数据库复习提纲

Kimokcheon, Lucas

2024 年 6 月 30 日

目录

1 绪论

1.1 基础概念

1.1.1 试述数据、数据库、数据库系统、数据库管理系统的概念

1. **数据 (Data)：** 描述事物的符号记录称为数据。
2. **数据库 (DataBase，简称 DB)：** 数据库是长期储存在计算机内的、有组织的、可共享的数据集合。数据库中的数据按一定的数据模型组织、描述和储存，具有较小的冗余度、较高的数据独立性和易扩展性，并可为各种用户共享。
3. **数据库管理系统 (DataBase Management System，简称 DBMS)：** 数据库管理系统是位于用户与操作系统之间的一层数据管理软件，主要功能又：
   (a) 数据定义功能
   (b) 数据组织、存储和管理
   (c) 数据操纵功能
   (d) 数据库的事务管理和运行管理
   (e) 数据库的建立和维护功能
4. **数据库系统 (DataBase System，简称 DBS)：** 数据库系统是指在计算机系统中引入数据库后的系统构成，一般由数据库、数据库管理系统（及其开发工具）、应用系统、数据库管理员构成。

1.1.2 DBMS 的主要功能：

1. 数据库的定义功能：DBMS 提供数据定义语言（DDL）定义数据库的三级结构及其相互之间的映射、完整性、安全控制等约束。
2. 数据库的存储管理：DBMS 的存储管理子系统提供了数据库中数据和应用程序的一个界面，其职责是把各种 DML 语句转换成低层的文件系统命令，起到数据的存储、检索和更新的作用。
3. 数据库的操纵功能：DBMS 提供数据操纵语言（DML）实现对数据库中数据的操作。
4. 数据库的保护功能：DBMS 对数据库的保护主要通过数据库的恢复、数据库的并发控制、数据库的完整性控制、数据库的安全性控制等四个方面实现。
5. 数据库的维护功能：DBMS 中实现数据库维护功能的实用程序主要有数据装载程序、备份程序、文件重组织程序、性能监控程序。

1.1.3 DBMS 的组成：

1. 查询处理器有四个主要成分：DDL 编译器、DML 编译器、嵌入型 DML 的预编译器、查询运行核心程序。
2. 存储管理器有四个主要成分：授权和完整性管理器、事务管理器、文件管理器、缓冲区管理器。

1.1.4 DBS 的组成：

DBS 是一个实际可运行的，按照数据库方法存储、维护和向应用系统提供数据支持的系统，它是数据库、硬件、软件、数据库管理员（DBA）的集合体。

1. 数据库（DB）：是与一个特定组织各项应用有关的全部数据的集合，由应用数据的集合（物理数据库）、关于各级数据结构的描述（描述数据库）两部分组成。
2. 硬件：包括中央处理机、内存、输入输出设备、数据通道等硬件设备。
3. 软件：包括 DBMS、OS、各种宿主语言和应用开发支持软件等程序。
4. 人员：DBA、系统分析师和数据库设计人员、应用程序员和最终用户。

1.1.5 DBS 的全局结构：

1. 数据库用户。可分为四类：DBA、专业用户、应用程序员、最终用户
2. DBMS 的查询处理器。包括四部分：DML 编译器、嵌入型 DML 的预编译器、DLL 编译器、查询运行核心程序。
3. DBMS 的存储管理器。包括四部分：授权和完整性管理器、事务管理器、文件管理器、缓冲区管理器。
4. 磁盘存储器中的数据结构。包括四种形式：数据文件、数据字典、索引文件、统计数据组织。

1.1.6 数据管理的三个阶段

阶段	人工管理	文件系统	数据库系统
数据能否保存	不能保存	可以保存	可以保存
数据面向的对象	某一应用程序	某一应用程序	整个应用系统
数据的共享程度	无共享，一组数据只能对应一个应用程序。	共享性差	共享性高
数据的独立性	不独立，它是应用程序的一部分。	独立性差	数据库与应用系统完全分开

1.1.7 试述文件系统与数据库系统的区别和联系

文件系统与数据库系统的区别是：

- 文件系统面向某一应用程序，共享性差，冗余度大，数据独立性差，记录内有结构，整体无结构，由应用程序自己控制。
- 数据库系统面向现实世界，共享性高，冗余度小，具有较高的物理独立性和一定的逻辑独立性，整体结构化，用数据模型描述，由数据库管理系统提供数据的安全性、完整性、并发控制和恢复能力。

文件系统与数据库系统的联系是：

- 文件系统与数据库系统都是计算机系统中管理数据的软件。解析文件系统是操作系统的重要组成部分
- 而 DBMS 是独立于操作系统的软件。但是 DBMS 是在操作系统的基础上实现的；
- 数据库中数据的组织和存储是通过操作系统中的文件系统来实现的。

1.1.8 数据库系统的特点

- 面向全组织的复杂的数据结构
  - 把文件系统中记录内部有结构的思想扩大到了记录型之间
  - 从整体从发来考虑数据结构问题，而不是仅仅考虑单个应用的数据结构
  - 这种面向整体的数据结构化不但要求描述数据本身，而且还要描述数据之间的联系: 数据的结构化是数据库系统的主要特征，是与文件系统的根本差别
- 数据冗余小、易扩充
  - 从整体的观点来看待和描述数据，数据就不再是面向某个特定应用，而是面向整个系统
  - 对数据库应用的支持也可以有更灵活的方式
- 数据独立性高
  - 数据库系统的重要目标之一：使数据与程序彼此分离
  - 把数据的定义和描述从应用程序中分离出去，数据的存取由 DBMS 管理，用户不必考虑存取路径等细节
  - 包括数据和应用程序物理独立性和逻辑独立性
    - **物理数据独立性**
    - **逻辑数据独立性**
- 统一的数据管理、控制功能
  - 允许多个用户或多个应用程序同时访问数据库中的相同数据
    - 数据的安全性（Security）保护
    - 数据的完整性（Integrity）控制
    - 并发控制（Concurrency）
    - 数据库恢复（Recovery）
  - 数据的最小存取单位是数据项

