CS3322 数据库原理

tauyoung

目录

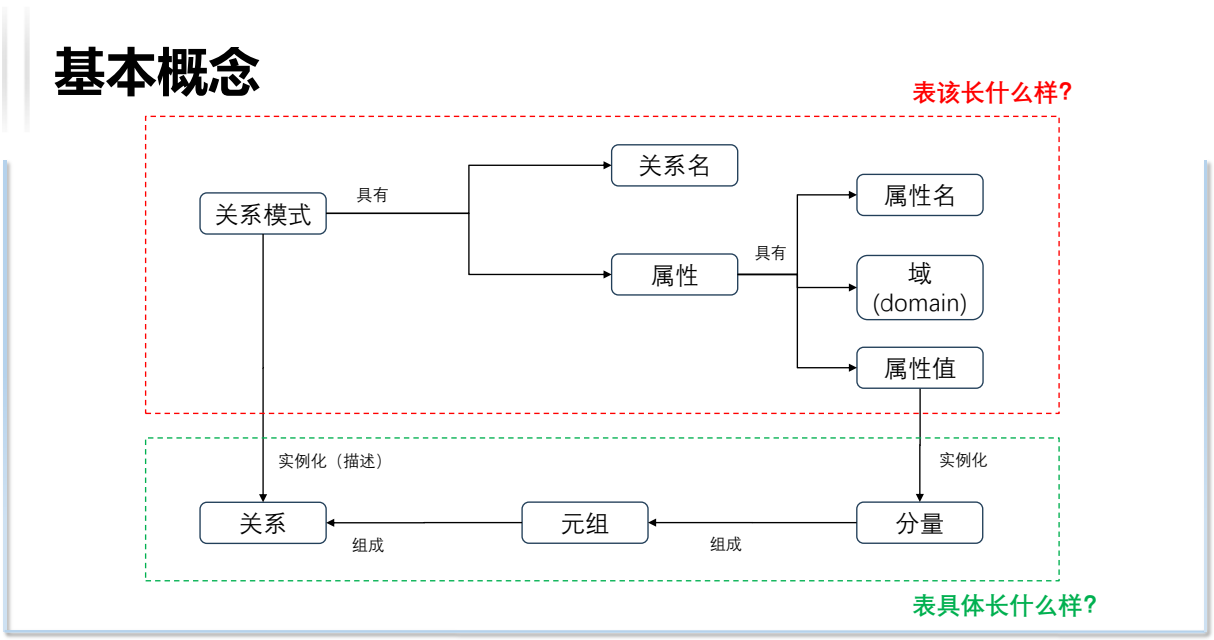
第一章　关系代数

关系代数是一阶逻辑的分支，是闭合于运算下的关系的集合。运算作用于一个或多个关系上来生成一个关系。

关系代数是关系空间和关系空间算子的运算。

1.1　关系空间

关系模型术语		一般表格的术语
关系模式	relation schema	表头
关系名	relation name	表名
关系	relation	二维表
元组	tuple	行/记录
属性	attribute name	列名
属性值	attribute value	列值
分量	component	一条记录中的一个列值



1.1.1　关系模式

关系模式是对关系的描述，由关系名和其属性集合组成，是相对不变的，记做 $R(A_1, A_2, \dots, A_n)$。

例

关系模式 1 = 火车票记录（乘客 ID，车次，车票号，出发时间，到达时间，出发地点，到达地点）

1.1.2　属性

域是一组具有相同数据类型的值的集合。一个属性由属性名和属性域来定义。一个属性在其域中的具体取值为属性值。

1.1.3　元组、分量和关系

给定关系模式 $R(A_1, A_2, \dots, A_n)$，元组是各个属性在其域中的一次取值，记为：

$$t = (a_1, a_2, \dots, a_n), \quad a_i \in \mathrm{Dom}(A_i)$$

元组中的每个元素成 a_i 为一个分量。关系是元组的集合。

1.1.4　超键、候选键、主键和外键

- 一个属性组称为一个键。
- 若关系中的某一属性组的值能唯一地标识一个元组，则称该属性组为超键 (super key)。
- 候选键 (candidate key) 是不含有多余属性的超键（一个超键去除任意一个属性就不再是超键）。
- 一个关系可能有一个或多个候选键，在定义关系数据库关系表的时候，一般选择一个最合适的候选键作为主键 (primary key)，主键的不同取值必须唯一标识不同的元组。
- 一个外键是一个关系（称为派生关系或子关系）中的一个属性或属性组合，其值必须匹配另一个关系（称为主关系或父关系）中主键的值。

1.2　关系完整性约束

1.2.1　实体完整性约束 (Entity Integrity Constraint)

- 如果键 K 是关系 R 的主键，则 K 不能取空值。
- 如果关系 R 的主键 K 是复合键，则构成复合键的多个属性均不能取空值。
- 如果键 K 是关系 R 的主键，则 K 的取值不能在 R 中重复。

1.2.2　参照完整性约束 (Referential Integrity Constraint)

- 删除规则：如果一个实体 E 被另一个实体 F 引用，可以禁止删除该实体 E。
- 插入规则：如果一个实体 F 引用了另一个实体 E，那么实体 E 必须存在数据库中。

1.2.3　用户定义完整性约束 (User-defined Integrity Constraint)

用户定义完整性约束则是用户根据具体的数据库应用场景，设置的具体的约束条件，用户定义完整性约束可以反映数据的特殊语义要求。

1.3　关系运算

1.3.1　基本关系代数

选择

选择运算 (σ) 可以从关系 R 中获取满足条件的元组：

$$\sigma_p(R) = \{t \mid t \in R \land p(t) = \text{True}\}$$

其中 p 是*选择谓词*，是由逻辑运算符与 ($\land$)、或 ($\lor$)、非 ($\neg$) 连接的若干*原子表达式*构成的公式。
原子表达式的形式为 $X\ \theta\ Y$，其中 X, Y 表示属性名、常量或者函数值，θ 是比较运算符，包括 $=$、$>$、$<$、$\geqslant$、$\leqslant$、$\neq$ 等。

投影

投影运算 (Π) 可以从关系 R 中获取某些列组成新的关系：

$$\Pi_{A_1,A_2,\ldots,A_n}(R) = \{t[A_1, A_2, \ldots, A_k] \mid t \in R\}$$

其中 $A_1, A_2, \ldots, A_n$ 为 R 的属性列，将返回 R 中元组在 $A_1, A_2, \ldots, A_k$ 列上的值并删除重复元组。

并

并运算 ($\cup$) 返回两个关系 R 和 S 中元组取并集的结果：

$$R \cup S = \{t \mid t \in R \lor t \in S\}$$

应满足：

- R 和 S 中属性个数要相同；
- R 和 S 中的属性应存在一一对应关系；
- R 中每个属性的域和 S 中对应属性的域要相同。

差

差运算 ($-$) 返回在关系 R 中但是不在关系 S 中的元组集合：

$$R - S = \{t \mid t \in R \lor t \notin S\}$$

同样需满足：

- R 和 S 中属性个数要相同；
- R 和 S 中的属性应存在一一对应关系；
- R 中每个属性的域和 S 中对应属性的域要相同。

笛卡尔积

笛卡尔积运算 ($\times$) 返回关系 R 中元组和关系 S 中的元组做笛卡尔积的结果：

$$R \times S = \{(t, q) \mid t \in R \land q \in S\}$$

其中 (t, q) 为 R 中元组 t 和 S 中元组 q 拼接在一起得到的元组。$R \times S$ 中有 $|R| \times |S|$ 个元组。

例

如果我们有 n 个关系，每个关系分别有 $k_1, k_2, \ldots, k_n$ 个元组，那么笛卡尔积的基数是 $\prod_{i=1}^{n} k_i$。

重命名

重命名运算 (ρ) 将关系 R 重命名为关系 S：

$$\rho_{S(A_1,A_2,\ldots,A_n)}(R)$$

同时将各属性按照从左到右的顺序重命名为 $A_1, A_2, \ldots, A_n$。
$\rho_S(R)$：只修改关系名，不修改属性名。

1.3.2　附加关系代数

交

并运算 ($\cap$) 返回两个关系 R 和 S 中元组取并集的结果：

$$R \cap S = \{t \mid t \in R \wedge t \in S\}$$

应满足：

- R 和 S 中属性个数要相同；
- R 和 S 中的属性应存在一一对应关系；
- R 中每个属性的域和 S 中对应属性的域要相同。

交运算 ($\cap$) 可以通过差运算 ($-$) 来表示：

$$R \cap S = R - (R - S)$$

连接

连接运算 ($\bowtie$) 返回关系 R 和 S 笛卡尔积运算结果中满足一定条件的元组：

$$R \underset{p}{\bowtie} S = \{(t, q) \mid t \in R \wedge q \in S \wedge p(t, q) = \text{True}\}$$

其中 p 是选择谓词。
连接运算 ($\bowtie$) 可通过组合笛卡尔积运算 ($\times$) 和选择运算 (σ) 来表示。

自然连接　自然连接是一种特殊的等值连接。自然连接将连接条件指定为 R 和 S 中属性名相同的列做等值连接，因此 p 可省略。

外连接　外连接是连接运算的扩展，可以处理缺失值。

- 左外连接 ($R \overset{\frown}{\bowtie} S$) 会保留左边关系 R 的所有元组，对于 R 中的元组，若在 S 中没有在同名属性上取值相同的元组，会用空值来填充 S 中的属性。
- 右外连接 ($R \overset{\frown}{\bowtie} S$) 会保留右边关系 S 的所有元组，对于 S 中在 R 中不存在同名属性上取值相同的元组，会用空值来填充 R 中的属性。
- 全外连接 ($R \overset{\frown}{\bowtie} S$) 的查询结果是左外连接和右外连接查询结果的并集。

赋值

赋值运算 (←) 将 ← 右侧的关系代数表达式结果赋值给 ← 左侧的关系变量：

$$T \leftarrow E$$

其中 T 为临时关系变量，E 为关系代数表达式。
赋值运算可以分解复杂的关系代数表达式，使查询变得简单。

例

$$t_1 \leftarrow \text{火车票记录} \times \text{乘客}$$

$$t_2 \leftarrow \sigma_{\text{火车票记录. 乘客 ID} = \text{乘客. 乘客 ID}}(t_1)$$

$$t_3 \leftarrow \pi_{\text{乘客 ID, 姓名, 性别, 车次号, 车票号}}(t_2)$$

除

设 $R(A_1, A_2, \dots, A_m, B_1, B_2, \dots, B_n)$ 和 $S(B_1, B_2, \dots, B_n)$ 是两个关系，则 $R \div S$ 的属性为 $A_1, A_2, \dots, A_m$，且

$$R \div S = \{t \mid t \in \Pi_{A_1, A_2, \dots, A_m}(R) \land (\forall q \in S, (t, q) \in R)\}$$

除运算会返回 R 中在属性 $A_1, A_2, \dots, A_m$ 上的元组 t，其中元组 t 和关系 S 中的任意元组 q 的组合都会出现在关系 R 中。如果 S 中有 R 中没有的属性，则无法进行除运算。

1.3.3　扩展关系代数

广义投影

广义投影运算 (Π) 允许在投影列表中使用算术运算和字符串函数等来对投影运算进行扩展：

$$\Pi_{F_1, F_2, \dots, F_k}(R)$$

其中 R 为关系，$F_1, F_2, \dots, F_k$ 为包含常量、变量、运算符、函数等的多个表达式。

例

$$\Pi_{\text{ID,name,gender,2023}-\text{age}}(\text{Passenger})$$

聚集

聚集运算 ($\mathcal{G}$) 可以查询关系 R 按某些列的值聚集在一起的结果：

$$\mathcal{G}_{F_1(A_1), F_2(A_2), \dots, F_k(A_k)}(R)$$

其中 $A_1, A_2, \dots, A_k$ 为 R 中等属性列，F_i 为作用在属性 A_i 上的聚集函数。
常见的聚集函数包括 count, sum, avg, min, max 等。

分组

分组运算首先对关系 R 中的元组按照某些列的值进行分组，然后在各组上应用聚集运算：

$$_{G_1,G_2,\ldots,G_l}\mathcal{G}_{F_1(A_1),F_2(A_2),\ldots,F_k(A_k)}(R)$$

其中 $G_1, G_2, \ldots, G_l$ 是用来分组的一系列属性，为 R 中等列，在这些列上取值都相同的元组将被分到同一组。查询结果中会包含 $G_1, G_2, \ldots, G_l$ 和 $F_1(A_1), F_2(A_2), \ldots, F_k(A_k)$ 列。

排序

排序运算 (τ) 将关系 R 中的元组按照一列或多列的值排序：

$$\tau_{A_1,A_2,\ldots,A_k}(R)$$

其中 $A_1, A_2, \ldots, A_k$ 是用来排序的列。首先将 R 中的元组按照 A_1 的值排序，对于 A_1 列取值相同的元组，按照 A_2 的值排序，以此类推。

1.3.4　运算优先级

关系代数表达式中各运算符号的计算顺序为从左到右，括号具有最高优先级。

第二章　SQL 语句的设计与应用

SQL 是一种结构化的、便于计算机所理解的查询语言。
关系代数是关系型数据库理论的一部分，是 SQL 的基础：SQL 在执行时需要先转为等价的关系代数表达式。
关系代数和 SQL 之间可以进行相互转换：

关系代数运算	对应的 SQL 语句	关系代数运算	对应的 SQL 语句
选择运算 (σ)	`WHERE`	连接运算 ($\bowtie$)	`JOIN`
投影运算 (Π)	`SELECT`	赋值运算 ($\leftarrow$)	`AS`
并运算 ($\cup$)	`UNION`	除运算 ($\div$)	`NOT EXISTS`
差运算 ($-$)	`EXCEPT`	去重运算 (δ)	`DISTINCT`
笛卡尔积运算 ($\times$)	`FROM`	广义投影运算 (Π)	`SELECT`
重命名运算 (ρ)	`AS`	聚集运算 ($\mathcal{G}$)	聚集函数
交运算 ($\cap$)	`INTERSECT`	分组运算 ($\mathcal{G}$)	`GROUP BY`

2.1　SQL 数据定义语言

2.1.1　创建表

```
CREATE TABLE <表名> (
    <列名> <数据类型> [列级完整性约束] ... [列级完整性约束]
    [, <列名> <数据类型> [列级完整性约束] ... [列级完整性约束]] ...
    [, <列名><数据类型>[列级完整性]]
    [, 表级完整性约束]
    ...
    [, 表级完整性约束]
);
```

