业定义：难以在文件系统层面实施统一的完整性约束

- 包括：

- 实体完整性 (Entity Integrity)

- 参照完整性 (Referential Integrity)

- 域完整性 (Domain Integrity)

5. 并发访问问题(Concurrent Access Problems)

- 专业定义：多用户同时访问和修改数据时的同步控制问题

- 英文表述：Concurrent Access Control

- 相关概念：

- 并发控制 (Concurrency Control)

- 死锁 (Deadlock)

- 事务处理 (Transaction Processing)

这些问题在现代数据库管理系统(DBMS)中通过以下机制得到解决：

1. 数据字典(Data Dictionary)：控制冗余

2. 事务管理(Transaction Management)：保证一致性

3. 统一管理(Centralized Management)：解决隔离问题

4. 约束机制(Constraint Mechanism)：保证完整性

5. 并发控制(Concurrency Control)：处理并发访问

数据库管理系统的特性

数据独立性（Data Independence）

高效的数据访问（Efficient Data Access）

数据完整性与安全性（Data Integrity and Security）

数据管理（Data Administration）

并发访问与崩溃恢复（Concurrent Access and Crash Recovery）

数据库设计（Database Design）

概念设计（Conceptual Design）：利用工具进行需求表示，便于维护和转化为数据库实现。

逻辑设计（Logical Design）：通过概念设计转换为数据模型并实现于DBMS。

物理设计（Physical Design）：进一步规范数据库的存储与访问。

数据库语言（Database Languages）

数据定义语言（DDL）：用于定义概念模式。

数据操作语言（DML）：用于请求和操作数据，有非过程性DML（如SQL）和过程性DML。

数据库用户类型：

数据库管理员（Database Administrator，DBA）：负责安全、授权、数据恢复和数据库调优。

应用程序员（Application Programmer）：实现具体需求。

最终用户（End User）：实际使用数据库的用户。

数据库模型：

数据模型的级别包括：高层次或概念模型（如ER模型）、实现模型（如关系模型）和物理模型（低层次）。

数据库管理系统类型：

主要包括关系型（Relational）、键值型（Key/Value）、图形型（Graph）、文档型（Document）、列族型（Column-family）。

数据库历史（History of Database Systems）

1960s：通用数据库管理系统（Integrated Data Store）。

1970s：关系模型的提出（Edgar Codd）。

1980s：SQL标准化及事务概念提出（Jim Gray）。

1990s至今：关系型数据库、NoSQL、大数据分布式处理的发展。

数据建模 (Data Modelling):

概念模型 (Conceptual Model)：抽象的高层数据模型，例如 ER 模型和 ODL (Object Data Language)，用户友好。

逻辑模型 (Logical Model)：为具体的数据库管理系统(DBMS)实现的模型，例如关系模型。

物理模型 (Physical Model)：在具体 DBMS 内部的文件存储方式。

设计阶段 (Design Stages): 概念设计 (Conceptual Design)、逻辑设计 (Logical Design)、物理设计 (Physical Design)

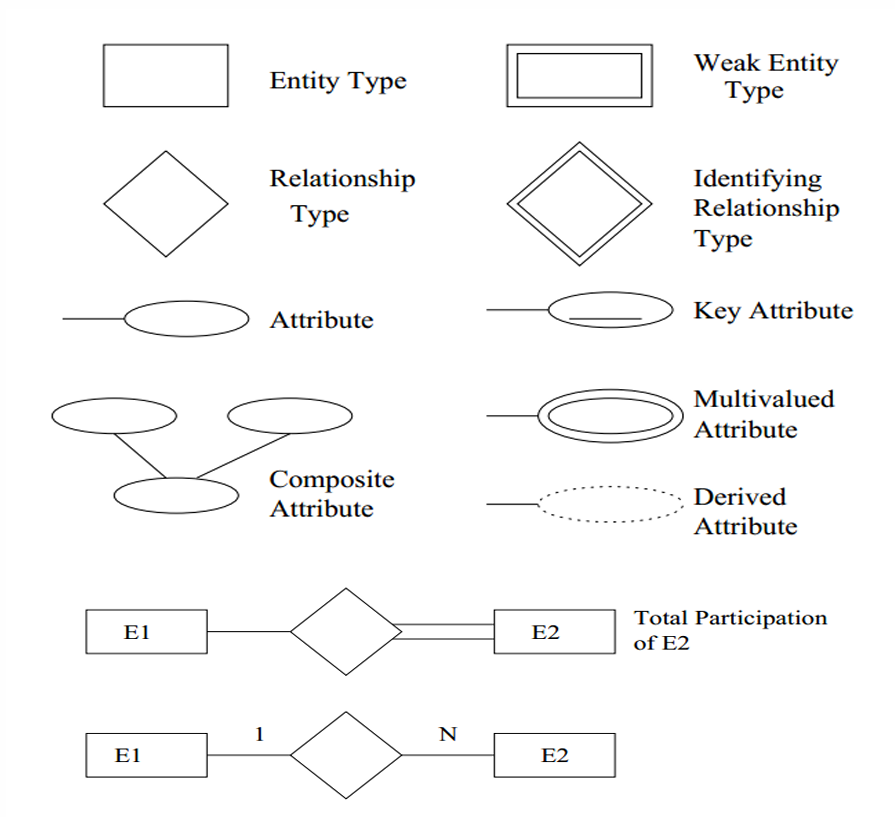
**实体-关系模型 (Entity-Relationship Model, ER):**