1.2 数据库系统的结构及功能

1.2.1 数据库系统的三级模式结构分别是怎样的，它如何保证数据独立性？

1. **模式：**是数据库中全体数据的逻辑结构和特征的描述。
   - 模式只涉及数据库的结构；
   - 模式既不涉及应用程序，又不涉及数据库结构的存储；
2. **外模式/子模式：**当模式改变时，由数据库管理员对各个外模式/模式映像做出改变，能使外模式保持不变，这样应用程序不用做出修改，保证了数据与程序的逻辑独立性。
   - 特点：一个应用程序只能使用一个外模式，但同一个外模式可为多个应用程序使用。
3. **内模式/存储模式：**当数据库的存储结构改变时，由数据库管理员对模式/内模式映像做出改变，能使模式保持不变，从而应用程序不用修改，保证了数据与程序的物理独立性。

1.2.2 两级映像

1. 外模式/模式映像：保证数据库的逻辑独立性；
2. 模式/内模式映像：保证数据库的物理独立性；

1.2.3 两级映象的意义

1. 使数据的定义和描述从应用程序中分离出去。
2. 数据的存取完全由 DBMS 管理，大大减少了应用程序的维护和修改。

1.3 两类数据模型

1.3.1 概念模型

按用户的观点来对数据和信息建模，主要用于数据库设计。例如 E-R 模型。

1.3.2 逻辑模型和物理模型

- 逻辑模型: 基于计算机系统视图对数据进行建模。例如关系模型。
- 物理模型: 底层数据的抽象。通常由 DBMS 完成。

2 概念模型—E-R 模型

2.1 关系约束

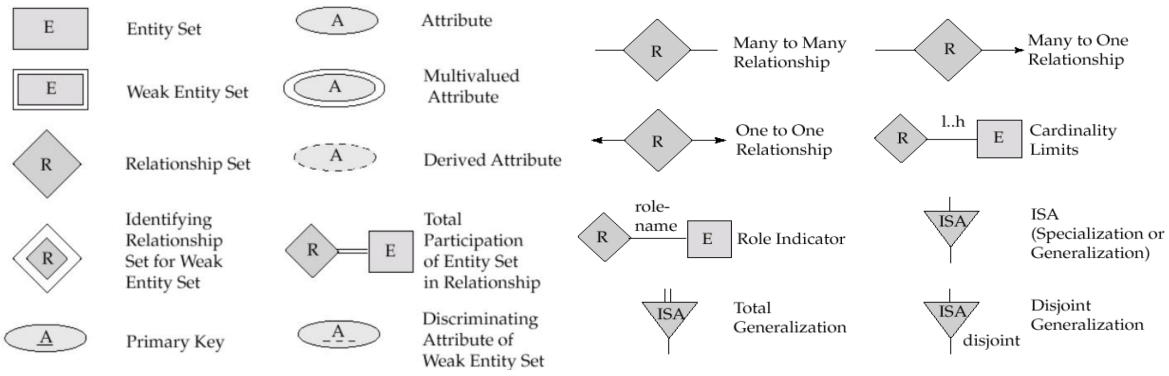
1. **基数约束：**一对一 (1:1)，一对多 (1:N) 和多对多 (N:M)。
2. **参与性约束：**全部参与，部分参与。

2.2 定义并解释概念模型中以下术语：实体，实体型，实体集，实体之间的联系

1. **实体：**客观存在并可以相互区分的事物叫实体。
2. **实体型：**具有相同属性的实体具有相同的特征和性质，用实体名及其属性名集合来抽象和刻画同类实体，称为实体型。
3. **实体集：**同型实体的集合称为实体集；
4. **实体之间的联系：**通常是指不同实体型的实体集之间的联系，实体之间的联系有一对一，一对多和多对多等多种类型。

2.3 作图

1. 矩形框: 实体型
2. 菱形框: 联系
3. 椭圆框: 实体型和联系的属性
4. 直线: 连接实体型和联系的属性，表示联系的种类



2.4 E-R 图转换为关系数据库

1. 1 : 1 联系可以与任意一端对应的关系模式合并。
2. 1 : N 联系可以与 N 端对应的关系模式合并。
3. N : M 联系转换为一个关系模式，与该联系相连的各实体的码以及联系本身的属性均转换为关系的属性，各实体的码组成关系的码或关系码的一部分。

3 逻辑模型—关系模型

3.1 逻辑模型分类

1. 层次模型: 树
2. 网状模型: 有向图
3. 关系模型: 不可分的二维表

3.1.1 试述网状、层次、关系模型的优缺点。

层次模型的优点主要有:

1. 模型简单，对具有一对多层次关系的部门描述非常自然、直观，容易理解，这是层次数据库的突出优点；

2. 用层次模型的应用系统性能好，特别是对于那些实体间联系是固定的且预先定义好的应用，采用层次模型来实现，其性能优于关系模型；

3. 层次数据模型提供了良好的完整性支持。

层次模型的缺点主要有：

1. 现实世界中很多联系是非层次性的，如多对多联系、一个结点具有多个双亲等，层次模型不能自然地表示这类联系，只能通过引入冗余数据或引入虚拟结点来解决；

2. 对插入和删除操作的限制比较多；

3. 查询子女结点必须通过双亲结点。

网状数据模型的优点主要有：

1. 能够更为直接地描述现实世界，如一个结点可以有多个双亲；

2. 具有良好的性能，存取效率较高。

网状数据模型的缺点主要有：

1. 结构比较复杂，而且随着应用环境的扩大，数据库的结构就变得越来越复杂，不利于最终用户掌握；

2. 其 DDL 、DML 语言复杂，用户不容易使用。由于记录之间联系是通过存取路径实现的，应用程序在访问数据时必须选择适当的存取路径。因此，用户必须了解系统结构的细节，加重了编写应用程序的负担。

关系模型具的优点主要有：

1. 关系模型与非关系模型不同，它是建立在严格的数学概念的基础上的。

2. 关系模型的概念单一，无论实体还是实体之间的联系都用关系表示，操作的对象和操作的结果都是关系，所以其数据结构简单、清晰，用户易懂易用。

3. 关系模型的存取路径对用户透明，从而具有更高的数据独立性、更好的安全保密性，也简化了程序员的工作和数据库开发建立的工作。

关系模型具的缺点主要有：

1. 由于存取路径对用户透明，查询效率往往不如非关系数据模型。

2. 为了提高性能，必须对用户的查询请求进行优化，增加了开发数据库管理系统的难度。

3.2 关系模型基础概念

1. 关系: 一个关系对应符合某个关系模式的一张具体的表。

2. 属性：表中的一列即为一个属性；

3. 元组: 表中的一行即为一个元组；

4. 域: 对于关系的每个属性都存在一个允许取值的集合（定义域）。域要有原子性，即域中的每个元素不可再分；

5. 分量: 元组中属性的取值;
6. 关系实例：指代一个关系的特定实例，关系实例包含一组特定的行
7. 关系模式: 数据库关系设计的一个模板。$R(U, D, dom, F)$
8. 关系数据库模式: 一组关系模式的总称，简称模式
9. 候选码：给定关系里可唯一区分不同元组的最小属性组
10. 主码：从候选码集合中选择的某一个码
11. 超码：给定关系里可唯一区分不同元组的属性组
12. 组合码: 不是单一属性的主码
13. 全码: 所有属性是关系的唯一候选码
14. 外码: 其他关系模式的主码。
15. 主属性: 包含在任意候选码中的属性
16. 非主属性: 不包含在任何候选码中的属性