表的数据类型

	数据类型	描述
文本型	CHAR(n)	长度为 n 的定长字符串
	VARCHAR(n)	长度为 n 的变长字符串
数字型	INT	整数（4 字节）
	SMALLINT	短整数（2 字节）
	BIGINT	大整数（8 字节）
	FLOAT	单精度浮点数
	DOUBLE	双精度浮点数
	DECIMAL(p,d)	定点数，由 p 位数字组成，小数点后有 d 位数字
	BOOLEAN	布尔型
时间型	DATE	日期，包含年、月、日，格式为 YYYY-MM-DD
	TIME	时间，包含时、分、秒，格式为 HH:MM:SS
	TIMESTAMP	时间戳，格式为 YYYY-MM-DD HH:MM:SS

MySQL 的基本构建

```
CREATE DATABASE student_db;  -- 创建数据库
USE student_db;              -- 使用数据库
CREATE TABLE student(...);   -- 创建表
SHOW tables;                 -- 查看数据库中所有的表
DESC student;                -- 查看表中的内容
```

完整性约束

```
CREATE TABLE Student (
    student_id CHAR(10) NOT NULL,        -- 列级完整性约束：非空约束
    student_name VARCHAR(50) NOT NULL,
    gender CHAR(2),
    age INTEGER,
    department VARCHAR(50),
    PRIMARY KEY (student_id)             -- 表级完整性约束：主键约束
);

CREATE TABLE Course (
    course_id VARCHAR(10) NOT NULL,
    course_name VARCHAR(50) NOT NULL,
    prior VARCHAR(50),
    credit INTEGER,
    PRIMARY KEY (course_id),
    UNIQUE UQ_Course_course_name(course_name)    -- 表级完整性约束：唯一性约束
);

CREATE TABLE CourseSelection (
    student_id CHAR(10),
    course_id VARCHAR(50),
    grade INTEGER NOT NULL,
    KEY (course_id),
    KEY (student_id)
```

```
);
-- 表级完整性约束：主键约束
ALTER TABLE CourseSelection ADD CONSTRAINT FK_CourseSelection_Course
    FOREIGN KEY (course_id) REFERENCES Course (course_id);
ALTER TABLE CourseSelection ADD CONSTRAINT FK_CourseSelection_Student
    FOREIGN KEY (student_id) REFERENCES Student (student_id);
```

比较和讨论有何不同?

```
CREATE TABLE CourseSelection (
    student_id CHAR(10),
    course_id VARCHAR(50),
    grade INTEGER NOT NULL,
    KEY (course_id),
    KEY (student_id),
    FOREIGN KEY (course_id) REFERENCES Course (course_id),
    FOREIGN KEY (student_id) REFERENCES Student (student_id)
);
```

2.1.2　修改表

- 添加列：ALTER Table <表名> ADD [COLUMN] <列名> <数据类型>
- 删除列：ALTER Table <表名> DROP [COLUMN] [RESTRICT | CASCADE]
  - RESTRICT：如果该列被其它列引用，则无法删除该列；
  - CASCADE：引用该列的其它列会和该列同时被删除。
- 修改列：ALTER Table <表名> CHANGE COLUMN <列名> <列名> <数据类型>

2.1.3　删除表

```
DROP Table <表名> [, <表名>] ... [, <表名>]
```

包含外键关系的表需要先删除外键才能删除表。

2.2　SQL 数据操作语言

2.2.1　数据查询

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ... [into_option]  -- 指定要显示的属性列
    [FROM table_references]                                            -- 表名或视图名
    [WHERE where_condition]                                            -- 选择条件
    [GROUP BY {col_name | expr | position}, ...]                       -- 分组属性
    [HAVING where_condition]                                           -- 分组筛选条件
    [ORDER BY {col_name | expr | position}] [ASC | DESC]               -- 指定排序顺序
```

选择和投影

```
SELECT student_id, student_name                    -- Π_{A1,A2,...,An}(R) = {t[A1, A2, ..., Ak] | t ∈ R}
    FROM Student
    WHERE department='CS' AND student_name='Mike'; -- σ_p(R) = {t | t ∈ R ∧ p(t) = True}
```

广义投影

```
SELECT 2023-age AS birthday, LOWER(student_name)
    FROM Student
    WHERE department='CS' AND student_name='Mike';
```

选择条件

选择操作通过 WHERE 子句选择若干满足条件的元组。WHERE 子句后面跟着一个条件表达式，满足该条件表达式的元组会被返回。
WHERE 子句常用查询条件：

查询条件	谓词
比较	=, >, <, >=, <=, !=, <>, !>, !<, NOT
确定范围	BETWEEN ... AND ..., NOT BETWEEN ... AND ...
确定集合	IN, NOT IN
空集	IS NULL, IS NOT NULL
逻辑运算	AND, OR
字符串运算	LIKE, NOT LIKE, %, _, ESCAPE

LIKE 可以用来查询与匹配串匹配的字符串：

- 匹配串一般由字符和通配符组成，通配符包括 _ 和 %；
- _：匹配字符串中的任意一个字符；
- %：匹配字符串中的任意多个字符（包括 0 个和 1 个）；
- 当字符串的所有字符均可匹配成功时，该字符串的 LIKE 查询结果为真；
- 当匹配串中本身就包含通配符 _ 或 % 时，需要在匹配串中包含的 _ 或 % 前面加上转义字符来进行转义，同时需要使用 ESCAPE <换码字符> 来指定换码字符。

DISTINCT：查询结果会消除取值重复的行。

聚集操作

为了查询一些数据聚集后的结果，比如查询一些统计值（平均值、最大值、最小值等），SQL 提供了许多聚集函数。
常见的聚集函数包括：

聚集函数	含义
COUNT([DISTINCT | ALL *])	统计元组个数
COUNT([DISTINCT | ALL <列名>])	统计一列值的个数
SUM([DISTINCT | ALL <列名>])	统计一列值的总和
AVG([DISTINCT | ALL <列名>])	统计一列值的平均值
MAX([DISTINCT | ALL <列名>])	统计一列值的最大值
MIN([DISTINCT | ALL <列名>])	统计一列值的最小值

例

查找所有学生数：SELECT COUNT(*) FROM Student;

分组操作

GROUP BY 子句可以将查询到的满足条件的元组按照某一列或多列的值进行分组，值相等的为一组。

- 未分组：聚集函数作用于整个查询结果；
- 分组：聚集函数分别作用于每个组

例

查询每门课程的选课人数：

```
SELECT course_id, COUNT(*) FROM courseselection GROUP BY course_id;
```

查询选课人数大于 5 人的课程：

```
SELECT course_id, COUNT(*) FROM courseselection GROUP BY course_id HAVING COUNT(*)>5;
```

排序操作

ORDER BY 子句将查询结果按照某一列（或者某多列）的值进行排序。
DESC：降序排序；ASC：升序排序（默认选项）。

连接操作

同时涉及多个（两个以及上）表的查询称为连接查询。在连接操作中，需在 FROM 子句中指定需要连接的表，在 WHERE 子句中指定连接条件。
连接条件的常见格式为：

```
[<表名1>.]<列名1> <比较运算符> [<表名2>.]<列名2>
[<表名1>.]<列名1> BETWEEN [<表名2>.]<列名2> AND [<表名2>.]<列名3>
```

2.2.2 数据更新

在执行插入、修改和删除语句时会检查所插入、修改和删除的元组是否破坏表中的完整性约束。如果不满足完整性约束，则可能会执行失败！

插入数据

```
INSERT INTO <表名> [(<列名1> [, <列名2>] ... [, 列名n])]
    VALUES (<常量1> [, <常量2>] ... [, 常量n])
```

对于没有在 INTO 子句中出现的列，新元组在这些列上的取值为空值。

修改数据

```
UPDATE <表名>
    SET <列名1>=<表达式1>, <列名2>=<表达式2>, ... <列名n>=<表达式n>,
    WHERE <条件>;
```

如果不包含 WHERE 子句，则表中的所有元组都将被修改。

删除数据

```
DELETE FROM <表名> [WHERE <条件>];
```

如果不包含 WHERE 子句，则表中的所有元组都将被删除。

2.2.3　约束

关系数据库中的约束保证了对数据库操作的任何修改都不会违反数据一致性。
实际生产环境中，有时候也许会放弃建立约束，可能因为：

- 某些数据的约束太过复杂，不适合在数据库层面验证；
- 会产生额外性能开销（例如在高并发环境下）；
- 历史数据可能会应用新规则。

实体完整性约束（主键）

参照完整性约束（外键）

自增长约束

```
CREATE TABLE ExchangeStudent (
    student_id INTEGER NOT NULL AUTO_INCREMENT,
    -- 通常用于主键，它允许每次插入新记录时自动生成一个唯一的数字值，插入数据时可缺省
    student_name VARCHAR(50) NOT NULL,
    home_university VARCHAR(50),
    age INTEGER,
    PRIMARY KEY (student_id)
);
```

默认值约束

```
CREATE TABLE ExchangeStudent (
    student_id INTEGER NOT NULL AUTO_INCREMENT,
    student_name VARCHAR(50) NOT NULL,
    home_university VARCHAR(50) DEFAULT 'Xi''an Jiao Tong University',  -- 插入数据时可缺省
    age INTEGER,
    PRIMARY KEY (student_id)
);
```

检查约束

```
CREATE TABLE ExchangeStudent (
    student_id INTEGER NOT NULL AUTO_INCREMENT,
    student_name VARCHAR(50) NOT NULL,
    home_university VARCHAR(50),
    age INTEGER CHECK (age>0),
    PRIMARY KEY (student_id)
);

ALTER TABLE ExchangeStudent
    ADD CONSTRAINT chk_age_positive
    CHECK (age > 0);
```

2.2.4　索引

索引可以加速查询操作。
在某属性上建立索引，可以快速定位到在该属性上取值的元组。
用户可以在一个表上建立一个或多个索引，以加速查询。
数据库中的索引有很多类型，包括 B+ 树索引，散列索引，位图索引等。

建立索引

```
CREATE INDEX <索引名>
    ON <表名> (<列名> [ASC | DESC] [, <列名>[ASC | DESC], ..., <列名>[ASC | DESC], ])
```

删除索引

```
DROP INDEX <索引名> ON <表名>;
```

```
ALTER TABLE <表名> DROP INDEX <索引名>;
```

索引的优劣

- 优点：
  - **提高查询速度**：索引允许数据库快速查找和检索数据，从而极大地提高查询的响应时间。
- 缺点：
  - **增加空间需求**：索引本身需要存储空间。对于大型的数据表，索引可能会占用大量的磁盘空间。
  - **插入、更新和删除的开销**：当对表中的数据进行插入、更新或删除操作时，相关的索引也需要被相应地更新，这会增加操作的时间开销。
  - **可能导致过度优化**：不恰当地选择索引策略或创建过多的索引可能会使某些查询的性能降低，因为查询优化器需要花费更多时间来选择最佳的索引。

2.2.5　视图

有些时候，有一些查询我们会反复用到。

- 创建视图：`CREATE VIEW <视图名> [<列名> (, <列名>, ..., <列名>)] AS <子查询>`
- 删除视图：`DROP VIEW <视图名>`
- 查询视图：`SELECT ... FROM <视图名> WHERE ...`

讨论与分析

视图是一种「虚关系」，实际查询时需要根据定义查询底层关系，当存在大量这样的查询时会有较高的成本。某些数据库支持物化视图，像存储表一样将创建的视图关系「物化」存储在数据库中。物化视图的创建、修改与删除语法同视图类似，区别是多了关键字 MATERIALIZED。
在 MySQL 中，可以对视图进行更新，本质上会对视图的表中的数据进行更新。

- 优势：– 简化操作 – 数据抽象 – 安全性
- 劣势：– 错误风险 – 性能问题

2.3　SQL 存储过程和函数

2.3.1　存储过程

存储过程和函数是事先经过编译并存储在数据库中的一段 SQL 语句的集合。
存储过程和函数可以对一段代码进行封装，以便日后调用。
数据库中创建存储过程的语句为 **CREATE PROCEDURE**，并通过 **CALL** 语句加存储过程名来调用存储过程。
数据库中创建函数的语句为 **CREATE FUNCTION**，并通过函数名来调用函数。

定义存储过程

```
DELIMITER //
CREATE PROCEDURE <存储过程名>([参数, ..., 参数])     -- 关键字，表示创建一个新的存储过程
BEGIN
<SQL 语句>
END <终止符>
DELIMITER ;
```

默认的语句结束符是;，但是在定义存储过程时，为了避免混淆，通常会临时改变它。
参数可以有以下几种模式：

- **IN**：输入参数。调用存储过程时必须提供的值。
- **OUT**：输出参数。存储过程结束后返回给调用者的值。
- **INOUT**：输入/输出参数。既可以作为输入也可以作为输出。

每个参数都有一个名称和数据类型，例如：**IN** param1 **INT**, **OUT** param2 **VARCHAR**(50)。

查看存储过程

```
SHOW PROCEDURE STATUS WHERE DB='your_database_name';
```

删除存储过程

```
DROP PROCEDURE [IF EXISTS] procedure_name;
```

变量声明

```
DECLARE variable_name data_type [DEFAULT default_value];
DECLARE v_counter INT;
DECLARE v_name VARCHAR(50) DEFAULT 'John Doe';
DECLARE v_date DATE;
```

变量赋值

```
SET variable_name=value;
SET v_counter=10;
SELECT <SQL 语句> INTO v_counter FROM <表名>;
SELECT COUNT(*) INTO v_counter FROM student;
```

变量类型

	数据类型	描述
数字	INT	整数类型
	TINYINT	很小的整数
	SMALLINT	小的整数
	MEDIUMINT	中等大小的整数
	BIGINT	大的整数
	FLOAT	单精度浮点数
	DOUBLE	双精度浮点数
	DECIMAL	小数点固定的数值
字符串	CHAR	固定长度的字符串
	VARCHAR	可变长度的字符串
	TEXT	长文本数据
	MEDIUMTEXT	中等长度的文本数据
	LONGTEXT	非常长的文本数据
日期和时间	DATE	日期，包含年、月、日，格式为 YYYY-MM-DD
	TIME	时间，包含时、分、秒，格式为 HH:MM:SS
	DATETIME	日期和时间，格式为 YYYY-MM-DD HH:MM:SS
	TIMESTAMP	时间戳，自动设置为当前日期和时间
	YEAR	年份
二进制字符串	BINARY	固定长度的二进制字符串
	VARBINARY	可变长度的二进制字符串
	BLOB	二进制长对象数据
枚举和集合	ENUM	字符串对象，其值限制为预定义的枚举列表中的值
	SET	字符串对象，其值是预定义的字符串列表中的零个或多个值

调试

在 MySQL 存储过程中，调试并不如传统的 Java 或者 Python 程序方便，一般来说来说我们使用 SELECT 来打印存储过程中的中间执行信息。

```
SELECT <变量名>;
```

控制判断

```
IF condition THEN
    -- statements
ELSEIF another_condition THEN
    -- other statements
ELSE
    -- more statements
END IF;
```

例