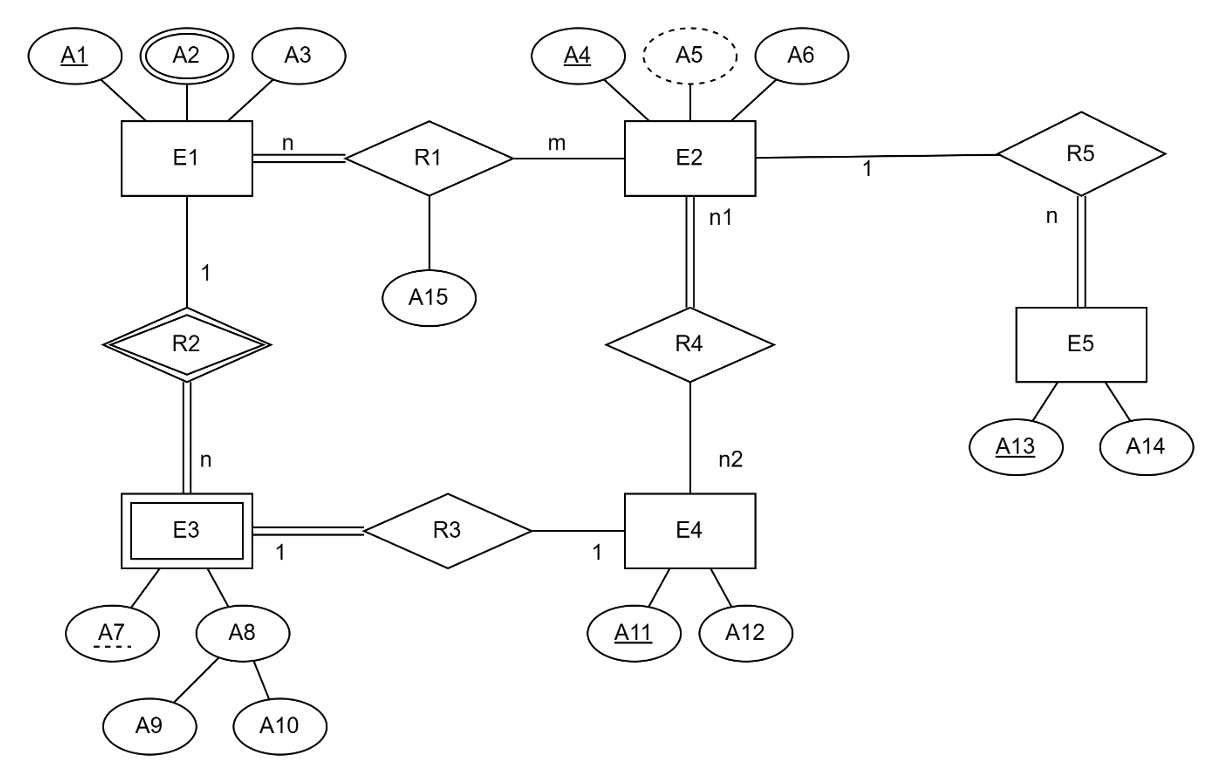
实体 (Entity)：描述感兴趣对象的属性集合。

关系 (Relationship)：实体之间的关联。

属性 (Attribute)：描述实体某种特性的具体数据项。

ER 图 (ER Diagram, ERD)：用于描述实体及其属性的图形化表示。





属性类型 (Attribute Types):

简单属性 (Simple Attribute)：不可再分的属性。

复合属性 (Composite Attribute)：可分为更小的子属性，例如地址 (Address) 可以分为街道 (Street) 和郊区 (Suburb)。

多值属性 (Multi-Valued Attribute)：可以有多个值的属性，例如衬衫的颜色。

派生属性 (Derived Attribute)：可以从其他存储属性推导得出，例如年龄可以从出生日期推导。

实体类型 (Entity Type)：定义具有相同属性的实体集合。在 ER 图中用矩形框表示。

键 (Keys):