3.3 试述关系模型的构成

关系模型由关系数据结构、关系操作集合和关系完整性约束三部分组成。在用户观点下，关系模型中数据的逻辑结构是一张二维表，它由行和列组成。

3.3.1 关系数据结构:

1. 关系代数
2. 关系演算
3. SQL

3.3.2 关系操作:

1. 一次一集合
2. 查询 & 更新

3.3.3 完整性约束:

1. 插入, 删除, 更新必须满足完整性约束
2. 关系的完整性约束:
   (a) **实体完整性:** 关系的主码中的属性值不能为空值。每个数据表都必须有主键，而作为主键的所有字段，其属性必须是独一且非空值。

(b) **参照完整性：**引用关系中的任意元组在指定属性上出现的取值也必然出现在被引用关系中至少一个元组的指定属性上。且引用的属性是被引用关系的主码。

(c) **用户定义完整性：**反映特定应用对数据的语义需求。

3.4 关系代数

3.4.1 关系代数运算符 (Relational Algebra Operators)

- **选择 (σ)**: 从关系中选择满足给定条件的元组。$\sigma_F(R) = \{t|t \in R \wedge F(t) =' true'\}$。它类似于 SQL 中的 WHERE 子句。例如，$\sigma_{age>25}(Persons)$ 可以选择出年龄大于 25 岁的人。
- **投影 (π)**: 从关系 R 中选择若干属性组成新的关系，并把新关系的重复元组去掉。$\Pi_A(R) = \{t[A]|t \in R\}$。它类似于 SQL 中的 SELECT 子句。例如，$\pi_{name,age}(Persons)$ 可以选择出关系 Persons 中的姓名和年龄属性。
- **连接 ($\bowtie$)**: 将两关系按一定条件连接成一个新关系。
  - θ-连接 ($\bowtie_\theta$)：$R \bowtie_\theta S = \sigma_\theta(R \times S)$
  - 等值连接 ($\bowtie_{A=B}$)：指定两个关系中的属性值相等时才连接
  - 自然连接 ($\bowtie$)：$R \bowtie S = \{t \| t = r \cup s, r \in R, s \in S,$ 并且对于所有在 R 和 S 中共有的属性 A, $r[A] = s[A]\}$
  - 半连接 ($\ltimes$)：给定两个关系 R 和 S，其中 R 和 S 具有一些共同的属性（或列）。R 对 S 的半连接，记作 $R \ltimes S$，是一个新的关系，包含了 R 中所有与 S 中至少一行在共有属性上相匹配的行。
  - 外连接：外连接是对两个关系进行连接操作，同时保留了未匹配的元组，并在新增的属性中填充空值。外连接分为左外连接和右外连接：
    - 左连接 (Left Outer Join): 保留左侧关系中未匹配的元组。
    - 右连接 (Right Outer Join): 保留右侧关系中未匹配的元组。
- **除 ($\div$)**: 给定两个关系 R 和 S，找出 R 中对于 S 中每个元组都有对应的元组的值。
- **集合运算符**:
  - 并 ($\cup$): 包含两个关系中的所有元组，不包括重复元组。类似于 SQL 中的 UNION 操作。
  - 交 ($\cap$): 包含同时存在于两个关系中的元组。类似于 SQL 中的 INTERSECT 操作。
  - 差 ($-$): 包含存在于第一个关系但不存在于第二个关系中的元组。类似于 SQL 中的 EXCEPT 操作。
  - 笛卡尔积 ($\times$): 由两个关系的所有可能的元组对组成的集合。在 SQL 中，可以使用 CROSS JOIN 来执行笛卡尔积操作。
- **逻辑运算符**:
  - 与 ($\wedge$): 连接两个关系代数表达式，返回同时满足两个表达式条件的元组。
  - 或 ($\vee$): 连接两个关系代数表达式，返回满足任一表达式条件的元组。
  - 非 ($\neg$): 对关系中的元组进行取反操作。

4 关系数据库标准查询语言 SQL

4.1 SQL 数据库的体系结构及术语

SQL 数据库的体系结构也是三级，但术语与传统的关系模型不同。模式称为“基本表”，内模式称为“存储文件”，外模式称为“视图”，元组称为“行”，属性称为“列”。

4.2 SQL 数据库的体系结构要点

1. 一个 SQL 数据库是表（table）的汇集，它用一个或多个 SQL 模式定义。一个 SQL 模式是表和授权的表态定义。
2. 一个 SQL 表由行集构成，一行是列的序列，每列对应一个数据项。
3. 一个表或者是一个基本表，或者是一个视图。（视图只保存定义，不保存数据）
4. 一个基本表可跨一个或多个存储文件，一个存储文件也可存放一个或多个基本表。每个存储文件与外部存储器上一个物理文件对应。
5. 用户可用 SQL 语句对视图和基本表进行查询等操作。
6. SQL 用户可以是应用程序，也可以是终端用户。

4.3 SQL 的组成

SQL 主要分成四部分：

1. 数据定义（SQL Data Definition/Description Language）：用于定义 SQL 模式、基本表、视图和索引的创建和撤消操作。
2. 数据操纵（SQL Data Manipulation Language）：数据操纵分成数据查询和数据更新两类。数据更新又分成插入、删除和修改三种操作。
3. 数据控制（SQL Data Control Language）：包括对基本表和视图的授权，完整性规则的描述，事务控制等内容。
4. 嵌入式 SQL 的使用规定：涉及到 SQL 语句嵌入在宿主语言程序中使用的规则。

4.4 SQL(Standard/Structured Query Language) 概述

SQL 是一种用于管理关系型数据库的标准化语言，可以用于数据库的访问、管理和操作。

4.4.1 基本类型

- char(n)：固定长度为 n 的字符串
- varchar(n)：最大长度为 n 的可变长字符串
- int, smallint：整数类型
- numeric(p, d)：p 位，小数点后 d 位的数值类型
- float(n)：至少精度为 n 的浮点数类型

4.4.2 数据操作语句

带有条件的查询:

```
SELECT * FROM students WHERE age > 20;
```

此查询将只返回年龄大于 20 岁的学生的所有数据.