```
DELIMITER //
CREATE PROCEDURE CheckNumber(IN num INT, OUT result_msg VARCHAR(50))
BEGIN
    IF num > 0 THEN
        SET result_msg = 'The number is positive.';
    ELSEIF num < 0 THEN
        SET result_msg = 'The number is negative.';
    ELSE
        SET result_msg = 'The number is zero.';
    END IF;
END //
DELIMITER ;

SET @message = '';
CALL CheckNumber(5, @message);
SELECT @message;
```

用户定义的会话变量是特定于当前会话的，意味着它们只在当前会话中存在，并且会在会话结束时被销毁。

多条件判断

```
CASE value
WHEN value1 THEN
    -- statements
WHEN value2 THEN
    -- other statements
ELSE
    -- default statements
END CASE;
```

例

```
DELIMITER //
CREATE PROCEDURE MonthNameFromDays(IN days INT, OUT month_name VARCHAR(50))
BEGIN
    SET month_name =
    CASE
        WHEN days BETWEEN 1 AND 30 THEN 'January'
        WHEN days BETWEEN 31 AND 60 THEN 'February'
        WHEN days BETWEEN 61 AND 90 THEN 'March'
        WHEN days BETWEEN 91 AND 120 THEN 'April'
        WHEN days BETWEEN 121 AND 150 THEN 'May'
        WHEN days BETWEEN 151 AND 180 THEN 'June'
        WHEN days BETWEEN 181 AND 210 THEN 'July'
        WHEN days BETWEEN 211 AND 240 THEN 'August'
        WHEN days BETWEEN 241 AND 270 THEN 'September'
        WHEN days BETWEEN 271 AND 300 THEN 'October'
        WHEN days BETWEEN 301 AND 330 THEN 'November'
        WHEN days BETWEEN 331 AND 360 THEN 'December'
        ELSE 'Invalid days input'
    END;
END //
DELIMITER ;

SET @month_result = '';
CALL MonthNameFromDays(45, @month_result);
SELECT @month_result;
```

注：你这月份，它保真吗?

循环

```
WHILE condition DO
    -- statements
END WHILE;
```

游标

在 SQL 中，游标是用于遍历并逐行处理查询结果集的机制。它的主要作用是在存储过程、函数或触发器中对结果集中的每一行执行某种操作。游标的使用步骤如下：

- 游标声明：`DECLARE cursor_name CURSOR FOR <SELECT_statement>;`
- 打开游标：`OPEN cursor_name;`
- 获取数据：`FETCH cursor_name INTO variable_name;`
- 关闭游标：`CLOSE cursor_name;`

例

```
DELIMITER //
CREATE PROCEDURE ProcessData()
BEGIN
    -- 变量声明必须放在最前面
    DECLARE count INT;
    DECLARE id CHAR(10);
    DECLARE name VARCHAR(50);
    -- 定义游标，指向年龄最小的学生
    DECLARE cur CURSOR FOR SELECT student_id, student_name FROM Student ORDER BY age;
    SET count = 0;
    OPEN cur;                           -- 打开游标
    WHILE count<10 DO
        FETCH cur INTO id, name;
        SELECT CONCAT(id, ' ', name);
        SET count = count + 1;
    END WHILE;
    CLOSE cur;                          -- 关闭游标
END //
DELIMITER ;
```

2.3.2 函数

函数是一个预编译的 SQL 代码段，它可以接受参数、执行操作并返回一个值。函数通常用于计算并返回单个值，而不是结果集。

```
CREATE FUNCTION <函数名> ([<参数>, ..., <参数>])
    RETURN <数据类型>
BEGIN
    <SQL 语句>
END <终止符>
```

例

```
DELIMITER //
CREATE FUNCTION AddTwoNumbers (a INT, b INT)
    RETURNS INT
    DETERMINISTIC
BEGIN
    RETURN a + b;
END //
DELIMITER ;

SELECT AddTwoNumbers(5, 3);
```

存储过程和函数的区别

- 存储过程可以通过 OUT 或 INOUT 参数返回多个值，而函数只能返回 RETURNS 子句中指定的某一类型的单值或表对象。
- 存储过程的参数可以为 IN、OUT 或 INOUT，而函数的参数只能是 IN 类型的。
- 存储过程可以通过 CALL 语句作为一个独立的部分来调用和执行，而函数可以作为查询语句的一部分来调用。

- 创建函数时必须指定返回值数据类型，且函数体内必须有一个 RETURNS 语句。

2.4　SQL 触发器

触发器是与表相关的特殊的存储过程，在满足特定条件时，它会被触发执行。
触发器是定义在基本表上的，当基本表被修改（比如插入、删除、更新数据）时，会激活定义在其上的触发器，该基本表称为触发器的目标表。
触发器可以用来保证数据库的完整性。

```
CREATE TRIGGER <触发器名>
    [BEFORE | AFTER] [INSERT | DELETE | UPDATE] ON <表名>
FOR EACH ROW
    <触发动作体>

DROP TRIGGER <触发器名>
```

对于触发事件作用的每一行（FOR EACH ROW），会执行触发动作体。
对于触发事件作用的每一行，在触发事件发生之前该行称之为 OLD，在触发事件发生之后该行称之为 NEW。
可以使用 OLD 和 NEW 来访问触发事件发生前后的元组的值。

例

```
DELIMITER //
CREATE TRIGGER CheckCourseGrades
AFTER INSERT ON CourseSelection
FOR EACH ROW
BEGIN
    DECLARE avg_grade DECIMAL(5, 2);
    SELECT AVG(grade) INTO avg_grade
    FROM CourseSelection
    WHERE course_id = NEW.course_id
        AND year = NEW.year
        AND term = NEW.term;
    IF avg_grade < 60 THEN
        INSERT INTO ExamExceptions (student_id, course_id, year, term, average_grade)
        VALUES (NEW.student_id, NEW.course_id, NEW.year, NEW.term, avg_grade);
    END IF;
END //
DELIMITER ;
```

2.5　SQL 事务控制

事务是数据库管理系统（DBMS）的一个操作单元或工作单元，通常有原子性、一致性、隔离性和持久性的特点。

- 开始事务：START TRANSACTION
- 提交事务：COMMIT
- 回滚事务：ROLLBACK
- 在事务内部设置回滚标记点：SAVEPOINT sp_name
- 删除回滚标记点：RELEASE SAVEPOINT

- 将事务回滚到标记点：`ROLLBACK TO sp_name`

2.6　SQL 数据控制语言

2.6.1　用户权限

创建用户

```
CREATE USER 'username'@'hostname' IDENTIFIED BY 'password';
```

'hostname' 是允许连接到 MySQL 服务器的主机。通常，可以将其设置为通配符 %，以允许来自任何主机的连接。如果要限制连接到特定主机，提供主机的名称或 IP 地址。

删除用户

```
DROP USER 'username'@'hostname';
```

查看用户

```
SELECT user, host FROM mysql.user;
```

权限授予

```
GRANT <权限>[, <权限>, <权限>, ..., <权限>]
    ON <对象类型> <对象名>[, <对象类型><对象名>, ..., <对象类型><对象名>]
    TO <用户名>[, <用户名>, ..., <用户名>]
    [WITH GRANT OPTION];
```

权限收回