超键 (Super Key)：能唯一标识实体实例的一组属性。

候选键 (Candidate Key)：最小的超键，即没有子集能成为超键。

主键 (Primary Key)：由数据库设计者选择的候选键，用于唯一标识实体。

弱实体 (Weak Entity)：没有自己的主键，通常通过部分键（与所有者实体关联的关系）进行标识。

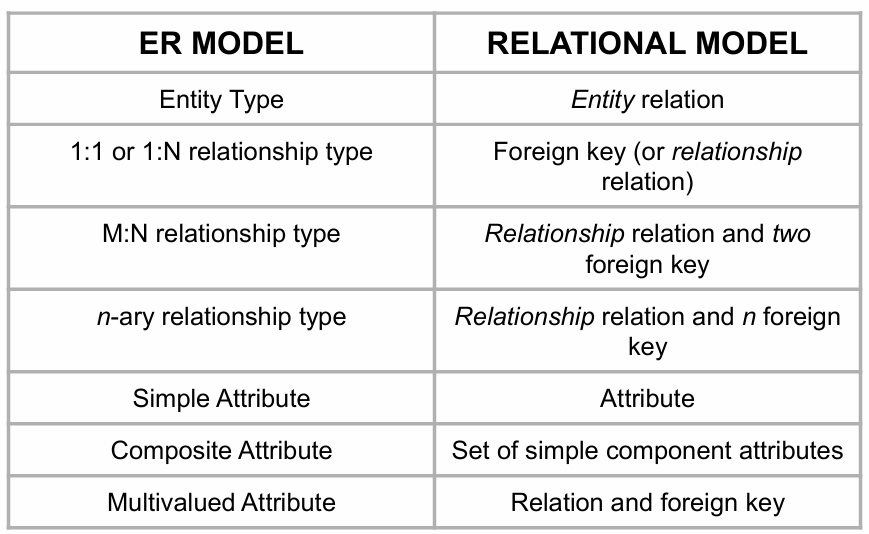
关系约束 (Relationship Constraints):

基数约束 (Cardinality Constraint)：实体参与关系实例的数量限制，如一对一 (1:1)、一对多 (1:N)、多对多 (N:M)。

参与约束 (Participation Constraint)：实体实例是否必须参与某个关系，例如全部参与 (Total Participation) 和部分参与 (Partial Participation)。

关系属性 (Relationship Attributes)：关系可以有自己的属性，例如研究人员在项目上的工作时间。

关系的度 (Degree of Relationship)：参与关系的实体数量，例如二元关系 (Binary Relationship) 和三元关系 (Ternary Relationship)。



**关系代数（Relational Algebra）**

1. **选择操作 (SELECT)**

* 选择满足特定条件的元组。
* 记号：𝜎<选择条件>(R)，如 𝜎 age≤24(Student) 表示选择年龄小于等于 24 的学生。

1. **投影操作 (PROJECT)**

* 用于选择关系中的部分属性（列），并删除重复的元组。
* 记号：𝜋<属性列表>(R)，如 𝜋 name, age(student) 表示只选择学生关系中的姓名和年龄属性。

1. **并集操作 (UNION)**

* 将两个关系中的元组合并，结果包含 R 或 S 或者两者都包含的元组。
* 记号：R ∪ S，要求 R 和 S 具有相同的属性集。

1. **交集操作 (INTERSECTION)**

* 返回两个关系中都包含的元组。
* 记号：R ∩ S，要求 R 和 S 具有相同的属性集。

1. **差集操作 (DIFFERENCE)**

* 返回在关系 R 中存在但不在关系 S 中的元组。
* 记号：R - S，要求 R 和 S 具有相同的属性集。

1. **笛卡尔积操作 (CARTESIAN PRODUCT)**

* 产生 R 和 S 中所有元组的组合，结果的属性包含 R 和 S 的所有属性。
* 记号：R × S，如 𝑅 × 𝑆 表示 R 和 S 的所有可能组合。

1. **连接操作 (JOIN)**

* 连接是用于将两个关系中相关的元组合并为单个“更长”元组。
* **θ连接 (Theta-join)**: 条件连接，记号：R ⋈<连接条件> S，如Student ⋈ CourseID=课程ID Course。
* **等值连接 (Equi-join)**: 仅使用等于比较的连接，记号同 θ连接，如Student ⋈ 导师ID=ID 导师。
* **自然连接 (Natural Join)**: 基于属性名和域相同的属性进行等值连接，重复的属性只保留一个，如Student ⋈ Enrolment。

1. **除法操作 (DIVISION)**

* 用于找出 R 中与 S 中的所有元组关联的元组。
* 记号：R ÷ S，通常用于回答“哪些课程被所有部门开设”之类的问题。

假设我们有以下两个关系：

1. **R (学生, 课程)**：表示每个学生选修的课程。

学生 课程

---- ----

张三 数学

张三 物理

李四 数学

李四 物理

王五 数学

1. **S (课程)**：表示所有的课程。

课程

----

数学

物理

**问题**：找出选修了**所有课程**的学生。

**计算除法 R ÷ S**

在这个例子中，我们想找到那些选修了所有课程（即数学和物理）的学生。

* **R ÷ S 的结果**：

学生

----

张三

李四

1. **重命名操作 (RENAME)**

* 用于更改关系或属性的名称。
* 记号：ρ<新名称>(R)，如 ρ(监护人,孩子)(家人) 表示将关系“家人”的属性“父母”重命名为“监护人”。

1. **聚合操作 (Aggregate Operators)**

* 用于求取诸如“员工工资总和”或“学生平均年龄”等聚合信息。
* 使用记号 γ，例如 γ(SUM(工资))(员工) 表示计算员工工资的总和。

**函数依赖**

**1. 定义**

* **函数依赖**描述了关系中不同属性之间的相互关系。
* 如果关系中任意两个元组在属性集 α 上的值相同，那么在属性集 β 上的值也相同，则称 α **函数决定** β，记作 α → β。
* 示例：
  + ID → Name 表示在所有元组中，若 ID 相同，则 Name 也相同。