表间连接

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
FROM weather JOIN cities ON weather.city = cities.name;
```

等值连接"weather" 表和 "cities" 表

```
SELECT * FROM weather LEFT OUTER JOIN cities ON weather.city = cities.name;
```

左连接"weather" 表和 "cities" 表

聚合函数

```
SELECT city, count(*), max(temp_lo)
FROM weather
GROUP BY city
HAVING max(temp_lo) < 40;
```

窗口函数 (Window Functions)

```
SELECT depname, empno, salary, rank()
OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;
```

根据 depname 分组并按照 salary 倒序排序。与聚合函数类似，不过窗口函数不会像非窗口聚集函数那样将行分组到单个输出行中。

模式匹配

```
string LIKE pattern
```

pattern 中的下划线 (_) 表示 (匹配) 任意单个字符; 百分号 (%) 匹配零或多个字符的任意序列。

```
string SIMILAR TO pattern
```

pattern 为正则表达式（不同处：下划线 (_) 表示 (匹配) 任意单个字符; 百分号 (%) 匹配零或多个字符的任意序列; 没有. 这个符号）

插入数据：

```
INSERT INTO students (student_name, age) VALUES ('John', 25);
```

在”students” 表中插入了一个名为”John” 且年龄为 25 的新学生的数据.

更新数据：

```
UPDATE students SET age = 26 WHERE student_name = 'John';
```

这将更新”students” 表中名为”John” 的学生的年龄为 26 岁.

删除数据：

```
DELETE FROM students WHERE student_name = 'John';
```

将从”students” 表中删除名为”John” 的学生的数据.

4.4.3 数据定义语句

创建表：

```
CREATE TABLE students (
    student_id INT,
    student_name VARCHAR(50),
    age INT,
    dept_name, VARCHAR(50),
    PRIMARY KEY(student_id),
    FOREIGN KEY(dept_name) references department
);
```

这条语句创建了一个名为”students” 的表，包含了”student_id”,”student_name”,”age” 和”dept_name” 四列，并规定了主键和外键。

修改表：

```
ALTER TABLE students ADD email VARCHAR(100);
```

这将在”students” 表中添加一个名为”email” 的新列.

```
ALTER TABLE students DROP email;
```

这将在”students” 表中删除属性”email”.

删除表：

```
DROP TABLE students;
```

这将删除名为”students” 的表及其所有数据.

创建视图

```
CREATE VIEW myview AS
SELECT name, temp_lo, temp_hi, prcp, date, location
FROM weather, cities
WHERE city = name;
```

创建物化视图

```
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;
```

物化视图和视图一样使用规则系统，但会以类似 table 的形式持久化结果。

表的继承

```
CREATE TABLE cities (
    name text,
    population real,
    elevation int
);
CREATE TABLE capitals (
    state char(2) UNIQUE NOT NULL
)   INHERITS (cities);
```

4.5 事务

创建事务 (Transaction)

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
WHERE name = 'Wally';
COMMIT;
```

4.5.1 数据控制语句

授予用户权限：

```
GRANT SELECT, INSERT ON students TO user1;
```

这将允许”user1” 用户在”students” 表上执行 SELECT 和 INSERT 操作.

收回用户权限：

```
REVOKE INSERT ON students FROM user1;
```

这将从”user1” 用户中收回对”students” 表的 INSERT 权限.

4.6 基本表和视图

基本表是本身独立存在的表，在 SQL 中一个关系就对应一个表。视图是从一个或几个基本表导出的表。视图本身不独立存储在数据库中，是一个虚表。即数据库中只存放视图的定义而不存放视图对应的数据，这些数据仍存放在导出视图的基本表中。视图在概念上与基本表等同，用户可以如同基本表那样使用视图，可以在视图上再定义视图。

4.6.1 视图的优点

1. 视图能够简化用户的操作。
2. 视图使用户能以多种角度看待同一数据。
3. 视图对重构数据库提供了一定程度的逻辑独立性。

4. 视图能够对机密数据提供安全保护。

4.6.2 SQL 中视图被称为“虚表”的原因

在 SQL 中创建一个视图时，系统只是将视图的定义存放在数据字典中，并不存储视图对应的数据。当用户使用视图时才去求对应的数据，因此，我们将视图称为“虚表”。这样处理的目的是为了节约存储空间，因为视图对应的数据都可从相应的基本表中获得。

4.6.3 关于视图的更新

并不是所有的视图都是可更新的，因为视图是不实际存储数据的虚表，对视图的更新最终要转换为对基本表的更新。有些视图的更新不能唯一有意义地转换成对相应基本表的更新，所以，并不是所有的视图都是可更新的。

4.6.4 可更新的和不可更新的视图

可更新的视图一般是基本表的行列子集视图。不可更新的视图的属性来自集合函数、表达式。更新视图是指通过视图来插入 (INSERT)、删除 (DELETE) 和修改 (UPDATE) 数据。由于视图是不实际存储数据的虚表，对视图的更新最终要转换为对基本表的更新。为防止用户通过视图对数据进行增加、删除、修改，有意无意地对不属于视图范围的基本表数据进行操作，所以一些相关措施使得不是所有的视图都可以更新。在 SQL 中，允许更新的视图在定义时，必须加上 WITH CHECK OPTION 子句，这样在视图上增删改数据时，DBMS 会检查视图定义中的条件，若不满足条件，则拒绝执行该操作。

4.6.5 物化视图（Materialized View）

物化视图是数据库管理系统中一种特殊的数据库对象，它的内容是通过查询语句从一个或多个基础表（Base Tables）中获取的，并将查询结果以实际数据的形式存储起来。物化视图与普通视图（View）的主要区别在于，普通视图是虚拟的，不存储数据，每次访问时都需要重新计算；而物化视图则存储了查询的结果，不需要每次访问时重新执行查询。