```
REVOKE <权限>[, <权限>, <权限>, ..., <权限>]
    ON <对象类型> <对象名>[, <对象类型><对象名>, ..., <对象类型><对象名>]
    FROM <用户名>[, <用户名>, ..., <用户名>]
    [CASCADE | RESTRICT];
```

CASCADE：支持级联收回，即由这些用户授予了以上权限的用户的这些权限也会被收回（默认选项）；
RESTRICT：MySQL 不支持级联收回。

第三章　数据库设计与理论

3.1　数据库设计流程

数据库设计核心步骤：

- 现实世界需求分析
- 概念结构设计阶段
- 逻辑结构设计阶段
- 物理结构设计阶段
- 数据库建设和维护

设计目标：

1. 满足用户对数据内容的要求；
2. 提供自然易懂的数据结构；
3. 支持应用所需的数据处理要求和性能目标（如：响应时间、处理时间、存储空间等）

3.1.1　概念结构设计阶段

通过对用户需求进行综合、归纳与抽象，形成一个独立于具体数据库管理系统的概念模型。
将现实世界的客观事物及其关系抽象为「实体」和「联系」等形式，用于描述业务领域的数据对象及其关系，常采用 Entity-Relationship (E-R) 模型。

E-R 模型	例
实体	学生/课程
属性	学生/课程号
联系	学生与课程实体之间存在「选课」联系

3.1.2　逻辑结构设计阶段

在概念模型的基础上，需要进一步考虑这些数据对象在计算机系统中的逻辑表示，形成某个数据库管理系统所支持的数据模型。
常见的逻辑数据模型有关系模型、层次模型和网状模型等。

关系模型	表示
关系	表
属性	列
域	属性值的类型和范围
元组	行
属性值	表中某个单元的值

3.1.3　物理结构设计阶段

在逻辑模型的基础上，具体考虑数据对象如何在数据库管理系统中物理实现。
根据数据库管理系统的特点和处理需要，进行物理存储安排，明确存储结构和存取方法，如：关系表、索引的设计。

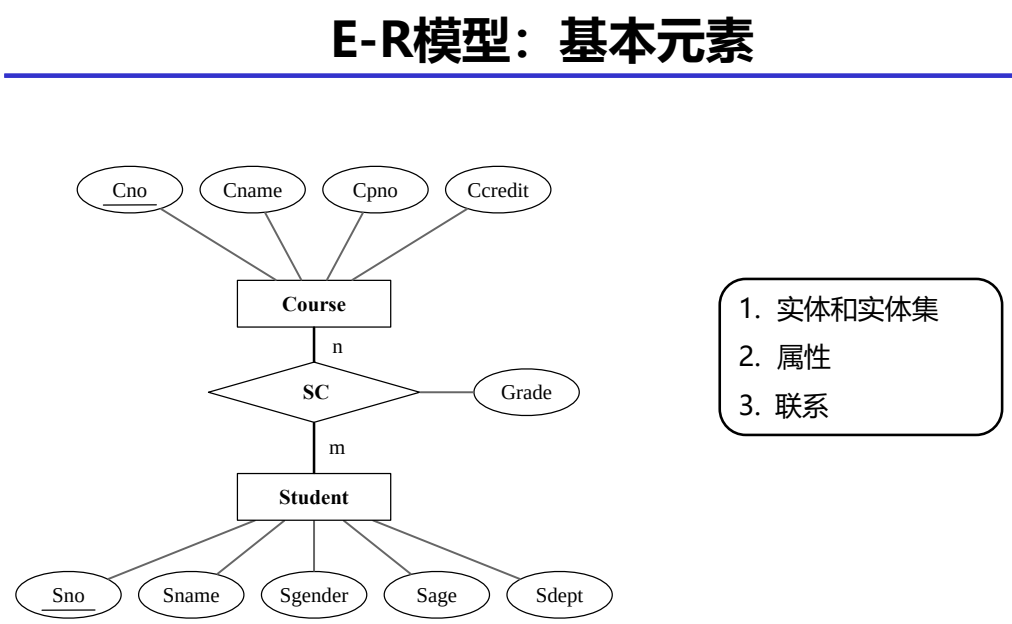
3.1.4　E-R 图转换为关系模式

在关系数据库的设计中，一个核心步骤就是将 E-R 图转化为关系模式。

3.2　概念结构设计：E-R 模型

E-R 模型是实体-联系模型 (entity-relationship model) 的简称，是用于描述现实世界的概念数据模型，也可以用于表示关系数据库的结构。
在数据库设计领域有着广泛的应用目前大部分的数据库设计工具和产品均采用 E-R 模型进行数据库的概念结构设计。

3.2.1　E-R 模型的基本元素



实体和实体集

实体是对现实世界中事物数据概念的某种抽象。
多个具有相同性质的同类实体构成的集合，称为实体集。

属性

实体集都可以被一组特征来描述，这些用来描述实体集的数据特征被称为实体集的属性。
通常不同实体集的属性是不同的。

标识符

标识符是可以唯一标识不同的实体集的某一属性。
当必须选取多个属性的组合来作为实体的唯一标识，称该属性的组合为复合标识符。

联系

正如现实世界中事物之间都有某种联系一样，这些事物在数据库中也必然存在联系。
属于不同实体集的实体之间，学生实体和课程实体之间的「选课」联系；
属于同一实体集的实体之间，课程实体和课程实体之间的「先修」联系。
E-R 模型中的联系一般是用动词来命名。

3.2.2　E-R 图

E-R 图是用来描述实体集、属性和联系的图形化表示。

- 实体集：用矩形表示，矩形框内标注实体集名。

- 属性：用椭圆形表示，并用无向边将其与相应的实体集连接起来。
  - 实体集的标识符用下划线标识出来。
- 联系：用菱形表示，菱形框内标注联系名，并用无向边分别与有关实体集连接起来，同时在无向边旁标上联系的类型（$1:1$，$1:n$ 或 $m:n$）。
  - 联系可具有属性：如果实体之间的联系也具有属性，则用无向边连接属性和菱形。

3.2.3　E-R 联系类型

E-R 联系类型是指实体之间不同关系的形式。

一元联系

实体集内部实体之间的联系称为一元联系。
学生实体之间存在同级「管理」的关系，例如，学生群体中设立了班长一职，协助老师管理学生。

二元联系

两个实体集之间的联系称为二元联系。
例如，学生实体集与课程实体集之间存在「选择」课程的联系。
一元联系可以看做是特殊的二元联系。

三元联系

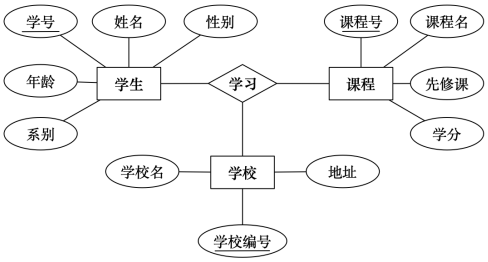
三个实体集之间的联系称为三元联系。
例如，学生、学校和课程之间存在「学习」的联系，即某学生在某学校学习某课程。

三元联系分解为二元联系

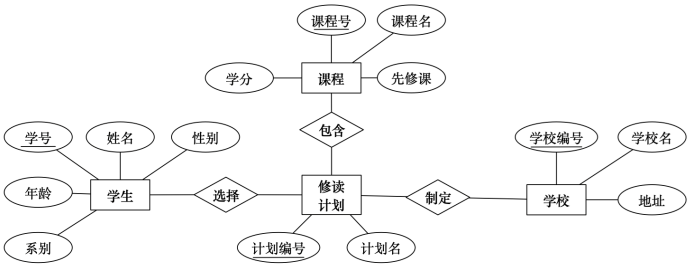
针对三元联系，可以将其分解为二元联系。
考虑一个抽象的三元联系 R，它将实体集 A、B 和 C 联系起来，将该三元联系转换成多个等价的二元联系的具体步骤如下：

1. 用新实体集 E 替代联系 R。如果联系 R 有属性，则将这些属性赋给新建的实体集 E；为 E 建立一个特殊的标识性属性。因为每个实体集都应该至少有一个属性或多个属性的集合，以区别实体集中的各个成员。
2. 建立三个新的联系 R_A, R_B, R_C。其中，R_A 是实体集 E 和 A 之间的联系；R_B 是实体集 E 和 B 之间的联系；R_C 是实体集 E 和 C 之间的联系。
3. 针对联系 R 中的每个联系 (a_i, b_i, c_i)，在 E 中创建一个新实体 e_i，e_i 代表 (a_i, b_i, c_i)。然后，在三个新联系集中分别建立新的联系：在 R_A 中插入 (e_i, a_i)；在 R_B 中插入 (e_i, b_i)；在 R_C 中插入 (e_i, c_i)。

三元联系分解为二元联系：示例



三元联系示例



三元联系分解为二元联系示例

二元联系的不同类型

在实际应用中，二元联系是最常见的实体联系类型。对于二元联系，又能细分成以下几种类型：

- 一对一联系 $(1:1)$
- 一对多联系 $(1:n)$
- 多对多联系 $(m:n)$

联系约束：一个实体通过联系可以关联到的实体的个数。

一对一联系　如果实体集 A 中每一个实体最多可以和另一实体集 B 中的一个实体有关系，反之亦然，则称实体集 A 和实体集 B 具有一对一联系，记作 $1:1$。

一对多联系　如果实体集 A 中的每一个实体，实体集 B 中有 n 个实体 $(n \geq 0)$ 与之相关，则称实体集 A 与实体集 B 有一对多联系，记作 $1:n$。

多对多联系　如果实体集 A 中的每个实体在实体集 B 中有 n 个实体 $(n \geq 0)$ 与之相关，反之，实体集 B 中的每个实体都在实体集 A 中有 m 个实体 $(m \geq 0)$ 与之相关，则称 E 和 F 具有多对多联系，记为 $m:n$。

N 元联系

涉及 N 个实体的复杂联系。N 个实体之间也存在一对一、一对多、多对多联系。

例

课程、教师、参考书三个实体集，假设一门课可以有若干个教师讲授，使用若干本参考书，而每个教授只讲授一门课，每一本参考书只供一门课程使用，则课程与教师、参考书之间的联系是一对多的。
课程、教师、参考书三个实体集，假设一门课可以有若干个教师讲授，使用若干本参考书，而每个教授可以讲授若干门课，每一本参考书可以供若干门课使用，则课程与教师、参考书之间的联系是多对多的。

3.2.4　E-R 图设计

实体与属性

实体与属性的划分原则：为了简化 E-R 图，现实世界的事物能作为属性对待的，尽量作为属性对待。

1. 作为属性，不能再具有需要描述的性质。属性必须是不可分的数据项，不能包含其他属性。
2. 属性不能与其他实体具有联系，即 E-R 图中所表示的联系是实体之间的联系。

属性的放置

联系属性 (relationship attributes) 通常只用于多对多联系，但也可以用于一对一和一对多联系。

约束 (Constraints)

约束：用于表达数据应具备的性质或者需要满足的限制，为数据增添了语义信息。

属性约束：域　*域约束* (Domain Constraint) 限制属性的取值范围。

属性约束：键　一组属性的取值能够唯一标识一个实体，则该属性为实体的*键* (key)。*候选键* (candidate key) 是能唯一标识实体的极小属性集合。

联系约束：基数 (Cardinality)　实体参与某一联系的最大次数。

联系约束：参与度 (Participation)　实体参与某一联系的最小次数。

3.2.5　E-R 图的集成

目的：多个分 E-R 图集成为一个 E-R 图。

合并 E-R 图

解决各分 E-R 图之间的冲突，将分 E-R 图合并起来生成初步 E-R 图。
子系统 E-R 图之间的冲突主要有三类：

- 属性冲突
  - 属性域冲突，如：学号可以被定义为整数，也可以被定义为字符型。
  - 属性值冲突，如：体重可以以克为单位，也可以以千克为单位
- 命名冲突
  - 同名异义，如：编号可以是课程编号，也可以是学生编号。

  - 异名同义，如：学号和学生编号。
  - 可能发生在实体、联系、属性。
- 结构冲突
  - 同一对象的不同抽象，如：工资在一些应用中被当作实体，在另一些应用被当作属性。
  - 同一实体包含属性个数和排列次序不同。
  - 实体间的联系在不同 E-R 图中类型不同。

解决方法：协商讨论。

消除冗余

冗余的数据是指可以由基本数据导出的数据。
冗余的联系是指可以由其他联系导出的联系。
不是所有的冗余数据和冗余联系都必须加以消除，有时候为了提高效率，不得不以冗余信息作为代价。
解决方法：规范化理论。

3.3　概念结构设计：从 E-R 图到关系模式

- 实体集的转换
  - 实体集转换成一个关系模式（表头）
  - 实体集的属性转换为关系模式的属性（表的列名）
  - 实体集的标识符转换为关系模式的键
- 联系的转换：参照完整性约束，将 E-R 模型中的联系转化为关系表间的参照完整性约束
- 规范化设计：正确规范化所有关系表

3.3.1　E-R 实体集的转换

将 E-R 图转换为关系模式的基本操作是**将实体集转换为关系表**。

- 实体集的名称 → 关系表的表名
- 实体集的属性 → 关系表的属性
- 实体集的标识符 → 关系表的主键

3.3.2　E-R 联系的转换

将 E-R 模型中的实体集转换成关系模式中的关系表后，还需要进一步将 E-R 模型中的联系转换成关系表之间的参照完整性约束。

- 一元联系：转换规则与二元联系类似。
- 二元联系：是 E-R 联系中最常见的联系类型，做重点介绍。
- 三元联系：可以将其先分解成二元联系，再用二元联系的规则进行转换。

一对一联系的转换

一对一 (1 : 1) 联系的两种转换方式：

1. 在两个实体集转换成的两个关系模式的基础上，在其中任意一个关系模式的属性中加入另一个关系模式的键和该联系自有的属性，这种转换不需要添加新的关系模式。

2. 在两个实体集转换成的两个关系模式的基础上，添加一个新关系模式，该关系模式的属性包含该联系相关的各实体集的标识符以及该联系自有的属性，其候选键可以是每个实体集的标识符。

一对多联系的转换

一对多 (1 : n) 联系的两种转换方式：

1. 在两个实体集转换成的两个关系模式的基础上，在 n 端实体类型转换成的关系模式中加入 1 端实体集的标识符和该联系自有的属性，这种转换不需要添加新的关系模式。
2. 在两个实体集转换成的两个关系模式的基础上，为 1 : n 联系添加一个新关系模式，该关系模式的属性是所有与该联系相关的各实体集的标识符以及该联系自有的属性，该关系模式的主键是 n 端实体集的标识符。

多对多联系的转换

多对多 (m : n) 联系的转换方式：对该联系添加一个新的关系模式，这个新的关系模式的属性是两端实体类型的标识符以及该联系自有的属性，其键为两端实体集的标识符的组合。

3.4　数据库规范化设计理论

关系模式设计不规范会带来一系列问题：• 数据冗余 • 更新异常 • 插入异常 • 删除异常
数据库规范化设计是指在数据库中减少数据冗余和定义一个规范的表间结构，以更好地实现数据完整性和一致性。
数据冗余是指一组数据不必要地重复出现在多个表中。冗余数据会占用更多存储空间并且提高数据库的维护成本，例如维护数据一致性。
数据库规范化设计具有以下几个优点：

1. 降低数据存储和维护数据一致性的成本。即减少冗余数据，减少存储空间的开销；当数据发生更新（以及插入和删除）的时候，不必同时更新那些冗余数据，降低了维护数据一致性的成本。
2. 便于设计合理的关系表间的依赖和约束关系，便于实现数据完整性和一致性，避免插入异常、删除异常、更新异常。
3. 便于设计合理的数据库结构，以提高数据库系统的整体性能。

注意：关系数据库规范化程度不是评估关系数据库模式的唯一准则；某些情况下，规范的关系模式在数据库系统中的实际表现并不好。

3.4.1　函数依赖（判定范式）

为了解决关系模式设计的规范化问题，需要引入一个重要概念：*函数依赖* (functional dependency)。
函数依赖反映了一个关系中属性或者属性组之间相互依存、相互制约的关系，即两个列或者列组之间的约束。
函数依赖定义：给定关系 $R(U)$，X 和 Y 是其列集合 U 的子集，t 和 l 分别是 R 中的任意两个元组。如果 $t[X] = l[X]$，则 $t[Y] = l[Y]$，那么称 Y 函数依赖于 X，或者 X 函数决定 Y，记为 $X \to Y$。如果 Y 不依赖于 X，则记为 $X \nrightarrow Y$。如果 $X \to Y$ 且 $Y \to X$，则 X 与 Y 一一对应，记为 $X \leftrightarrow Y$。
一个函数依赖要成立，不但要求关系 R 中当前的值都能满足函数依赖条件，而且还要求关系中的任一可能取值都满足函数依赖的条件。此外，函数依赖还具有数据语义特征，即函数依赖在某种程度上也是现实世界的反映。

函数依赖不是指关系模式 R 的某个或某些关系实例满足的约束条件，而是指 R 的所有关系实例均要满足的约束条件。

用函数依赖重新定义关系的键

如果一个属性或者多个属性的集合 K 满足下列性质，则认为 K 是关系 $R(U)$ 的*候选键* (candidate key)：

- 其他属性函数依赖于该属性或属性集；
- 其他属性都不函数依赖于该属性或属性集的任一真子集。

如果一个关系中有多个候选键，则选择其最重要的一个候选键作为关系 $R(U)$ 的*主键* (primary key)。
给定关系 $R(U)$，$K \subseteq U$ 是 R 的*超键*当且仅当 $K \to U$。
与关系的键相比，函数依赖能够（语义上）表达更多约束。

函数依赖的类型

平凡与非平凡函数依赖

- *平凡的函数依赖* (trivial functional dependency)：$X \to Y, Y \subseteq X$
- *非平凡的函数依赖* (non-trivial functional dependency)：$X \to Y, Y \nsubseteq X$

所有关系都满足平凡函数依赖，通常所指的函数依赖一般都是指非平凡函数依赖！

完全和部分函数依赖　假设 X 和 Y 是关系 $R(U)$ 属性集 U 的不同子集，如有 $X \to Y$ 且不存在 X 的真子集 X'，使得 $X' \to Y$，则有 $X \xrightarrow{f} Y$，称 Y *完全函数依赖* (full functional dependency) 于 X；否则，记为 $X \xrightarrow{p} Y$，称 Y *部分函数依赖* (partial functional dependency) 于 X。

传递函数依赖　如果 X, Y 和 Z 分别是关系 $R(U)$ 不同的列属性集合，如果存在 $X \to Y, Y \nrightarrow X, Y \to Z$，则称 Z 对 X *有传递函数依赖* (transitive functional dependency)。

多值依赖　对于关系 R 的属性集 U，设 X 和 Y 是 U 的子集，$Z = U - X - Y$，令 (x, y, z) 表示属性集 (X, Y, Z) 的元组。如果存在 (x, y_1, z_1) 和 (x, y_2, z_2) 时，也存在 (x, y_1, z_2) 和 (x, y_2, z_1)，那么称*多值依赖* (multi-value dependency) $X \to\to Y$ 在关系 R 上成立。
直观上，x 值决定了 y 的取值范围，但 y 的具体取值与 z 值无关。
对于任意的元组 a, b 且 $a[X] = b[X]$，交换元组 a, b 的 Y 属性的值，交换后的元组仍存在于关系 $R(U)$ 中，则 $X \to\to Y$。
如果 Z 为空集，则 $X \to\to Y$ 是*平凡多值依赖*；如果 Z 非空，则 $X \to\to Y$ 是*非平凡多值依赖*。
函数依赖其实是多值依赖的一个特例，函数依赖是单值依赖，即如果 Y 组的值仅有一个，则多值依赖就成为了函数依赖。

连接依赖　如果一个关系可以被分解为若干子关系，并且对这若干子关系通过连接操作后得到原始关系，则*连接依赖* (join dependency) 成立。
多值依赖是连接依赖的特例，即多值依赖可以分解成两个子关系的连接，而连接依赖则可以是多个（包含 2 个）子关系的连接。

3.4.2　关系模式的范式

关系数据库中的关系必须满足一定的要求。满足不同程度要求的为不同范式。
常见的范式：

$$1NF \supset 2NF \supset 3NF \supset BCNF \supset 4NF \supset 5NF$$

一个低一级范式的关系模式，通过模式分解可以转换为若干个高一级范式的关系模式的集合，这种过程就叫规范化 (normalization)。

第一范式 1NF

第一范式 (1NF) 是指关系 R 的每一属性都是不可再分的基本数据项，同一属性中不能有多个值，即关系表中的某个属性不能有多个值或者不能有重复的属性。
在关系数据库中，满足最低要求的范式是第一范式，不满足第一范式的不是关系数据库。如果出现包含多个值的属性，则根据第一范式，需要将该属性进行细分。
满足 1NF 的 SLC 关系，会存在以下问题：

- 插入异常：新入学一批学生，已安排宿舍，但还没有选课，则无法将这批学生插入。
- 删除异常：如果某学生他不再选最后一门课，则删除课程后，整个元组的其他信息（所在系和宿舍信息）也被删除了。
- 修改复杂：如果一个学生选了多门课，则学生信息被重复存储多次。如果该生转系，则需要修改所有关联的信息，造成修改的复杂化。
- 存储冗余

为了解决上述问题，可以通过分解关系 SLC 进行消除部分函数依赖。

第二范式 2NF

第二范式 (2NF) 是指关系 R 首先要满足第一范式，并且每一个非主属性都完全函数依赖于任何一个候选键。

- 主属性：包含在候选键中的属性
- 非主属性：不包含在任何候选键中的属性

满足 2NF 的 SL 关系，依然存在以下问题：

- 插入异常：不能插入一个已分配宿舍但还没有学生的刚成立的系。
- 删除异常：如果该系的所有学生都毕业了，在删除这些学生的时候，也会同时将该系的（系名，宿舍）信息删除。
- 修改复杂：如果某个系的学生统一换宿舍了，需要重复修改。
- 存储冗余：同系的学生宿舍住址信息重复存储；如果添加新的学生信息，还需要重复添加他们的住址信息。

为了解决上述问题，可以通过分解关系 SL 进行消除传递函数依赖。

第三范式 3NF

第三范式 (3NF) 是指在关系 R 满足第二范式，并且不存在非主属性对候选键的传递函数依赖。
将关系模式 SL 分解成两个关系 SL 和 DL 以消除传递函数依赖，使之满足第三范式。
满足 3NF 的 SCT 关系，依然存在以下问题：

- 插入异常：存在函数依赖关系，如果某位教师开设的课程没有学生选修，无法将相关信息插入到数据库中。
- 删除异常：如果选修某课程的学生都毕业了，那么在删除这些学生信息时，相应的课程和教师信息也都同时被删掉了，造成了信息丢失。
- 修改复杂：如果某课程升级改版成另一课程，则需要对所有选修的该课程的元组进行修改。
- 存储冗余：虽然一位教师只教授一门课，但每位选修该教师的课程的学生元组都需要记录该信息，造成了重复存储。

为了解决该问题，可以通过分解关系 SCT 进行消除主属性对键的部分函数依赖。

巴斯-科德范式 BCNF

巴斯-科德范式 (Boyce-Codd, BCNF)：关系 R 在满足第三范式的基础上，任何属性都（非平凡地）完全函数依赖于 R 的候选键。若 $X \to Y$ 且 $Y \nsubseteq X$ 时 X 必是超键。
BCNF 与 3NF 的联系比较紧密，主要体现在以下两个特性上：

1. 如果关系模式 R 满足 BCNF，那么该关系模式 R 必定满足 3NF。
2. 如果关系模式 R 满足 3NF，且只有一个候选键，那么 R 必定满足 BCNF。

如果一个关系模式 R 满足 BCNF，则说明在函数依赖的范畴内，它已经实现了关系模式的彻底分解，达到了最高的规范化程度，消除了插入异常和删除异常。

第四范式 4NF

第四范式 (4NF) 是指关系 R 在满足巴斯-科德范式的基础上，消除了多值依赖。

3.4.3　数据依赖的公理系统

推导函数依赖：给定一组函数依赖，推导其蕴含的其他函数依赖。
分解关系模式：指导正确地分解关系模式，以满足规范化的要求。

Armstrong 公理系统

逻辑蕴含 (logical implication)：对于满足一组函数依赖 F 的关系模式 $R\langle U,F\rangle$，其中任何一个关系 r，如果函数依赖 $X\to Y$ 都成立（即关系 r 中的任意两元组 t,s，若 $t[X]=s[X]$，则 $t[Y]=s[Y]$），则称 F 逻辑蕴含 $X\to Y$，或者称 $X\to Y$ 是 F 的逻辑蕴含。

Armstrong 公理系统：$U=\{A_1,A_2,\dots,A_n\}$ 是 R 中所有属性的集合，F 是 U 上的一组函数依赖，有关系模式 $R\langle U,F\rangle$，对于 $R\langle U,F\rangle$ 有以下推理规则：

1. *自反律*：若 $Y\subseteq X\subseteq U$，则 $X\to Y$ 为 F 所蕴含。
2. *增广律*：若 $X\to Y$ 为 F 所蕴含，且 $Z\subseteq U$，则 $X\cup Z\to Y\cup Z$ 为 F 所蕴含。
3. *传递律*：若 $X\to Y$ 和 $Y\to Z$ 为 F 所蕴含，则 $X\to Z$ 为 F 所蕴含。
4. *合并规则*：若 $X\to Y,Y\to Z$，则 $X\to Y\cup Z$。
5. *分解规则*：若 $X\to Y,Z\subseteq Y$，则 $X\to Z$。
6. *伪传递规则*：若 $X\to Y,Y\cup W\to Z$，则 $X\cup W\to Y\cup Z$。

基于 Armstrong 公理可以得到下述引理：$X\to A_1,A_2,\dots,A_n$ 成立的充分必要条件是 $X\to A_i$ $(i=1,2,\dots,n)$ 成立。

闭包及其计算

函数依赖的闭包　在关系模式 $R\langle U,F\rangle$ 中，为 F 所逻辑蕴含的函数依赖的全体叫作 F 的闭包，记作 F^+。（简而言之，F^+ 包含了 F 中所有的函数依赖。）

属性集的闭包　$U=\{A_1,A_2,\dots,A_n\}$ 是关系模式 R 中所有属性的集合，F 是 U 上的一组函数依赖（即 R 的函数依赖集），$X\subseteq U,Y\subseteq U,X_F^+=\{A\mid X\to A\}$，则 X_F^+ 称为属性集 X 关于函数依赖集 F 的闭包。

Algorithm 1: 求解属性集 $X\subseteq U$ 关于 U 上的函数依赖集 F 的闭包 X_F^+