**2. 作用与意义**

* 函数依赖用于：
  + **描述属性之间的语义关系**，例如主键和其他属性之间的关系。
  + **作为数据库设计的约束**，确保数据的一致性和完整性。
  + **消除冗余**，通过合理的分解将关系划分成符合规范化形式的子关系，减少数据重复。

**3. 函数依赖的种类**

* **完全函数依赖**：若 α → β 且 β 不能仅由 α 的子集唯一确定，则称 β **完全函数依赖**于 α。
* **部分函数依赖**：如果 β 仅能由 α 的某个子集唯一确定，则称 β 对 α **部分依赖**。
* **传递函数依赖**：如果 α → β 且 β → γ，则可得出 α → γ。这种情况称为**传递依赖**。

**4. 冗余与更新异常**

函数依赖直接关系到**冗余数据**和**更新异常**的问题：

* **插入异常**：插入新数据时，需要提供冗余信息，可能因为无相关数据而被迫使用 NULL。
* **删除异常**：删除某些元组会导致相关重要信息的丢失。
* **修改异常**：修改某个属性的值时，可能需要修改多行数据，增加了出错的风险。

**5. 设计函数依赖**

* 函数依赖不能直接从一个关系的具体实例中推断出来，而是基于**对属性语义的理解**。
* 设计良好的函数依赖有助于判断关系是否满足规范化条件，并决定如何对关系进行分解，以减少冗余和异常。

**6. 阿姆斯特朗公理（Armstrong's Axioms）**

* 阿姆斯特朗公理是一组推理规则，用于从已知的函数依赖中推导其他依赖：
  1. **自反性（Reflexivity）**：如果 β ⊆ α，则 α → β 成立。
  2. **增广性（Augmentation）**：如果 α → β，则对任何属性集 γ，有 γα → γβ。
  3. **传递性（Transitivity）**：如果 α → β 且 β → γ，则 α → γ。
* **其他推理规则**：
  1. **加法性（Additivity）**：若 α → β 且 α → γ，则 α → βγ。
  2. **伪传递性（Pseudo-Transitivity）**：若 α → β 且 γβ → δ，则 αγ → δ。

**7. 计算属性闭包（Closure of Attributes）**

* **属性闭包（α+）**：指的是给定属性集 α，通过函数依赖推导出所有由 α 函数确定的属性集。
* **计算属性闭包的方法**：
  + 初始令 result = α。
  + 循环遍历每个函数依赖 β → γ，如果 β ⊆ result，则将 γ 加入 result，直到 result 不再变化。

**8. 计算候选键（Candidate Key）**

* **候选键**是能够唯一标识关系中元组的最小属性集。
* **计算候选键的方法**：
  + 设定一个包含所有属性的初始集合 X。
  + 通过不断移除 X 中的属性，保留那些能够保持 X+ 覆盖整个关系的属性，最终得到的最小 X 就是候选键。

**9. 闭包与函数依赖推导**

* **函数依赖闭包（F+）**：指的是从给定的函数依赖集合 F 推导出的所有可能的函数依赖。
* 通过计算闭包 F+，可以确定哪些新的函数依赖是由原始集合 F 推导出来的。

**10. 实例练习**

给定关系 R(A, B, C, G, H, I) 和函数依赖 F = {A → B, A → C, CG → H, CG → I, B → H}，可以使用阿姆斯特朗公理推导其他的函数依赖，例如：

* A → H 通过传递性从 A → B 和 B → H 推导得出。
* AG → I 通过增广和传递性推导得出。

**11. 应用示例与依赖图**

* **依赖图**：用于可视化地表示属性之间的函数依赖关系，横线代表函数依赖的左侧，箭头指向依赖的右侧属性。

**内存管理**

**一、存储管理与内存层次**

数据库系统中的数据存储在多个层次中，这些层次决定了数据的访问速度、成本和持久性。

**1. 主存 (Primary Storage)**

* **特点**：
  + 存储在主内存中（例如 RAM），访问速度非常快，时间量级在纳秒（ns）范围。
  + 主内存价格昂贵，容量有限，通常不够大，无法存储整个数据库。
  + 容量较小，且是易失性存储，意味着断电或者系统崩溃时，主存中的数据将会丢失。
* **作用**： 主存主要用于处理查询和计算，操作需要的数据首先要从磁盘读入主存。数据库管理系统（DBMS）在主存中维护一个**缓冲池（Buffer Pool）**，将需要经常访问的数据放入其中以减少磁盘的读取次数。