5 关系数据库设计理论

5.1 函数依赖 $(\alpha \to \beta)$

如果在一个给定的关系模式中，对于两组属性 α 和 β，每次 α 的值出现时 β 的值都是固定的，那么就说 β 函数依赖于 α，记作 $\alpha \to \beta$。其中，如果 $\alpha \to \beta$ 且 β 不是 α 的子集，则是非平凡函数依赖，否则是平凡函数依赖。我们一般讨论的都是非平凡函数依赖。

1. 完全函数依赖 $(\alpha \xrightarrow{F} \beta)$：$\alpha \to \beta$ 且对于任意 α 的真子集 α'，α' 不能决定 β。
2. 部分函数依赖 $(\alpha \xrightarrow{P} \beta)$：$\alpha \to \beta$ 且存在 α 的真子集 α'，α' 能决定 β。
3. 传递函数依赖 $(\alpha \xrightarrow{T} \beta)$：$\alpha \to \beta$ 且 $\beta \to \gamma$，但 β 不决定 α, 则 γ 传递函数依赖于 α。

5.2 多值依赖 ($\alpha \to\to \beta$)

多值依赖描述了关系模式的属性之间的关系。对于关系模式 $R(U)$，若属性集合 $\alpha\beta\gamma$ 是三个不相交的子集，且 γ 是 $U-\alpha-\beta$，则如果对于任一关系 r，对 α 给定值，存在一组 β 的值与之对应，且这些 β 的值与 γ 无关，那么就称 β 多值依赖于 α。

5.3 连接依赖

连接依赖描述了关系模式中属性之间的连接关系。对于关系模式 $R(U)$，如果 U 的若干子集 α_i 的并集是 U，那么称关系具有连接依赖，记作 $*(\alpha_i)$.

5.4 异常

以下的学生课程关系的函数依赖为 $\{Sno, Cname\} \to \{Sname, Sdept, Mname, Grade\}$，键码为 $\{Sno, Cname\}$。也就是说，确定学生和课程之后，就能确定其它信息。

Sno	Sname	Sdept	Mname	Cname	Grade
1	学生-1	学院-1	院长-1	课程-1	90
2	学生-2	学院-2	院长-2	课程-2	80
2	学生-2	学院-2	院长-2	课程-1	100
3	学生-3	学院-2	院长-2	课程-2	95

不符合范式的关系，会产生很多异常，主要有以下四种异常：

- 冗余数据：例如学生-2 出现了两次。
- 修改异常：修改了一个记录中的信息，但是另一个记录中相同的信息却没有被修改。
- 删除异常：删除一个信息，那么也会丢失其它信息。例如删除了课程-1 需要删除第一行和第三行，那么学生-1 的信息就会丢失。
- 插入异常：例如想要插入一个学生的信息，如果这个学生还没选课，那么就无法插入。

5.5 范式

范式理论是为了解决以上提到四种异常。高级别范式的依赖于低级别的范式。
核心思想：将一个大的关系分解为几个小的关系满足:key → 非主属性。

5.5.1 第一范式 (1NF)

关系模式的所有属性都是原子的，不可再分。

5.5.2 第二范式 (2NF)

假设 R 属于 1NF，对于 R 中的每一个非平凡函数依赖 $\alpha \to X$，X 是主属性或 α 不是任何键的真子集。我们说 R 属于 2NF。

- 2NF 消除了非主属性的部分依赖。

Example:

Sno	Sname	Sdept	Mname	Cname	Grade
1	学生-1	学院-1	院长-1	课程-1	90
2	学生-2	学院-2	院长-2	课程-2	80
2	学生-2	学院-2	院长-2	课程-1	100
3	学生-3	学院-2	院长-2	课程-2	95

以上学生课程关系中，$\{Sno, Cname\}$ 为键，有如下函数依赖：

- $Sno \to Sname, Sdept$
- $Sdept \to Mname$
- $Sno, Cname \to Grade$

Grade 完全函数依赖于键，它没有任何冗余数据，每个学生的每门课都有特定的成绩。

Sname, Sdept 和 Mname 都部分依赖于键，当一个学生选修了多门课时，这些数据就会出现多次，造成大量冗余数据。

关系-1

Sno	Sname	Sdept	Mname
1	学生-1	学院-1	院长-1
2	学生-2	学院-2	院长-2
3	学生-3	学院-2	院长-2

有以下函数依赖：

- $Sno \to Sname, Sdept$
- $Sdept \to Mname$

关系-2

Sno	Cname	Grade
1	课程-1	90
2	课程-2	80
2	课程-1	100
3	课程-2	95

有以下函数依赖：

- $Sno, Cname \to Grade$

5.5.3 第三范式 (3NF)

假设 R 属于 1NF，对于 R 中的每一个非平凡函数依赖 $\alpha \to X$，X 是主属性或 α 包含一个键。我们说 R 属于 3NF。

- 3NF 基于 2NF 消除了非主属性对键的传递依赖。

Example: 上面的关系-1 中存在以下传递函数依赖：

- $Sno \to Sdept \to Mname$

可以进行以下分解：

关系-11

Sno	Sname	Sdept
1	学生-1	学院-1
2	学生-2	学院-2
3	学生-3	学院-2

关系-12

Sdept	Mname
学院-1	院长-1
学院-2	院长-2

5.5.4 BCNF（Boyce-Codd 范式）

假设 R 属于 1NF，对于 R 中的每一个非平凡函数依赖 $\alpha \to X$，α 包含一个键。我们说 R 属于 BCNF。

- BCNF 完全消除了由函数依赖引起的过度冗余及相应问题。

5.5.5 4NF（第四范式）

假设 R 属于 4NF，对于 D 中的所有多值依赖形式 $\alpha \to\to \beta$，$\alpha \to\to \beta$ 是一个平凡的多值依赖或 α 包含一个键。

- 4NF 消除了多值依赖。

5.6 数据依赖理论

5.6.1 阿姆斯特朗公理

1. 自反律：如果 α 是一组属性且 $\beta \subseteq \alpha$，则 $\alpha \to \beta$ 成立。
2. 增广率：如果 $\alpha \to \beta$ 成立，且 γ 是一组属性，那么 $\gamma\alpha \to \gamma\beta$ 成立。
3. 传递率：如果 $\alpha \to \beta$ 成立且 $\beta \to \gamma$ 成立，那么 $\alpha \to \gamma$ 成立。