```
Input: X, F
Output: X_F^+
X^(0) ← X;
i ← 0;
repeat
    B ← {A | (∃V)(∃W)(V → W ∈ F ∧ V ⊆ X^(i) ∧ A ∈ W)};
    X^(i+1) ← B ∪ X^(i);
    if X^(i+1) ≠ X^(i) then
        i ← i + 1;
    end
until X^(i+1) = X^(i) or X^(i) = U;
X_F^+ ← X^(i)
```

属性集闭包的使用 1：判断函数依赖是否成立　$U=\{A_1,A_2,\dots,A_n\}$ 是关系模式 R 中所有属性的集合，F 是 U 上的一组函数依赖，$X\subseteq U,Y\subseteq U,X\to Y$ 能由 F 根据 Armstrong 公理系统推出的充分必要条件是 $Y\subseteq X_F^+$。根据上述引理，可以将判断 $X\to Y$ 能否由 F 根据 Armstrong 公理系统推导出来的问题转化为求解 X_F^+，判定 Y 是否为 X_F^+ 的子集问题。

$$X\to Y\in F^+\Leftrightarrow Y\subseteq X_F^+$$

属性集闭包的使用 2：判断键　对于关系模式 $R\langle U,F\rangle$，其中 $U=\{A_1,A_2,\dots,A_n\}$ 是 R 中所有属性的集合，F 是 U 上的一组函数依赖，如何判断 $X\subseteq U$ 是否是超键？如何判断 X 是否是候选键？

- 如果 $X_F^+=U$，则 X 是超键。
- 如果 $X_F^+=U$ 且 X 的任意子集 Z 不满足 $Z_F^+=U$，则 X 是候选键。

函数依赖集的等价和最小函数依赖集

函数依赖集的冗余　一个函数依赖集 F 存在冗余的函数依赖（可以从其他函数依赖推理出来）

$$\{A\to B,B\to C,A\to C\}$$

一个函数依赖中存在冗余的属性

$$\{A\to B,B\to C,A\to CD\}$$

函数依赖集的等价　若 $F^+=G^+$，则有函数依赖集 F 覆盖 G（或者 G 覆盖 F），也称 F 和 G 等价。（两个函数依赖集等价是指它们的闭包等价。）
$F^+=G^+$ 的充分必要条件是 $F\subseteq G^+$ 且 $G\subseteq F^+$。

最小函数依赖集（最小覆盖）　若函数依赖集 F 满足以下 3 个条件，则称 F 为一个*极小函数依赖集*，亦称为*最小函数依赖集*或*最小覆盖* (canonical cover)。

- F 中任一函数依赖的右部仅含一个属性，即右侧是单个属性；
- F 中不存在这样一个函数依赖 $X\to A$，使得 F 与 $F-\{X\to A\}$ 等价，即无多余的函数依赖；
- F 中不存在这样一个函数依赖 $X\to A$，X 有真子集 Z，使得 $(F-\{X\to A\})\cup\{Z\to A\}$ 与 F 等价，即左部无多余的属性。

每一个函数依赖集 F 均等价于一个最小函数依赖集 F_m。

求最小函数依赖集　使用构造性证明的方式，找出 F 的最小依赖集：

1. 首先依次检查 F 中的各函数依赖 FD_i：使 F 中每一个函数依赖的右部属性单一化。$X\to Y$，若 $Y=A_1,A_2,\dots,A_n(n\geq 2)$，则用 $\{X\to A_j\mid j=1,2,\dots,n\}$ 来取代 $X\to Y$。
2. 然后依次检查 F 中的各函数依赖 $FD_i:X\to A$，设 $X=B_1,b_2,\dots,B_m(m\geq 2)$，逐一检查 $B_i(i=1,2,\dots,m)$，若 $A\in(X-B_i)_F^+$，则用 $X-B_i$ 代替 X。
3. 最后依次检查 F 中的各函数依赖 FD_i：$X\to A$，令 $G=F-\{A\to A\}$，若 $A\in X_G^+$，则从 F 中去掉此依赖。

F 的最小函数依赖集不一定是唯一的，它与对各函数依赖 FD_i 以及 $X\to A$ 中 X 各属性的处理顺序有关。

3.5　数据库规范化设计实现

关系模式的规范化过程是通过对关系模式的分解来实现的。
将满足低一级范式的关系模式分解成若干个满足高一级范式的关系模式的方法并不是唯一的，因此需要规范化算法。
数据库规范化实现是基于数据依赖公理系统，通过关系模式的分解算法，这些分解算法可以保证分解后的关系模式与原关系模式等价。

3.5.1　关系模式的分解

对于关系模式 $R\langle U,F\rangle$，它的一个分解是指 $\rho=\{R_1\langle U_1,F_1\rangle,R_2\langle U_2,F_2\rangle,\dots,R_n\langle U_n,F_n\rangle\}$，并且满足

1. 关系不丢失：$U=\bigcup_{i=1}^{n}U_i$
2. 模式不冗余：不存在 $U_i\subseteq U_j(1\le i,j\le n)$
3. 依赖不丢失：F_i 是 F 在 U_i 上的投影，即 $F_i=\{X\to Y\mid X\to Y\in F^+\wedge X\cup Y\subseteq U_i\}$

关系模式 $R\langle U,F\rangle$ 的分解方式有多种，但都遵循一个原则：分解之后的若干关系模式 ρ 与原模式 R 等价。

关系模式等价分解的概念可以从以下两个角度进行考虑：

1. 数据等价：分解具有无损连接性 (lossless join)，无损连接是指分解后的关系通过自然连接可以恢复分解前的关系；
2. 语义等价：分解要保持函数依赖 (preserve functional dependency)。

因此关系模式等价分解既要保持函数依赖，又要具有无损连接性。（这，废话?）

分解的无损连接性

通过自然连接得到的关系与分解前的关系相比，既不多出信息、又不丢失信息；模式分解过程应可逆。无损的模式分解：关系模式 $R\langle U,F\rangle$ 的一个分解是 $\rho=\{R_1\langle U_1,F_1\rangle,R_2\langle U_2,F_2\rangle,\dots,R_n\langle U_n,F_n\rangle\}$，若 $R_1,R_2,\dots,R_n$ 自然连接的结果相等，则分解 ρ 具有无损连接性，ρ 是关系模式 $R\langle U,F\rangle$ 无损连接分解。

检验无损分解的具体算法

【算法】：检验一个分解是否为无损分解（分解为2个以上关系模式时的判定方法）

输入：关系模式$R<\{A_1,A_2,\dots,A_n\},F>$, $\rho=\{R_1,R_2,\dots,R_m\}$

输出：确定ρ是否为无损分解

算法步骤：

- 步骤（1）构造一个n列m行的表，每一列对应于属性，每一行对应于分解中的一个关系模式。
- 步骤（2）如果$A_j\in R_i$，则在第 i 行第 j 列上置为a_j，否则置为b_{ij}。
- 步骤（3）依次检查F中的每一个函数依赖，并修改表中的元素。对于F中一个函数依赖$X\to Y$，在X的分量上寻找相同的行，然后将这些行的Y分量赋相同符号，如果其中有a_j，则将b_{ij}改为a_j，反之则改为b_{ij} (i为最小行号)。
- 步骤（4）如果有一行为$a_1a_2\dots a_n$，则ρ是无损分解，算法结束，退出；否则进行步骤（5）。
- 步骤（5）如果F中所有的函数依赖都不能再修改表，且没有发现某行变成$a_1a_2\dots a_n$，则ρ不是无损分解，算法结束，退出。

另一种检验无损分解方法（分解为 2 个关系模式）：设 $\rho=\{R_1,R_2\}$ 是关系模式 R 的一个分解，F 是 R 的函数依赖集，那么 ρ 是 R（关于 F）的无损分解的充分必要条件是

$$R_1\cap R_2\to R_1-R_2\in F^+\qquad\text{or}\qquad R_1\cap R_2\to R_2-R_1\in F^+$$

- $R_1\cap R_2$ 指属性的交，返回共同属性集；
- R_1-R_2 指属性的差，返回属于 R_1 但不属于 R_2 的属性集。

分解的保持依赖性

保持函数依赖的模式分解：如果 $F^+ = \left(\bigcup_{i=1}^{n} F_i\right)^+$，则关系模式 R 的一个分解 ρ 保持依赖。

对比保持无损连接的分解和保持函数依赖的分解：

- 若一个分解具有无损连接性，则它能够保证不丢失信息；
- 若一个分解保持了函数依赖，则它可以减轻或者解决各种异常情况；
- 然而，无损分解和保持函数依赖的分解是两个相互独立的标准。
  - 具有无损连接性的分解不一定能够保持函数依赖；
  - 保持函数依赖的分解也不一定是无损分解。

分解模式算法

如何实现关系模式的分解

- 人工分解
  - 优点：对关系模式的语义具有深入理解；
  - 缺点：只能对属性较少的关系模式进行分解。
- 关系模式分解算法
  - 优点：能够对具有很多属性的关系模式进行自动分解；
  - 缺点：缺少对关系模式语义的理解，可能会「过度分解」。

模式分解算法一

【模式分解算法一】分解到3NF，并保持函数依赖的模式分解算法。

输入：关系模式$R < U, F >$

输出：$\rho = \{R_1 < U_1, F_1 >, R_2 < U_2, F_2 >, \ldots, R_m < U_m, F_m >\}$保持函数依赖

算法步骤：

- 步骤（1）令$\rho = \emptyset$，计算F的最小函数依赖集F_m。
- 步骤（2）令$U_0 = \emptyset$，对U中的每个属性A_i，若其不出现在F_m中任一函数依赖的左端和右端，令U_0=$U_0 \cup \{A_i\}$。以U_0为属性集，构造一个新的关系模式R_0，令$\rho = \rho \cup \{R_0\}$，$U = U - U_0$。
- 步骤（3）若$X \to Y \in F_m$，且$XY = U$，则输出$\rho = \rho \cup \{R\}$。即R为3NF，算法终止；否则转步骤(4)。
- 步骤（4）若F_m中存在左端相同的函数依赖$X \to Y_1, X \to Y_2, \ldots, X \to Y_m$，对其进行合并，令$F_m = (F_m - \{X \to Y_1, X \to Y_2, \ldots, X \to Y_m\}) \cup \{X \to (Y_1 \cup Y_2 \cup \cdots \cup Y_m)\}$。 重复步骤（4），直到$F_m$不存在左端相同的函数依赖。
- 步骤（5）对F_m中每个函数依赖$X_i \to Y_i$，令$U_i = X_i \cup Y_i$，构造$R_i(U_i)$，令$\rho = \rho \cup \{R_i\}$。
- 步骤（6）算法终止，输出ρ。

模式分解算法二

【模式分解算法二】分解到3NF，既保持无损连接性又保持函数依赖的模式分解。

输入：关系模式$R<U,F>$

输出：$\rho=\{R_1<U_1,F_1>,R_2<U_2,F_2>,\ldots,R_m<U_m,F_m>\}$保持无损连接性和函数依赖。

算法步骤：

- 步骤（1）基于算法一，求出保持函数依赖的分解$\rho=\{R_1<U_1,F_1>,R_2<U_2,F_2>,\ldots,R_k<U_k,F_k>\}$。
- 步骤（2）若有某个U_i包含任意一个候选键，则输出ρ；否则，选择$R<U,F>$的任一候选键X，组成新的关系模式$\boldsymbol{R_{k+1}<X,F_X>}$，输出$\rho\cup\{\boldsymbol{R_{k+1}}\}$。

模式分解算法三

【模式分解算法三】分解到BCNF，具有无损连接性的模式分解算法。

迭代

输入：关系模式$R<U,F>$

输出：$\rho=\{R_1<U_1,F_1>,R_2<U_2,F_2>,\ldots,R_m<U_m,F_m>\}$保持无损连接性。

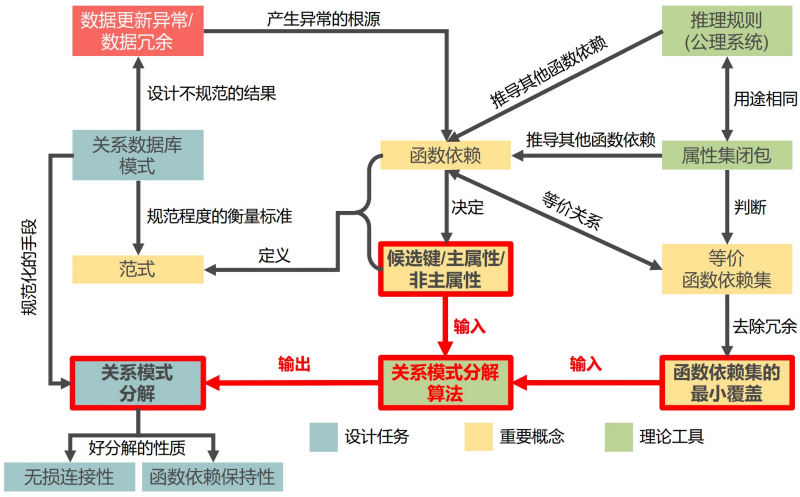
算法步骤：

- 步骤（1）令$\rho=\{R<U,F>\}$。
- 步骤（2）若ρ中各个关系模式都满足BCNF，转步骤（4），否则步骤（3）。
- 步骤（3）任取ρ中不满足BCNF的模式$R_i(U_i)$，F在U_i上的投影为F_i，由于$R_i(U_i)$不满足BCNF，则必定存在函数依赖$X\rightarrow Y\in F_i^+$，其中X不是$R_i(U_i)$的候选键，且$Y\not\subseteq X$。分别以属性集$U_i-\{Y\}$和$X\cup Y$构造模式R_i'和R_i''，令$\boldsymbol{\rho=(\rho-\{R_i\})\cup\{R_i',R_i''\}}$，转（2）。
- 步骤（4）算法终止，输出ρ（由于U中的属性个数有限，该算法必然终止）。

如果要求分解具有无损连接性，则模式分解一定可以达到 4NF。

可以基于模式分析算法三转换为 BCNF 的无损连接分解后，针对其中不满足 4NF 的关系模式，可以再进一步分解，使得每一关系模式都满足 4NF，此时分解的结果都是满足 4NF 的无损连接分解。

知识点总结



第四章　Web 开发与数据库

内容不重要，反正也不考，看个乐呵就可以了。

- Spring Boot
- HTTP 请求
- 数据库编程接口 (API)
- ODBC (Open Database Connectivity) 与 JDBC (Java Database Connectivity)

连接管理