**2. 次级存储 (Secondary Storage)**

* **磁盘存储**：
  + 数据通常存储在磁盘上，典型的如机械硬盘（HDD）或者固态硬盘（SSD）。
  + 访问速度比主存慢得多，但比主存价格便宜，可以用来存储大量数据。
* **特点**：
  + **数据持久性**：次级存储的数据不会因电源故障而丢失。
  + 访问速度较慢，因为磁盘的读取速度受物理硬件的影响，包括磁盘旋转和机械臂的移动等因素。
* **磁盘存储架构**：
  + 磁盘通常由多个**盘片（Platter）组成，每个盘片包含多个磁道（Track）**，每个磁道又分为若干**扇区（Sector）**。
  + \*\*块（Block）\*\*是存储和数据传输的基本单位，磁盘上的数据会按块读写。每次读写操作，磁盘都会以块为单位从磁盘移动数据到主存。即使只需要一个记录，也必须读取整个包含该记录的块。

**3. 存储访问机制**

* **块与页的概念**：
  + 在数据库中，磁盘存储单位称为**块（Block）或者页（Page）**。它是存储分配和数据传输的单位。
  + 一个**页**可以包含多个数据记录，因此数据库操作通常以页为单位进行。
* **访问机制**：
  + 数据要被操作时必须先从磁盘读取到主存中，数据的读写速度受到磁盘物理限制，如**寻道时间**（找到合适的磁道）、**旋转延迟**（找到合适的扇区）和**传输时间**（读取数据）等。
  + 磁盘访问时间的影响主要来自寻道时间和旋转延迟，这两者会导致磁盘的随机访问性能不如顺序访问。

**二、缓冲池管理**

缓冲池是数据库系统中用于减少磁盘 I/O 操作的重要机制，以下是详细的缓冲池管理机制及相关概念：

**1. 缓冲池（Buffer Pool）**

缓冲池是主存中用于缓存磁盘块的内存空间，由\*\*页槽（frames）\*\*组成，每个页槽可以缓存一个磁盘块。

* **作用**： 缓冲池用于存储从磁盘读取的数据块，当数据库管理系统需要频繁访问某些数据时，可以通过缓冲池直接访问这些数据，避免反复从磁盘读取，提高系统效率。

**2. 缓冲区管理的操作**

* **请求块 (request\_block)**：
  + 当上层需要访问某个数据块时，首先会检查该块是否已经在缓冲池中。如果该块在缓冲池中，则可以直接使用，避免额外的 I/O 操作。
  + 如果该块不在缓冲池中，则需要从磁盘读取数据到缓冲池中。
* **释放块 (release\_block)**：
  + 当一个数据块使用结束后，系统会调用**释放块**操作来标记该块不再使用，以便将来可以被替换。
  + 如果块数据被修改过，还需要使用**写块 (write\_block)** 操作将修改后的数据写回磁盘。
* **缓冲池的组成**：
  + 每个缓冲池页槽（frame）中包含一些管理信息，例如：
    - **脏位（Dirty Bit）**：指示该页槽是否被修改过。如果被修改过，那么在将其从缓冲池中移除时，需要将数据写回磁盘。
    - **固定计数（Pin Count）**：记录当前页槽被多少个事务使用。如果计数大于零，则该页槽不能被替换。

**三、页替换策略**

当缓冲池中没有空闲帧时，系统需要决定应该替换哪一个现有页。数据库管理系统通常使用不同的替换策略来优化性能：

1. **最近最少使用 (LRU)**：
   * 替换最久未使用的页。该策略通过维护一个链表，记录页最近的使用情况，越早使用的页会被移到链表的尾部。
   * **优点**：在某些应用中（如经常访问的数据页比较稳定），LRU 策略表现较好。
   * **缺点**：在“顺序访问”场景中，LRU 策略可能表现较差（称为“**顺序泛滥（Sequential Flooding）**”），每次访问新的数据页都会导致缓存命中率降低。
2. **最常使用 (MRU)**：
   * 替换最近使用的页，通常用于“栈”模式的工作负载。
   * **优点**：适用于重复扫描的场景，比如当同一组数据被反复访问时，MRU 能有效提高命中率。
   * **缺点**：在非重复的访问模式下性能可能较差。
3. **先进先出 (FIFO)**：
   * 按照页进入缓冲池的顺序进行替换，最先进入的页最先被替换。
   * **优点**：实现简单，不需要记录复杂的访问历史。
   * **缺点**：可能会替换掉仍然需要的页，导致较低的缓存命中率。
4. **随机选择 (Random)**：
   * 随机选择缓冲池中的页进行替换。
   * **优点**：实现简单，可以避免某些替换策略中的“最坏情况”。
   * **缺点**：性能无法保证，缓存命中率完全依赖于随机性。

**例题：页替换策略的应用**

假设数据页有 P1, P2, P3, P4，查询顺序为：

* Q1: 读取 P1；Q2: 读取 P2；Q3: 读取 P3；Q4: 读取 P1；Q5: 读取 P2；Q6: 读取 P4。
* 对于不同策略：
  + **LRU 策略**：当读取 P4 时，需要替换掉最久未使用的页（例如 P3）。
  + **MRU 策略**：最近使用的页（例如 P2）可能会被替换。