阿姆斯特朗公理的推论：

1. 合并规则：如果 $\alpha \to \beta$ 成立且 $\alpha \to r$ 成立，那么 $\alpha \to \beta r$ 成立。
2. 分解规则：如果 $\alpha \to \beta\gamma$ 成立，那么 $\alpha \to \beta$ 和 $\alpha \to \gamma$ 成立。
3. 伪传递规则：如果 $\alpha \to \beta$ 成立且 $\gamma\beta \to \delta$ 成立，那么 $\alpha\gamma \to \delta$ 成立。
4. $\alpha \to \beta_1\beta_2...\beta_K$ 成立，当且仅当 $\alpha \to \beta_i (i = 1, 2, \ldots, k)$ 成立。

5.6.2 函数依赖集合的闭包

F 的闭包，记为 F^+，是由 F 逻辑推导出的所有函数依赖的集合。

1. α_F^+：在一组函数依赖 F 下，由 α 唯一决定的所有属性集合，即 α 在 F 下的闭包。
2. 如果 $\alpha_F^+ = U$，且对于 $\alpha' \subset \alpha$，$\alpha_F^+ \neq U$，则 α 是候选键。
3. 计算候选键：
   (a) 找到属性集 α，使得 α 不出现在所有非平凡函数依赖的右侧。
   (b) 计算 α_F^+。如果 $\alpha_F^+ = U$，停止。否则，继续第 3 步。
   (c) 将 α 扩展到 α'，并计算 α'^+_F。如果 $\alpha'^+_F = U$，则 α' 是候选键。找出所有的 α'。

5.6.3 最小依赖集

1. 每组函数依赖都等同于一个最小依赖集 F_m。
2. 计算最小依赖集 F_m：
   (a) 分解函数依赖: 将 F 中的每个函数依赖分解，使得每个依赖的右侧只有一个属性。
   (b) 消除冗余的函数依赖: 检查 F 中是否有冗余的函数依赖。具体来说，要检查函数依赖 $\alpha \to A$ 是否多余，考虑集合 $G = F - \{\alpha \to A\}$。如果在集合 G 下，闭包 α_G^+ 包括 A，则 G 暗含函数依赖 $\alpha \to A$ ，故在 F 中是冗余的。
   (c) 消除左侧的多余属性: 对于 F 中的每个函数依赖，检查左侧是否有冗余的属性。具体来说，对于像 $\alpha A \to X$ 这样的依赖，确定 A 是否必要。如果在 F 下，闭包 α_F^+ 包括 X，则在函数依赖 $\alpha A \to X$ 中，A 是冗余的。

5.7 模式分解

1. 无损分解：通过用两个关系模式 $r_1(U_1)$ 和 $r_2(U_2)$ 替换 $R(U)$ 而不丢失信息。
   - 无损分解测试
   - 一个模式可以无损地被分解为符合第四范式（4NF）的模式。
2. 依赖保持分解：判断 $F \subseteq (F_1 \cup F_2 \cup \ldots \cup F_k)^+$ 是否成立
   - 为了保持依赖性，一个模式可以被分解为符合第三范式（3NF）的模式，但可能不符合 BCNF。
3. 无损 & 依赖保持的 3NF 分解方法：
   (a) 求最小依赖集 F_m 和候选码。
   (b) 找出不在 F_m 出现的属性，并构成 $R_0 < U_0, F_0 >$，把这些属性从 U 中去掉。
   (c) 若有 $\alpha \to A \in F_m$，且 $\alpha A = U$，则 $\rho = \{R\}$，算法结束。
   (d) 否则，对 F 按具有相同左部的原则分组，每一组函数依赖所涉及的全部属性形成了一个属性组 U_i。若 $U_i \subseteq U_j (i \neq j)$ 就去掉 U_i。

(e) 任选一候选码 α，得到 $R^* < \alpha, F_\alpha >$。

(f) 令 $\rho = \{R_1 < U_1, F_1 >, \cdots, R_k < U_k, F_k >\} \bigcup R_0 < U_0, F_0 > \bigcup R^* < \alpha, F_\alpha >$。若有某个 U_i，$\alpha \subseteq U_i$，将 $R^* < \alpha, F_\alpha >$ 从 ρ 中去掉，或者 $U_i \subseteq \alpha$，将 $R < U_i, F_i >$ 从 ρ 中去掉。

(g) 标出主键和外键。

4. 无损 4NF 分解方法：

(a) 令 $\rho = \{R < U, F >\}$

(b) 若 ρ 中某项不属于 BCNF，依据其 F_i 中违反 BCNF 的 $\alpha \to \beta, \alpha \cap \beta = \emptyset$(非平凡且 α 中没有码)，拆分为 $\alpha\beta$ 和 $U - \beta$。循环。

(c) 如 ρ 中某项不属于 $4NF$，依据其 D_i 中违反 $4NF$ 的 $\alpha \to\to \beta$ 拆分 $\rho = \{R_1 < \alpha\beta, F_1 >, R_2 < \alpha\gamma, F_2 >\}$。循环。

(d) 标出主键和外键。

6 数据库设计

6.1 数据库设计的基本步骤

1. 需求分析
2. 概念结构设计：构建概念模型 (E-R 模型)
3. 逻辑结构设计：将概念模型转换为某个 DBMS 支持的数据模型，生成一组关系模式
4. 物理结构设计：设计具体 DBMS 下的表、视图、索引
5. 数据库实施
6. 数据库运行和维护

6.2 一个好的模式设计方法应符合哪三条原则：

1. 表达性：表达性涉及到两个数据库模式的等价性问题，即数据等价和依赖等价，分别用无损联接和保持函数依赖性来衡量。
2. 分离性：分离性是指属性间的“独立联系”应该用不同的关系模式表达。
3. 最小冗余性：最小冗余性要求在分解后的数据库能表达原来数据库的所有信息这个前提下实现。

7 查询优化

7.1 查询树启发式优化

1. 将并联选择操作分解成一系列单独的选择
2. 尽早进行选择

3. 尽早进行投影
4. 同时执行多个选择和投影

8 数据库的保护

8.1 数据库的安全性

数据库的安全性是指保护数据库免受不合法使用所造成的数据泄露、更改或破坏。

8.1.1 数据库安全性和计算机系统的关系

安全性问题不仅仅存在于数据库系统中，所有计算机系统都面临着这个问题。然而，在数据库系统中，大量数据集中存放，并为许多最终用户直接共享，从而使安全性问题更加突出。系统安全保护措施的有效性是数据库系统的主要指标之一。数据库的安全性和计算机系统的安全性，包括操作系统和网络系统的安全性是紧密联系、相互支持的。