```
Connection connection = null;
try {
    // 加载数据库驱动
    Class.forName("com.mysql.cj.jdbc.Driver");
    // 创建数据库连接
    String url = "jdbc:mysql://localhost:3306/your_database";
    String user = "username";
    String password = "password";
    connection = DriverManager.getConnection(url, user, password);
    // 在这里可以执行数据库操作，比如执行 SQL 查询
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    // 释放数据库连接
    if (connection != null) {
        try {
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

语句管理

```
Statement statement = null;
try {
    statement = connection.createStatement();
    String sql = "INSERT INTO students (id, name) VALUES (1, 'Alice')";
    statement.executeUpdate(sql);
} catch (SQLException e) {
    e.printStackTrace();
```

```
} finally {
    if (statement != null) {
        statement.close();
    }
}
```

结果集管理

```
Statement statement = null;
ResultSet resultSet = null;
try {
    statement = connection.createStatement();
    String sql = "SELECT id, name FROM students";
    resultSet = statement.executeQuery(sql);
    while (resultSet.next()) {
        int id = resultSet.getInt("id");
        String name = resultSet.getString("name");
        System.out.println("ID: " + id + ", Name: " + name);
    }
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    if (resultSet != null) {
        resultSet.close();
    }
    if (statement != null) {
        statement.close();
    }
}
```