从例子中可以看出，不同策略在不同访问模式下的表现差异较大，没有一种策略能够在所有情况下表现最佳。

**四、磁盘空间管理**

**1. 磁盘访问时间**

磁盘的访问时间由多个因素组成，包括：

* **寻道时间**：磁头移动到目标磁道的时间。
* **旋转延迟**：磁盘旋转使得目标扇区到达磁头位置所需的时间。
* **传输时间**：将数据块读写到内存中所需的时间。

磁盘的随机访问时间通常受寻道时间和旋转延迟的影响，因此顺序访问通常比随机访问更快。

**2. 提高磁盘访问效率的方法**

* **聚簇 (Clustering)**：将经常一起访问的记录放在同一个块中，以减少多次读取操作。
* **位图 (Bitmap)**：用来记录磁盘中的空闲块，方便快速查找可用空间。

**五、记录管理**

记录在数据库系统中以块的形式存储，可以是固定长度或者可变长度。

**1. 固定长度记录**

* **特点**：每个字段都有固定的长度，便于空间管理和定位。
* **优势**：由于每个记录大小固定，数据库系统可以通过计算偏移量快速定位记录。
* **劣势**：可能浪费空间，如果某些字段经常为空，或者字段长度远大于实际需要的数据。

**2. 可变长度记录**

* **特点**：某些字段可能具有可变长度，例如姓名、地址等文本字段。
* **存储方式**：
  + **前缀长度 (Prefix Length)**：在字段前记录长度，方便读取。
  + **字段终止符 (Delimiter)**：使用特殊字符（如 '/'）标识字段结束。
  + **偏移数组 (Offset Array)**：使用偏移数组记录字段在记录中的位置。

**3. 插槽页管理**

* **插槽页（Slotted Page）**：
  + 用于管理存储在页中的多个记录。插槽页包含一个插槽目录（Slot Directory），该目录用于记录每个记录的开始位置和长度。
  + **插入和删除操作**：插入新记录时，会在插槽目录中创建新的条目；删除记录时，标记相应插槽为空，其他记录不受影响。

**六、索引**

索引用于提高查询效率，帮助数据库系统更快地定位需要的数据。索引的类型和使用场景如下：

**1. B+ 树索引**

* **特点**：B+ 树是一种平衡树，叶子节点包含指向数据记录的指针。
* **应用场景**：适合进行范围查询和等值查询，尤其在数据有序存储时，B+ 树能够通过叶节点的链表实现快速的顺序扫描。

**2. 哈希索引**

* **特点**：哈希索引使用哈希函数将键值映射到特定的桶中，桶中存储对应的索引数据。
* **应用场景**：适合进行等值查询（如查找特定学生的记录），因为哈希函数能将键快速映射到特定位置。但它不适合范围查询，因为哈希映射无法保留数据的有序性。

**七、缓冲区替换策略的性能比较**

\*\*缓存命中率（Cache Hit Rate）\*\*是评估缓存性能的一个关键指标，表示有多少请求可以通过缓存直接提供而不需要从磁盘读取。在特定访问模式下，不同策略的表现会有所不同：

* **顺序泛滥（Sequential Flooding）**：在顺序访问大量页的情况下，LRU 策略表现不佳，每次新页的读取都会导致旧页的移除，导致缓存命中率下降。
* **重复访问**：在重复访问同一组数据的情况下，MRU 策略可以显著提高命中率。

**事务管理**

**一、事务管理**

事务是数据库系统中一个执行单元，它包含了一组对数据库的操作，这些操作要么全部执行成功，要么全部执行失败，确保数据的完整性和一致性。

**1. 事务的ACID属性**

ACID 是事务管理中的核心原则：

* **Atomicity（原子性）**：
  + 事务是不可分割的单位，所有操作要么全部执行，要么全部不执行。系统需要在发生失败时回滚未完成的操作，确保数据库不被部分更新。
  + 例如，在从账户A转移$50到账户B的事务中，如果在“写入账户B”的操作前失败，系统必须撤销账户A的变化，以确保原子性。
* **Consistency（一致性）**：
  + 每个事务的执行必须使数据库从一个一致状态变为另一个一致状态。在执行事务的过程中，数据库可能处于临时的不一致状态，但事务结束后，必须使得所有约束都得到满足。
  + 例如，银行转账中，无论转账过程如何，最终账户A和B的余额总和应该保持不变。
* **Isolation（隔离性）**：
  + 即使多个事务同时执行，它们的执行效果应该与某一个串行执行的事务序列的结果相同。事务的中间状态对其他事务不可见。
  + 例如，如果一个事务修改了账户A，但尚未完成对账户B的更新，则其他事务在此期间不能看到账户A的变化，否则会看到不一致的数据。
* **Durability（持久性）**：
  + 一旦事务提交，修改的数据应该持久保存，即使系统崩溃也不应该丢失。这是通过将事务的日志持久化到磁盘上来实现的。