8.1.2 实现数据库安全性控制的常用方法和技术

实现数据库安全性控制的常用方法和技术包括：

1. 用户标识和鉴别：系统提供一定的方式让用户标识自己的名字或身份，并在用户请求进入系统时核对其身份，通过鉴定后才提供系统的使用权。
2. 存取控制：通过用户权限定义和合法权检查，确保只有合法权限的用户访问数据库，而未被授权的人员无法存取数据。例如，自主存取控制（DAC）和强制存取控制（MAC）。
3. 视图机制：为不同的用户定义视图，通过视图机制把要保密的数据对无权存取的用户隐藏起来，从而自动地对数据提供一定程度的安全保护。
4. 审计：建立审计日志，自动记录用户对数据库的所有操作，并对数据库管理员（DBA）提供审计跟踪信息，帮助找出非法存取数据的人、时间和内容等。
5. 数据加密：对存储和传输的数据进行加密处理，使得未知解密算法的人无法获知数据的内容。

8.1.3 数据库中的自主存取控制方法和强制存取控制方法

自主存取控制方法是指定义各个用户对不同数据对象的存取权限，防止不合法用户对数据库的存取。强制存取控制方法是指每个数据对象被标记为一定的密级，并且每个用户被授予某一级别的许可证，系统规定只有具有某一许可证级别的用户才能存取某一密级的数据对象。

8.2 数据库的完整性

8.2.1 什么是数据库的完整性？

数据库的完整性是指数据的正确性和相容性。

8.2.2 数据库的完整性概念与数据库的安全性概念有什么区别和联系？

数据的完整性和安全性是两个不同的概念，但是有一定的联系。前者是为了防止数据库中存在不符合语义的数据，防止错误信息的输入和输出，即所谓垃圾进垃圾出（Garbage In, Garbage Out）所造成的无效操作和错误结果。后者是保护数据库防止恶意的破坏和非法的存取。也就是说，安全性措施的防范对象是非法用户和非法操作，完整性措施的防范对象是不合语义的数据。

8.2.3 什么是数据库的完整性约束条件？可分为哪几类？

完整性约束条件是指数据库中的数据应该满足的语义约束条件。一般可以分为六类：静态列级约束、静态元组约束、静态关系约束、动态列级约束、动态元组约束、动态关系约束。

8.2.4 DBMS 的完整性控制机制应具有哪些功能？

DBMS 的完整性控制机制应具有三个方面的功能：

1. 定义功能，即提供定义完整性约束条件的机制；
2. 检查功能，即检查用户发出的操作请求是否违背了完整性约束条件；
3. 违约反应：如果发现用户的操作请求使数据违背了完整性约束条件，则采取一定的动作来保证数据的完整性。

8.2.5 RDBMS 在实现参照完整性时需要考虑哪些方面？

RDBMS 在实现参照完整性时需要考虑以下几个方面：

1. 外码是否可以接受空值；
2. 在删除被参照关系的元组时的考虑，包括级联删除（CASCADES），受限删除（RESTRICTED），置空值删除（NULLIFIES）；
3. 在参照关系中插入元组时的问题，包括受限插入和递归插入；
4. 修改关系中主码的问题，一般是不能用 UPDATE 语句修改关系主码的。如果需要修改主码值，只能先删除该元组，然后再把具有新主码值的元组插入到关系中。

8.2.6 实现机制

数据库完整性约束条件作用对象粒度可分为列级、元组级和关系级；其状态可分为静态和动态。

8.2.7 完整性检查

控制机制的功能包括定义功能、检查功能和执行动作。检查功能可分为立即执行约束和延迟执行约束。

- 原子性 A
  - 操作不可分割：事务中的操作要么全部执行成功，要么全部不执行，不存在部分执行的情况。
  - 由 DBMS 事务管理子系统实现：数据库管理系统（DBMS）负责确保事务的原子性。
- 一致性 C

  - 保证数据库正常一致：事务执行后，数据库应保持一致性，即满足所有的完整性约束和业务规则。
  - 由程序员和系统完整性约束检查实现：程序员定义的完整性约束以及系统内置的完整性检查机制负责维护数据库一致性。

- 隔离性 I
  - 多事务并发时，应保证互相不干扰：并发执行的多个事务之间应该是相互隔离的，互不干扰。
  - 由 DBMS 并发控制子系统实现：数据库管理系统的并发控制子系统负责确保事务隔离性。

- 持久性 D
  - 事务完成提交后，更新应永久反映在数据库中，不丢失：一旦事务成功提交，其所做的修改应该永久保存在数据库中，不会因系统故障而丢失。
  - 由数据库恢复管理子系统实现：数据库的恢复管理子系统负责保证事务的持久性。

8.3 数据库的恢复

8.3.1 故障种类

事务故障（undo）

- 可能是由事务本身发起的，如事务自己 Abort。也可能是非预期的，如运算溢出、并发事务死锁而被撤销、违反了完整性约束而被终止。

系统故障（undo+redo）

- 系统故障是指造成系统停止运转的任何事件，导致系统要重新启动。如 CPU 故障、操作系统故障、DBMS 代码错误、发生断电。

介质故障（备份 +redo）

- 是指外存故障，如磁盘损坏、磁头碰撞、瞬时强磁场干扰等。

8.3.2 数据转储

1. 静态转储：在系统无运行事务时进行的转储操作。
2. 动态转储：转储期间允许对数据库进行存取或修改。即转储和用户事务可以并发执行。
3. 海量转储：每次转储全部数据库。
4. 增量转储：每次只转储上一次转储后更新过的数据。

8.3.3 数据库系统如何进行事务故障的恢复？

数据库系统在面对事务故障时，需要执行以下步骤进行恢复：

1. 反向扫描日志文件，查找该事务的更新操作。
2. 对该事务的更新操作进行逆操作。
3. 继续反向扫描日志文件，查找该事务的其他更新操作，并做同样的处理。
4. 直到读到该事务的开始标记，事务故障恢复完成。

8.3.4 数据库系统如何进行系统故障的恢复？

1. 正向扫描日志文件，找出在故障发生前已经提交的事务，标记进入 REDO 队列。同时找出故障发生时未完成的事务，标记进入 UNDO 队列。
2. 对 REDO 队列中的各个事务进行重做操作。
3. 对 UNDO 队列中的各个事务进行撤销操作。