参数绑定

JDBC (Java Database Connectivity) 的参数绑定机制是一个重要特性，它提供了一种安全且高效的方式来设置 SQL 语句中的参数。了解这一机制对于编写高效且安全的数据库交互代码非常重要。

```
String sql = "SELECT * FROM users WHERE name = ? AND age = ?";
PreparedStatement statement = connection.prepareStatement(sql);
statement.setString(1, "Alice");
statement.setInt(2, 30);
```

```
try (Connection connection = DriverManager.getConnection(url, user, password)) {
    String sql = "INSERT INTO your_table (column1, column2) VALUES (?, ?)";
    try (PreparedStatement statement = connection.prepareStatement(sql)) {
        statement.setString(1, "value1");
        statement.setString(2, "value2");
        int rowsInserted = statement.executeUpdate();
        if (rowsInserted > 0) {
            System.out.println("A new row was inserted successfully!");
        }
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

第五章　并发控制

5.1　回顾：事务控制

事务（ACID）

- 原子性：事务单元全部成功或者全部失败
- 一致性：一个正确（一致）的状态转移到另一个正确的状态
- 隔离性：互不干扰，多个事务在并发执行的过程中所得到的结果，和串行执行得到的结果是一致
- 持久性：永久保持，执行结果不会丢失

5.2　并发事务与隔离控制

5.2.1　并发事务的问题

- 脏读：事务之间可以看到未提交的数据；
- 不可重复读：事务之间可以看到已提交的数据，但不能看到未提交的数据；
- 幻读：事务之间可以看到已提交的数据，但不能看到未提交的数据；
- 写冲突：事务之间可以看到已提交的数据，但不能看到未提交的数据。

不可重复读：面对更新操作；幻读：面对插入和删除操作。

5.2.2　隔离机制

- 读未提交：最低的隔离级别，事务可以读取其他未提交事务的更改。它允许「脏读」，即一个事务可以读取另一个事务未提交的数据。
- 读已提交：一个事务只能读取其他事务已提交的更改。这避免了脏读的问题，但仍然可能出现「不可重复读」。
- 可重复读：事务可以确保在整个事务执行过程中多次读取同一数据，并且每次读取的结果都相同。它防止了不可重复读，但可能仍然允许「幻读」。
- 串行化：事务以完全串行的方式执行，从而防止了脏读、不可重复读和幻读。这提供了最强的事务隔离性，但可能会显著降低系统的并发性能。

	隔离级别	脏读	不可重复读	幻读
性能	**读未提交**	✓	✓	✓
↑	**读已提交**	✗	✓	✓
↓	**可重复读**	✗	✗	✓
数据一致性	**可串行化**	✗	✗	✗

5.3　事务、数据项、锁

观察：并发错误来自于对同一数据项的同时读写。

5.3.1　共享锁（读锁，S 锁）

如果某事务在数据项上加了共享锁，则该事务只能读取但不能修改；
其他事务在不对加锁的情况下无法读写，在加共享锁的情况下可以读取。

5.3.2　互斥锁（写锁，X 锁）

如果某事务在数据项上加了互斥锁，则该事务能够读写；
其他事务在不对加锁的情况下无法读写，也无法对其加共享锁或者互斥锁。

相容性	共享锁	互斥锁
共享锁	✓	✗
互斥锁	✗	✗

5.3.3　锁的使用方式

- 事务访问数据项前应先加锁
  - 只进行读取操作则申请共享锁
  - 要进行读写操作则申请互斥锁
- 授予锁的条件
  - 数据项未被加锁，则可允许当前加锁请求
  - 数据项已被加锁，根据相容性处理
    - 可授予相容的共享锁
    - 拒绝不相容的加锁请求，申请加锁事务应当等待当前持有锁的事务释放锁
- 事务释放锁的两种方式
  - 在执行期间释放锁
  - 在终止时释放锁，包括事务撤销或者完成提交

事务开始	事务开始
`S-Lock(d)`	`X-Lock(d)`
读取余额 d	等待
`Unlock(d)`	
等待	更新余额 d
	`Unlock(d)`
`X-Lock(d)`	提交
更新余额 d	
`Unlock(d)`	
提交	

5.3.4　锁带来的问题 1：饿死

锁导致等待，等待可以很久很久 ⇒ 饿死
饿死避免机制：先来先服务策略……
锁的实现数据结构：哈希表。

二阶段锁 (two-phase locking, 2PL) 协议

- 加锁阶段 (locking phase)：
  - 在这个阶段，事务可以获取所需的任何锁。
  - 事务在执行过程中，每当需要访问一个新的数据项时，就必须首先为这个数据项加锁。
  - 这个阶段持续到事务获取了它所需的所有锁。
- 解锁阶段 (unlocking phase)：
  - 一旦事务开始释放锁，它就不能再获取任何新的锁。
  - 这个阶段开始于事务释放其第一个锁。
  - 在解锁阶段，事务逐渐释放所有锁，直到完成所有操作。

5.3.5　锁带来的问题 2：死锁

死锁预防

关键思路：不允许等待。

		T_n 申请锁	T_o 申请锁
Wait-Die	T_n 持有锁		T_o 等待
	T_o 持有锁	T_n 被撤销	
Wound-Wait	T_n 持有锁		T_n 被撤销
	T_o 持有锁	T_n 等待	

死锁检测与恢复

检测事务依赖环（会产生额外的资源开销）。
当检测到环的时候，撤销一个事务作为牺牲者，根据：

- 事务开始运行的时间戳
- 事务已经执行的 SQL 语句数量
- 事务已经加锁的数据项的数量

设锁超时重启

设置事务最长等待时间。

- 如果申请加锁的事务在该时间结束时仍未获得锁，即事务超时，那么此时将该事务回滚并重启。
- 当事务之间确实存在死锁时，该方法会让死锁中的一个或者多个事务回滚并重启，而其他事务继续执行，解决了死锁问题。

适用场景：事务执行时间都比较短；事务长时间的等待大多都是死锁造成的。

5.3.6　多粒度锁

数据项的粒度：属性、元组、表、整个数据库。
锁的粒度：基于数据项的粒度。

- 粒度越大，数据项越少，管理锁的开销越小；事务之间发生冲突的可能性越大，并发度越低；
- 粒度越小，系统开销越大，并发度越高。

多粒度锁

- 数据库中的所有对象按照大小的不同构成了一棵*多粒度树*
- 多粒度树中的每一个节点可以独立地加锁
- 对某个节点加锁表示对该节点的后代节点加同类型的锁

数据项的两种加锁状态

- *显示锁*：该数据项被直接加上的锁
- *隐式锁*：该数据项没有被直接加锁，由于其祖先节点被加锁而变为加锁状态

意向锁　提供了一种对某个数据项读写的「意向」，相比于普通锁有更高的并发性。
使用时，对多粒度树中任一节点加锁时，需要先对其所有祖先点加意向锁。

- *意向共享锁*：如果要对某个数据项加共享锁，先对其祖先点加意向共享锁（记为 IS 锁）
- *意向互斥锁*：如果要对某个数据项加互斥锁，先对其祖先点加意向互斥锁（记为 IX 锁）
- *共享意向互斥锁*：记为 SIX 锁，表示共享锁 + IX 锁。对某数据项加 SIX 锁，表示在对该数据项显式加共享锁的同时加 IX 锁

相容性	S	X	IS	IX	SIX
S	✓	✗	✓	✗	✗
X	✗	✗	✗	✗	✗
IS	✓	✗	✓	✓	✓
IX	✗	✗	✓	✓	✗
SIX	✗	✗	✓	✗	✗

5.3.7　谓词锁

作用于满足特定条件的所有对象，例如 `age > 17` 的谓词锁作用于所有满足该条件的元组。

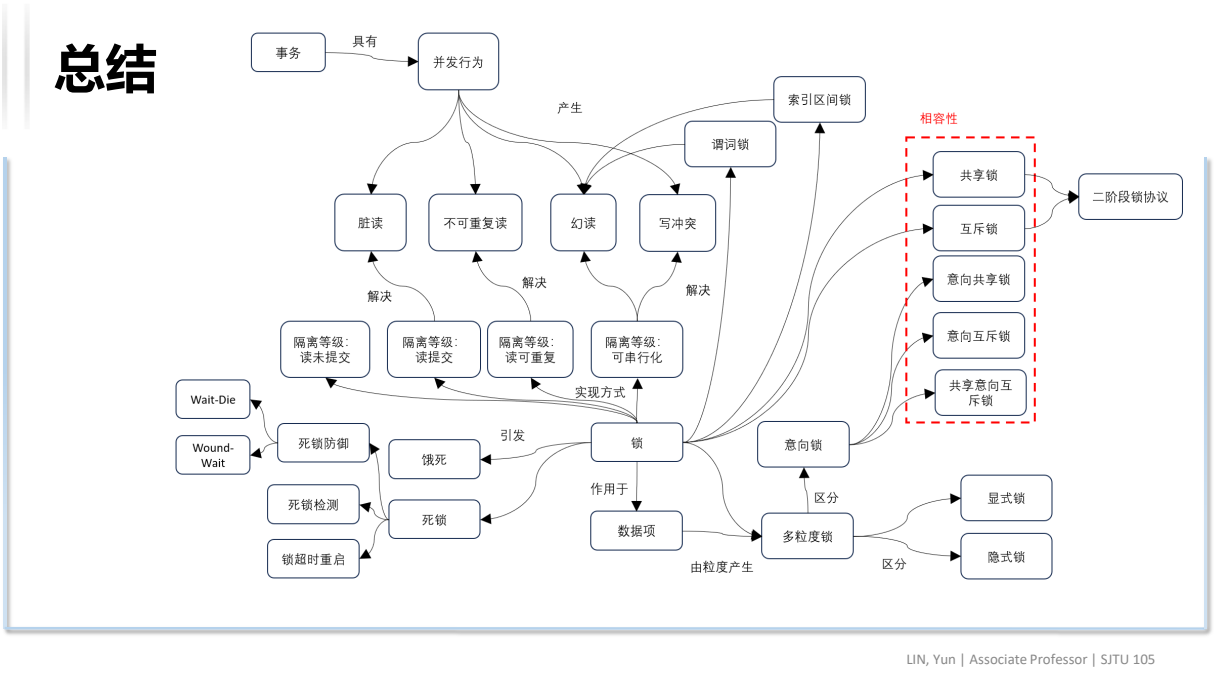
- 读取 (`select`)：如果事务 T_1 想读取满足某谓词的对象，则该事务需要获得相应的谓词锁。若此时事务 T_2 正在持有满足该谓词的某个对象的互斥锁，则 T_1 需要等到 T_2 释放该互斥锁，才能加谓词锁，并继续执行查询。事务 T_1 提交或回滚时释放持有的所有谓词锁。
- 插入、更新和删除 (`insert`, `update`, and `delete`)：如果事务 T_1 想要插入、更新或删除一些对象，需要检查这些对象的旧值和新值是否与现有的谓词锁匹配。如果与某些谓词锁匹配，则需要等待这些谓词锁释放，之后事务 T_1 才能继续执行。

5.3.8　索引区间锁

一种在数据库索引上施加的锁，用于锁定一个特定的索引键值范围。这种锁不仅锁定实际存在的索引键值，还包括这个范围内可能插入的未来键值。

性能分析：

- 索引区间锁的检查与索引更新同步，开销较小；
- 索引区间锁扩大了锁的范围，导致更多锁冲突和等待；
- 是一种折衷的方法。



5.4　调度 (Schedule)

并发事务的并发过程中所有事务中操作的执行顺序。

5.4.1　可串行化调度 (serializable schedule)

给定一个并发调度 S，存在一个串行调度 S'，在任何数据库状态下，按照调度 S 和调度 S' 执行后所产生的结果都是相同的。

如果一个调度是可串行化的，我们理论上就不需要锁机制，同时也就不需要锁带来的相应问题了。

如果我们能想办法控制事务调度变成一个可串行化的调度，那么同时也就避免了很多冲突问题。

判定一个调度是可串行化调度的算法复杂度是 $n! \approx \sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^n$。（还是一样，看个乐呵）

- 冲突可串行化：判断操作序列之间是否冲突
- 视图可串行化：判断事务可见结果是否一致
- 终态可串行化：判读最终执行结果的状态

交换：一个调度中时间上不同事物上相邻两个操作的顺序置换。

当交换不会影响两个调度的一致性时，我们称该交换为等价操作，得到的调度也是等价的。

- 等价操作：对相同或不同数据项的读操作；对不同数据项的读写操作
- 非等价操作：对相同数据项的读写操作

5.4.2　冲突可串行化调度

给定一个调度 S，如果我们可以通过一系列等价操作将 S 变换称一个串行调度 S'，那么我们称 S 为一个冲突可串行化调度。
调度优先图无环的调度即冲突可串行化的调度。

5.4.3　视图可串行化调度

每个事务读取（看到）数据的结果必须与它在某个序列化执行顺序中读取的结果相同。
如果存在一个串行调度 S' 与目标视图等价，那么对任意数据项 X：

- S 与 S' 需要读取相同的 X 的初始值
- S 与 S' 需要读取相同的 X 的更新值
- S 与 S' 需要最后写入相同的 X 的更新值

改进优先图来检测视图可串行化调度：

- 引入 T_0，作为数据项的写操作；引入 T_4，作为数据项的读操作
- 如果 T_j 读取了 T_i 的变量，那么从 T_i 指向 T_j，意味着 T_i 需要先于 T_j 发生
- 由于 T_1 是初始化数据项的事务，所以它需要优先执行
- 由于 T_3 是最后写入数据项的事务，所以它需要最后执行
- 对于任一事务 T 中对数据项的写操作，我们寻找图中其它事务之间的一条数据流边（比如 $T_2 \to T_3$）
  - 可能性 1：将该事务放到数据流前
  - 可能性 2：将该事务作放到数据流后

  只要存在一种优先图无环，则视图可串行化。

5.5　乐观并发控制技术

假设：多个事务同时操作同一数据的几率很低。

- 读取阶段：这个阶段不会锁定数据，从而允许其他事务并行访问相同的数据。
- 验证阶段：当事务准备提交更改时，先检查在读取阶段期间是否有其他事务修改了它所读取的数据。
- 提交/回滚阶段：如果验证阶段发现没有冲突（即自事务开始以来数据没有被其他事务更改），则事务将其更改提交到数据库。如果发现冲突，事务可能会回滚并可能重试。

时间戳排序协议：每个以事务有一个唯一的时间戳。读写规则：

- 每个数据项上都记录一个最新的读写记录时间
- 通过对比事务的时间戳和所要读写的数据项上的时间戳来决定是否执行或者回滚事务
  - 事务读：T 只能读 T 之前的事务写的数据
  - 事务写：T 只能写 T 之前读写的数据

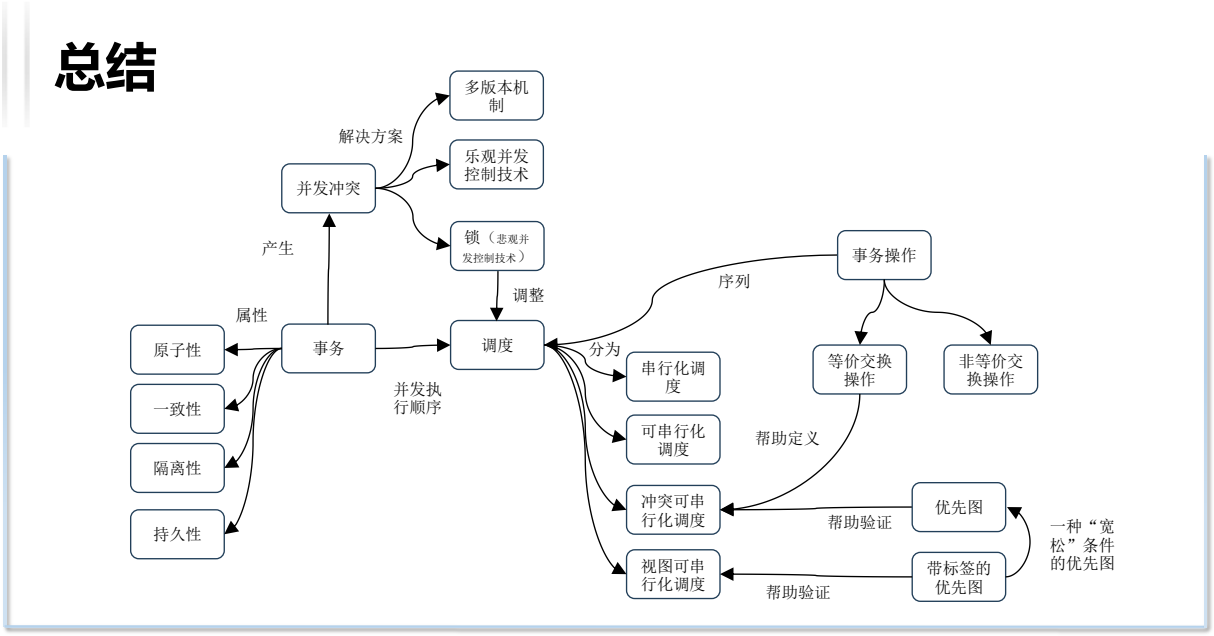
5.6　多版本机制

对于一个数据项，保存多个物理版本，供不同的事务使用。
前面介绍的方法通过时间复用实现并发控制，多版本机制通过空间复用实现高效并发控制。
基本原则：

- 事务对数据项进行写操作时，产生该数据项的一个新版本
- 事务对数据项进行读操作时，读取事务开始时该数据项的最新版本

优点：

- 事务的读操作无需等待其他事务的写操作
- 事务的写操作无需等待其他事务的读操作



第六章　数据库恢复

6.1　回顾：事务及其特性

6.1.1　事物的特性

数据库事务包含 ACID 四大特性：

- 原子性 (atomicity)：不可分割，事务单元全部成功或者全部失败
- 一致性 (consistency)：正确一致，一个正确（一致）的状态转移到另一个正确的状态
- 隔离性 (isolation)：互不干扰，多个事务在并发执行的过程中所得到的结果，和串行执行得到的结果是一致
- 持久性 (durability)：永久保持，执行结果不会丢失

并发控制实现事务的一致性和隔离性，数据库恢复实现事务的原子性和持久性。

原子性

事务的原子性规定了事务是一个不可再分的基本操作单元，「事务全部成功或者失败」就是事务原子性的要求。

持久性

持久性是指，一旦事务完成提交，即使数据库发生故障，该事务的执行结果也不会丢失，可以被正确地恢复，仍然对后续事务可见。

6.2　数据库故障

6.2.1　故障的类型

- 事务故障（transaction failures）：数据库事务因为资源冲突或者死锁等原因导致执行失败，必须终止。
- 系统故障（system failures）：数据库自身或操作系统的故障导致数据库进程意外退出。
- 磁盘故障（存储介质故障 storage media failures）：数据因为磁盘（其他非易失性存储）损坏导致无法被读取。
- 自然灾害：自然灾害对数据库系统所在的环境造成了彻底性破坏。

6.2.2　数据库故障与恢复机制对应关系

问题类型	出现频率	对事务的影响	解决方法
事务故障	较高	原子性	故障恢复机制
系统故障能重启	中等	原子性/持久性	
磁盘故障	低	持久性	数据多备份
系统故障不能重启	低	持久性	一主多备
自然灾害	极低	持久性	异地多机恢复

数据库备份

问题：磁盘数据丢失?
解决方法：单机数据多副本，如 RAID 技术、Paxos 技术
问题：机器不能重启?
解决方案：备份数据库或数据库 + 日志

异地容灾

在同城、异地建立数据备份中心，比主备模式具备更高的安全性。
挑战：数据同步代价高、多节点带来的数据一致性问题。

6.3　缓冲池策略

6.3.1　系统故障恢复

系统故障诱因：数据库进程被操作系统中止、管理员错误操作、软件故障、死锁……
故障系统重启后，内存 RAM 数据丢失，磁盘 Disk 数据不丢失。
DBMS 在进行故障恢复时会执行两种操作：

- 撤销 (Undo)：撤销未完成事务对数据库的修改；
- 重做 (Redo)：重做已提交事务对数据库的修改。