**二、事务状态**

事务在执行过程中会经历以下几个状态：

1. **Active（活跃）**：事务在执行中。
2. **Partially Committed（部分提交）**：事务的所有操作都已执行，但还未完成持久化。
3. **Failed（失败）**：事务执行过程中出现了错误。
4. **Aborted（中止）**：事务由于失败而被回滚到初始状态，可以选择重新启动或终止。
5. **Committed（提交）**：事务完成了所有操作，并且所有的修改已被持久化。

**三、并发控制与调度**

并发控制的目标是确保多个事务并发执行时不破坏数据库的一致性，同时提高系统的并行性。

**1. 调度（Schedule）与可串行化**

* **调度（Schedule）**：调度是并发事务的操作执行顺序。
* **串行调度（Serial Schedule）**：所有事务的操作按顺序依次执行，不存在交错。
* **可串行化（Serializable）**：如果一个调度与某个串行调度结果等效，则称其为可串行化的。通过检测冲突等价性可以判断调度的可串行化。

**2. 冲突可串行化（Conflict Serializability）**

* **冲突操作**：
  + 例如，如果两个事务中的操作访问相同的数据项且至少有一个是写操作，则称它们是冲突的。
  + 对于非冲突的操作，可以通过交换操作的顺序来保持调度等价。
* **冲突可串行化的检测**：
  + 通过构建\*\*优先图（Precedence Graph）\*\*来检测可串行化，如果图中无环，则调度是可串行化的。

**四、锁机制与并发控制协议**

为了保证隔离性，数据库使用锁机制来控制并发事务的操作顺序。

**1. 锁的种类**

* **共享锁（Shared Lock，S锁）**：允许多个事务同时读数据，但不允许写。
* **排他锁（Exclusive Lock，X锁）**：允许事务读取和写入数据，其他事务不能获得该数据的锁。

**2. 锁协议**

* **两阶段锁协议（Two-Phase Locking, 2PL）**：
  + **增长阶段**：事务可以申请新锁，但不能释放任何锁。
  + **收缩阶段**：事务释放锁，但不能申请新锁。
  + 2PL可以保证调度是可串行化的，但可能会导致**死锁**。

**3. 死锁与死锁预防**

* **死锁（Deadlock）**：多个事务相互等待对方持有的锁，导致无法继续执行。
* **死锁预防**：
  + **等待超时**：如果一个事务等待超过一定时间，则认为其进入了死锁状态，系统强制中止该事务。
  + **等待图检测**：系统通过构建等待图（Wait-for Graph）来检测死锁，如果存在环则发生死锁。

**五、事务恢复与数据库日志**

在事务执行失败时，数据库必须能够回滚事务的操作以恢复到一致状态，这通过维护\*\*系统日志（System Log）\*\*实现。

**1. 系统日志**

* **日志记录**：
  + **[start, T]**：记录事务T的开始。
  + **[write, T, X, old\_value, new\_value]**：记录事务T对数据项X的修改，包括修改前后的值。
  + **[commit, T]**：记录事务T的提交，表明T的所有修改都可以永久保存。
  + **[abort, T]**：记录事务T的中止。

**2. 写前日志（Write-Ahead Logging, WAL）**

* **写前日志策略**：在将数据页写入磁盘前，必须先将对应的日志记录持久化。这确保了即使系统崩溃，日志也能提供足够的信息来恢复数据库。
* **UNDO和REDO操作**：
  + **UNDO**：用于回滚失败事务的操作，将数据恢复到旧的值。
  + **REDO**：用于重新执行已提交事务的操作，确保所有的修改持久化。

**3. 检查点（Checkpoint）**

为了减少恢复时间，系统会周期性地进行**检查点**操作。检查点将当前所有活跃事务的状态和缓冲区的内容写入磁盘，从而减少在崩溃后需要重新执行的日志数量。

**六、并发控制协议与恢复策略**

**1. 并发控制协议**

* **基于锁的协议**：
  + **简单锁协议**：每次使用数据前申请锁，使用后释放锁。
  + **两阶段锁协议（2PL）**：通过分阶段申请和释放锁来保证可串行化。

**2. 恢复策略**

* **基于日志的恢复**：
  + 在事务失败或系统崩溃时，通过日志来恢复数据库。
  + 通过**Undo**恢复未提交事务，通过**Redo**重做已提交的事务，确保数据库的持久性。

**七、例子总结**

以下是关于并发控制与恢复的一个经典例子——银行转账问题：

* 假设事务T1要从账户A中取出$50并将其存入账户B，而事务T2要从账户A中取出10%的余额并存入账户B。
* 如果两个事务T1和T2并发执行，可能会导致不一致的状态。因此需要对这些操作加锁，并确保按正确的顺序执行，以保持数据库的ACID属性。
* 通过优先图的构建，系统可以检测冲突并确保调度是可串行化的。同时，如果系统崩溃，基于日志的恢复策略会确保事务的原子性和持久性。