8.3.5 数据库系统如何进行介质故障的恢复？

当数据库系统遭遇介质故障时，需要执行以下步骤进行恢复：

1. 装入最新的数据库后备副本，并让数据库恢复到最近一次转储时的一致性状态。
2. 再装入相应的日志文件副本，重做已完成的事务。

8.3.6 检查点技术

数据库系统利用检查点技术来动态维护日志文件，以实现周期性保存数据库状态的操作：

1. 将当前日志缓冲区中所有的日志记录写入磁盘的日志文件上。
2. 在日志文件中插入一个检查点记录。
3. 将当前数据缓冲区的所有数据记录写入磁盘的数据库中。
4. 把检查点在日志文件中的地址写入重新开始文件。

8.3.7 使用检查点进行恢复的步骤是：

数据库系统在使用检查点进行恢复时，需要经历以下步骤：

1. 从重新开始文件中找到最后一个检查点记录在日志文件中的地址，由该地址在日志文件中找到最后一个检查点记录。
2. 由该检查点记录得到检查点建立时刻所有正在执行的事务清单，建立两个事务队列：UNDO-LIST 和 REDO-LIST，首先将事务清单放入 UNDO-LIST 队列。
3. 如有新开始的事务，就暂时放入 UNDO 队列。
4. 如果有提交的事务，就从 UNDO 队列移到 REDO 队列；直到日志文件结束。
5. 对 REDO-LIST 中的事务执行 REDO 操作，对 UNDO-LIST 中的事务执行 UNDO 操作。

9 并发控制

9.1 三大问题

- 修改丢失问题：指在多个事务同时尝试修改同一数据时，一个事务的修改可能会被另一个事务的修改覆盖，导致数据丢失。

  - 具体表现为写后写：即一个事务覆盖了另一个事务已经提交的数据，导致前一个事务的修改丢失。

- 脏读问题：指一个事务读取了另一个事务未提交的数据，当另一个事务回滚时，读取到的数据就是无效的，即“脏数据”。
  - 具体表现为写后读回退：即一个事务读取了另一个事务未提交的数据，而后者回滚了导致读取的数据无效。

- 不可重复读问题：指在一个事务中，多次读取同一数据可能会得到不同的结果，这是由于其他事务对该数据进行了修改所导致的。
  - 具体表现为执行代码写后读不一致：即一个事务在多次读取同一数据时，由于其他事务对该数据进行了修改，导致读取结果不一致。
  - 幻影读：指在一个事务中，同一查询可能会返回不同数量的数据行，这是由于其他事务对数据进行了插入或删除操作所导致的。

9.2 数据库封锁类型

- 排他锁 X：用于在数据修改时加锁，防止其他事务读取或修改相同数据。
  - 具体表现为在修改数据时加上排他锁 X，阻止其他事务同时修改相同数据。

- 共享锁 S：用于在数据查询时加锁，允许其他事务同时读取相同数据但不允许修改。
  - 具体表现为在查询数据时加上共享锁 S，允许其他事务同时读取相同数据，但不允许修改。

- 意向锁 I: 属于隐式锁。如果一个节点以意向模式被锁定，则在该树的更低级别进行显式锁定。
  - 在显式锁定一个节点之前，将意向锁放在所有祖先节点上。
  - 意向共享锁（IS）、意向排他锁（IX）、共享和意向排他锁（S IX）。

9.3 封锁协议

1. 第一级别：在更新数据 R 之前，事务 T 需要获得 R 上的 X 锁，直到事务结束。事务的结束包括正常的 COMMIT 和异常的 ROLLBACK。(避免修改丢失问题)
2. 第二级别：除了第一级别的锁定协议外，事务 T 在读取数据 R 之前必须获得 S 锁，直到读取结束。(避免脏读问题)
3. 第三级别：除了第一级别的锁定协议外，事务 T 在读取数据 R 之前必须获得 S 锁，直到事务结束。(避免不可重复读问题)

9.4 两种需要处理的危险

- 活锁：指一个事务由于等待其他事务释放资源而无法继续执行，导致一直处于等待状态。
  - 具体表现为一个事务永远等待，需要先申请先服务解决。

- 死锁：指两个或多个事务互相持有对方需要的资源，并且互相等待对方释放资源，导致它们都无法继续执行。
  - 具体表现为两个事务占用互相需要的资源互相等待。

9.4.1 数据库死锁及解决方法

数据库死锁是指在多个事务并发执行过程中，彼此互相请求锁资源，导致所有事务都无法继续执行的状态。数据库有两种策略来解决死锁问题：

1. 预防死锁
   - 一次封锁法：即一次性全加锁来避免死锁的发生。
   - 顺序封锁法：尽量按相同的顺序访问资源，减少死锁的发生概率。
2. 诊断与解除死锁
   - 超时法：当一个事务等待的时间超过设定的超时阈值时，系统会自动回滚这个事务，从而破坏死锁环路。
   - 等待图法：通过构建等待图来识别死锁，并选择撤销一个或多个事务来解除死锁。

DBMS 会选择一个处理死锁代价最小的事务，将其撤销，释放此事务持有的所有的锁。

9.5 事务隔离级别

1. 未提交读（read uncommitted）
2. 提交读（read committed）
3. 可重复读（repeatable read）
4. 可串行化（serializable）

9.5.1 可串行化

可串行化是并发事务正确调度的准则。

1. 定义：多个事务的并发执行是正确的，当且仅当其结果与按某一次序串行地执行这些事务时的结果相同。
2. 如果一个调度 S 与一个串行调度在冲突上等价，则称为冲突可串行化。冲突可串行化调度是可串行化调度的充分条件。
3. 冲突指令：R&W 或 W&W。
4. 冲突等价：如果通过一系列非冲突指令的交换，可以将调度 S 转换为串行调度 S'。

9.5.2 简述两段锁协议的内容

并发控制的主要方法。两阶段锁定协议确保了冲突可串行化，但不保证不发生死锁。
所有事务必须分两个阶段对数据项加锁和解：

1. 增长阶段：事务可以获得锁，但不能释放任何锁。对任何数据进行读、写操作之前，首先要申请并获得对该数据的封锁；
2. 收缩阶段：事务可以释放锁，但不能获得任何新的锁。在释放一个锁后，事务不再申请和获得任何其他锁。