**NoSQL**

**一、NoSQL 数据库简介**

* **NoSQL** 是“not only SQL”的缩写，指的是与传统 SQL 不同的数据库管理系统。它们最早于 2009 年提出，主要用于处理**大数据** (Big Data)，包括非结构化和半结构化的数据。
* **大数据的特点**：
  + **Volume（数据量）**：数据量非常大，例如 Facebook 每天产生数百 TB 的用户日志和图片。
  + **Velocity（速度）**：数据生成和插入速度非常高，例如物联网设备、社交媒体数据等。
  + **Variety（多样性）**：数据类型复杂，涵盖文本、图片、音频等多种格式。

**二、RDBMS 和 NoSQL 的比较**

* **RDBMS** 采用**表**和**列**来存储数据，使用结构化查询语言 (SQL) 进行数据操作，提供**ACID** 特性（原子性、一致性、隔离性、持久性），支持事务和数据的一致性。
* **NoSQL** 则不使用关系模型，大多数采用分布式集群架构，具有动态数据模式，可以进行水平扩展，特别适合处理大规模非结构化数据。

在**大数据**背景下，传统的关系型数据库往往面临扩展性瓶颈，因此一些应用场景选择牺牲 ACID 特性来换取更高的扩展性和灵活性，例如使用 NoSQL 来存储和处理这些数据。

**三、CAP 定理**

CAP 定理指出，在分布式系统中，最多只能同时满足以下三个特性中的两个：

1. **一致性 (Consistency)**：所有节点在同一时间的数据状态一致。
2. **可用性 (Availability)**：每个请求都能接收到成功的响应，无论系统内部状态如何。
3. **分区容错性 (Partition Tolerance)**：系统即使在部分节点发生故障或通信失败的情况下仍能继续运作。

对于 NoSQL 数据库，通常会在一致性和可用性之间做出权衡，选择合适的特性组合来适应不同的应用需求。

**四、NoSQL 数据库的类型**

NoSQL 数据库根据数据模型可以分为以下几种类型：

**1. 键值存储 (Key-Value Stores)**

* **数据模型**：类似哈希表，以键值对形式存储数据，键用于唯一标识数据，值为未结构化的数据。
* **特点**：
  + 数据简单，不要求固定的模式。
  + 可以快速进行数据插入、删除和查找，具有很好的扩展性。
* **缺点**：对于复杂的数据查询和连接操作支持不足。
* **代表**：Redis、Amazon DynamoDB。

**2. 文档数据库 (Document Stores)**

* **数据模型**：使用类似 JSON 或 BSON 格式来存储数据。每个文档是自描述的，可以包含嵌套的数组和对象。
* **特点**：
  + 支持复杂的层次结构，数据可以包含嵌套的集合。
  + 可以根据文档中的字段创建索引，提升查询效率。
* **使用场景**：适用于需要灵活数据模式的应用，如内容管理系统。
* **代表**：MongoDB。
* **示例**：文档可以类似于：

{

"\_id": 1,

"name": "John Doe",

"age": 29,

"addresses": [

{ "city": "New York", "zip": "10001" },

{ "city": "San Francisco", "zip": "94105" }

]

}

**3. 列族存储 (Column-Family Stores)**

* **数据模型**：起源于 Google 的 BigTable，将数据按列族进行存储，每个行键对应多个列，且每个列族内的列可以动态增加。
* **特点**：
  + 适合 OLAP（在线分析处理），可以进行高效的数据聚合操作。
  + 数据按列存储，更适合进行大规模的列访问。
* **缺点**：不适用于高并发的 OLTP（在线事务处理）。
* **代表**：Apache Cassandra、HBase。

**4. 图数据库 (Graph Databases)**

* **数据模型**：使用图结构来表示数据，包括节点（实体）和边（关系），每个节点和边都可以有属性。
* **特点**：
  + 适合表示高度互联的数据，特别是社交网络、推荐系统等场景。
  + 查询语言通常是图专用的语言，例如 Neo4j 的 Cypher。
* **使用场景**：社交网络、推荐系统、物联网等需要复杂关系的数据。
* **代表**：Neo4j。
* **示例**：在社交网络中，节点表示用户，边表示“朋友”关系，查询某用户的朋友可以通过图遍历实现。

**五、MongoDB 和 Neo4j 的对比**

* **MongoDB** 主要用于存储具有层次结构的半结构化数据，适合需要灵活数据结构和高扩展性的场景。
* **Neo4j** 主要用于处理复杂关系的数据，适合需要频繁进行关系查询和遍历的场景，例如社交网络中的好友推荐。

**六、选择合适的数据库**

在选择数据库时，需要根据应用的具体需求来决定：

* **结构化数据**：RDBMS 更适合。
* **非结构化或半结构化数据**：NoSQL 更具灵活性，尤其是在高并发、大数据量场景下。
* **高度互联的数据**：图数据库如 Neo4j 是更好的选择。

例如：

* 如果要存储社交网络的好友关系并进行推荐，那么图数据库是最佳选择，因为它能够高效地处理复杂的关系。
* 如果需要快速扩展且不要求复杂事务管理的系统，可以选择键值存储或文档存储，例如用 MongoDB 来管理用户个人信息和偏好。