6.3.2　缓冲池策略

STEAL	允许将未提交事务所做的修改写到磁盘并覆盖现有数据
NO-STEAL	不允许将未提交事务所做的修改写到磁盘并覆盖现有数据
FORCE	强制事务在提交前必须将其所做的修改全部写回磁盘
NO-FORCE	不强制事务在提交前必须将其所做的修改全部写回磁盘

	I/O 效率低	I/O 效率高
缓冲池效率低	NO-STEAL + FORCE	NO-STEAL + NO-FORCE
缓冲池效率高	STEAL + FORCE	STEAL + NO-FORCE

No-steal 缺点：事务执行过程中不能刷盘，占较大的缓冲区空间，不利于并发；
Force 缺点：事务完成必须刷脏，导致大量 I/O 读写；
几乎所有 DBMS 都采用 STEAL + NO-FORCE。

6.3.3　系统崩溃恢复示例

假设 T_1 在系统崩溃前已提交，T_2 在系统崩溃前未结束。

- NO-STEAL + FORCE
  - T_1：已刷脏，不影响持久性；
  - T_2：未刷脏，不影响原子性。
- NO-STEAL + NO-FORCE
  - T_1：未刷脏，影响持久性，需要重做；
  - T_2：未刷脏，不影响原子性。
- STEAL + FORCE
  - T_1：已刷脏，不影响持久性；
  - T_2：已刷脏，则影响原子性，需要撤销。
- STEAL + NO-FORCE
  - T_1：未刷脏，影响持久性，需要重做；
  - T_2：已刷脏，则影响原子性，需要撤销。

6.3.4　数据库恢复算法分类

	FORCE	NO-FORCE
NO-STEAL	✗ redo log | ✗ undo log	✓ redo log | ✗ undo log
STEAL	✗ redo log | ✓ undo log	✓ redo log | ✓ undo log

6.4　数据库日志

日志文件 (log file) 是用来记录事务对数据库的更新操作的文件，数据库通过对日志分析进行事务的回滚和重做，实现原子性和持久性。
日志日志记录 (log record) 的序列，顺序写入磁盘，且不会被修改。

6.4.1　Undo 回滚日志

当事务 T 修改数据项 X 产生回滚日志，用于实现事务回滚。
格式：$\langle T, X, v_{\text{old}} \rangle$

- T：事物的唯一标识符
- X：数据项
- v_{old}：数据项修改以前的值

注意：一般还包含一个日志序号 log sequential number (LSN)。

6.4.2　Redo 重做日志

当事务 T 修改数据项 X 产生重做日志，用于实现事务重做。
格式：$\langle T, X, v_{\text{new}} \rangle$

- T：事物的唯一标识符
- X：数据项
- v_{new}：数据项修改以后的值

注意：一般还包含一个日志序号 log sequential number (LSN)。

6.4.3　预写日志 WAL

日志只有在持久存储中才能发挥作用。
数据库日志需要满足*预写日志条件* (Write Ahead Logging, WAL)，即日志必须比数据更早的写入磁盘。
日志写回磁盘的顺序必须和日志生成的时间相一致。
只有 $\langle T, \text{commit} \rangle$ 写入磁盘后，事务才算提交。

基于 WAL 的故障恢复

第一部分：事务正常执行时的行为
- 记录日志
- 按照缓冲池策略将修改过的对象写到磁盘

第二部分：故障恢复时的行为
- 根据日志和缓冲池策略，对事务进行 undo 或 redo

6.5　故障恢复机制

6.5.1　事务的分类

根据日志将事务分为 3 类：
- *已提交事务*：既有 $\langle T, \text{start} \rangle$ 又有 $\langle T, \text{commit} \rangle$；
- *不完整事务*：只有 $\langle T, \text{start} \rangle$ 没有 $\langle T, \text{commit} \rangle$；
- *已中止事务*：既有 $\langle T, \text{start} \rangle$ 又有 $\langle T, \text{abort} \rangle$。

在事务正常执行和故障恢复过程中，如果 T 所做的修改已经全部撤销，则将日志记录 $\langle T, \text{abort} \rangle$ 写到日志。
已中止事务相当于从未执行过，因此不需要 undo，更不需要 redo。

故障恢复时的行为

- *已提交事务*：如果一个已提交事务的修改已全部写入磁盘，则无需 redo，否则需要 redo；
- *不完整事务*：如果一个不完整事务的任何修改都未写入磁盘，则无需 undo，否则需要 undo；

缓冲池策略决定了上述行为。

6.5.2　影子拷贝方法

适用于 NO-STEAL + FORCE (✗ redo log | ✗ undo log)。
事务修改在拷贝数据库（或者影子拷贝页面）上，提交事务时，切换数据库指针。
优化：影子页面，仅拷贝修改的页面；
缺点：效率低，难以支持事务并发。

6.5.3　基于 undo 日志的恢复

适用于 WAL + STEAL + FORCE (✗ redo log | ✓ undo log)。
不需要处理事务重做，需要依靠 undo 日志来回滚事务。

故障恢复时，需要找到所有未提交（也称不完整）的事务，回滚这些未提交的事务并写入该事务中止的日志。

基于 undo 日志的事务正常执行时的行为

- 开始事务：向日志中写入事务开始记录 $\langle T, \text{start}\rangle$
- 修改数据项 X：向日志中写入 undo 日志记录 $\langle T, X, v_{\text{old}}\rangle$（修改过的脏页允许写入磁盘，但必须先将脏页对应的 undo 日志写入磁盘）
- 提交事务：将 T 关联的脏页及 undo 日志写入磁盘，写入 $\langle T, \text{commit}\rangle$
- 中止事务：将 T 关联的脏页及 undo 日志写入磁盘，写入 $\langle T, \text{abort}\rangle$

基于 undo 日志的恢复流程

识别崩溃发生时刻数据库中事务的状态，恢复系统会扫描整个日志并且条件识别出需要回滚的事务。
已提交的事务：FORCE 不需要被回滚。
不完整的事务：STEAL 需要被回滚。
回滚所有不完整的事务。
从后（最后一条记录）向前（第一条记录）逆序扫描整个 undo 日志，根据每条日志记录的类型执行相应的动作。

- $\langle T, \text{commit}\rangle$：将 T 记录为已提交事务（无需 undo）
- $\langle T, \text{abort}\rangle$：将 T 记录为已中止事务（无需 undo）
- $\langle T, X, v_{\text{old}}\rangle$：如果 T 是不完整事务，则将磁盘上的 X 恢复为 v_{old}
- $\langle T, \text{start}\rangle$：如果 T 是不完整事务，表示 T 恢复完毕，向日志中写入 $\langle T, \text{abort}\rangle$（今后故障恢复时无需再 undo）

6.5.4　基于 undo 日志的恢复

适用于 WAL + NO-STEAL + NO-FORCE (✓ redo log | ✗ undo log)。
不需要处理事务回滚，需要依靠 redo 日志来重做事务。
故障恢复时，需要找到所有已提交的事务，重做这些已提交的事务并写入该事务结束的日志。

基于 undo 日志的事务正常执行时的行为

- 开始事务：向日志中写入事务开始记录 $\langle T, \text{start}\rangle$
- 修改数据项 X：向日志中写入 redo 日志记录 $\langle T, X, v_{\text{new}}\rangle$（未提交事务修改过的脏页不允许写入磁盘）
- 提交事务：将 T 关联的 redo 日志写入磁盘，写入 $\langle T, \text{commit}\rangle$，$T$ 关联的脏页（且该脏页无相关未提交事务）允许写入磁盘
- 中止事务：使缓冲区内该事务修改的页面失效 (invalidate)，写入 $\langle T, \text{abort}\rangle$

基于 undo 日志的恢复流程

识别崩溃发生时刻数据库中事务的状态，恢复系统会扫描整个日志并且条件识别出需要重做的事务（即已提交的事务）。
已提交的事务：NO-FORCE 已提交事务所做的修改可能尚未全部写入磁盘。
不完整的事务：NO-STEAL 不完整事务所做的任何修改都未写入磁盘。

回滚所有不完整的事务。
从前（第一条记录）向后（最后一条记录）顺序扫描整个 redo 日志两遍。

- 第 1 遍扫描：记录已提交事务和已中止事务
  - $\langle T, \text{commit}\rangle$：将 T 记录为已提交事务（需要 redo）
  - $\langle T, \text{abort}\rangle$：将 T 记录为已中止事务（无需 redo）
- 第 2 遍扫描：根据每条日志记录的类型执行相应的动作
  - $\langle T, X, v_{\text{new}}\rangle$：如果 T 是已提交事务，则将磁盘上的 X 覆写为 v_{new}
  - $\langle T, \text{start}\rangle$：如果 T 是不完整事务，向日志中写入 $\langle T, \text{abort}\rangle$

6.5.5　基于 undo/redo 日志的恢复

适用于 WAL + STEAL + NO-FORCE (✓ redo log | ✓ undo log)。
需要依靠 undo 日志处理事务回滚，需要依靠 redo 日志处理事务重做。
故障恢复时，需要所有需要重做以及需要回滚的事务（分析阶段），重做这些已提交的事务（重做阶段）并回滚不完整的事务（撤销阶段）。

基于 undo/redo 日志的事务正常执行时的行为

- 开始事务：向日志中写入事务开始记录 $\langle T, \text{start}\rangle$
- 修改数据项 X：向日志中写入 redo 日志记录 $\langle T, X, v_{\text{old}}, v_{\text{new}}\rangle$（修改过的脏页允许刷盘）
- 提交事务：写入事务提交记录 $\langle T, \text{commit}\rangle$，并且将日志刷盘，页面可不刷盘
- 中止事务：写入事务中止记录 $\langle T, \text{abort}\rangle$，并且将日志刷盘，页面可不刷盘

基于 undo/redo 日志的恢复流程

识别崩溃发生时刻数据库中事务的状态，恢复系统会扫描整个日志并且条件识别出需要重做的事务以及需要回滚的事务。
已提交的事务：NO-FORCE 已提交事务所做的修改可能尚未全部写入磁盘。
不完整的事务：STEAL 不完整事务所做的一部分修改可能已经写入磁盘。
回滚所有不完整的事务。

undo/redo 恢复的分析阶段　系统从日志起始位置开始扫描整个日志，找出需要重做和需要回滚的事务。

- 在扫描过程中出现 $\langle T, \text{start}\rangle$ 的日志记录而没有 $\langle T, \text{commit}\rangle$ 或 $\langle T, \text{abort}\rangle$，那么该事务在数据库崩溃的时刻是不完整的，需要被回滚（标注回滚）
- 在扫描过程中出现了 $\langle T, \text{commit}\rangle$，那么事务已经提交，需要被恢复子系统重做（标注重做）

undo/redo 恢复的重做阶段　系统按时间顺序正向扫描日志，如果出现了一条标注重做的日志记录，系统便重做它。
由于系统重做了所有的日志更新记录，这个过程和数据库的执行历史是相同的，因此该过程也被称作*重放历史* (repeating history)。

undo/redo 恢复的撤销阶段　从日志末尾反向扫描整个日志，如果出现了一条标注撤销的日志记录，那么系统会撤销它（包括修改缓冲池）。
一旦事务撤销完成（即扫描中遇到了 $\langle T, \text{start}\rangle$），数据库会自动写入 $\langle T, \text{abort}\rangle$，代表该事务已经回滚完成。

6.5.6　检查点机制

数据库的日志会随着事务的执行不断变长，这会使恢复过程中系统需要扫描并处理更多的日志，恢复时间也相应地变长，需要压缩日志大小来降低恢复的时间。
数据库设计了一种**检查点**(checkpoint)机制，检查点定义了一个脏页刷盘的时刻，要求检查点之前的日志记录对应的缓冲区数据页面修改已经刷新到磁盘。

全量检查点

- 停止接受新的事务或修改请求，确保没有新的脏数据产生。
- 将当前所有未写入磁盘的脏数据页写入磁盘，更新对应的数据文件。（全量刷脏）
- 记录检查点。
- 恢复接受新的事务或修改请求，继续正常的数据库操作。

涉及检查点的故障恢复

在有检查点的情况下，系统恢复首先定位到检查点时候的日志，并分为以下三种情况处理日志。

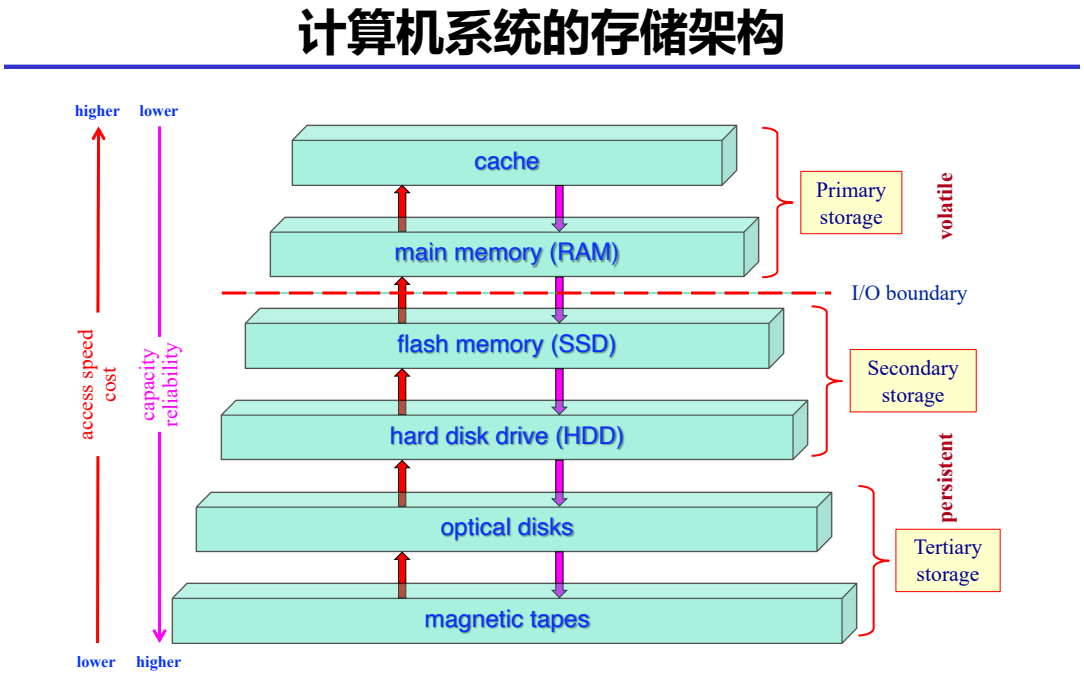
1. 在检查点之前完成（commit/abort）的事务不需要处理；
2. 在检查点之后完成（commit/abort）的事务需要重做；
3. 所有未完成的事务（不含 commit/abort）需要回滚。

数据库管理系统往往会定时地执行检查点操作，用户也可以在终端手动输入 `checkpoint` 令数据库系统执行检查点操作。
当数据库开始检查点操作后，它会持续将缓冲区的脏页写入磁盘。

第七章　数据库物理存储和索引

7.1　数据库物理存储

7.1.1　计算机系统的存储架构



一个典型的存储架构包括：

- 主存储器 (primary storage)：按字节寻址 byte-addressable
  - 寄存器 (register)
  - 高速缓存 (cache)
  - 内存 (main memory)
- 二级存储器 (secondary storage)：按块寻址 block-addressable
  - 磁盘 (magnetic disk)/机械硬盘 (hard disk drive, HDD)
  - 闪存 (flash memory)/固态硬盘 (solid state drive, SSD)
- 三级存储器 (tertiary storage)：按块寻址 block-addressable
  - 磁带 (magnetic tape)
  - 光盘 (optical dick)
  - 网络存储 (network storage)

访存时间

访存时间	存储介质
0.5	L1 cache
7	L2 cache
100	DRAM
150,000	SSD
10,000,000	HDD
30,000,000	网络存储
1,000,000,000	磁带

为什么不把数据都存在内存里?

- 内存容量小
- 内存价格高
- 内存介质易失

存储层次之间的数据传输

两种局部性:

- 时间局部性 (temporal locality)
- 空间局部性 (spatial locality)

磁盘

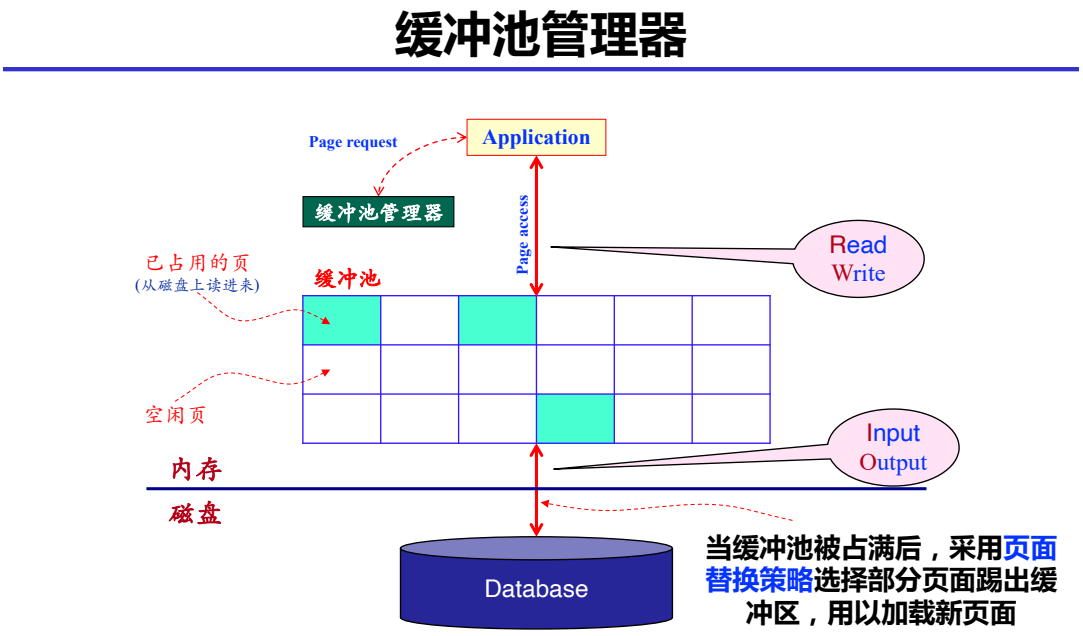
最常见的二级储存介质。
组织结构：*盘面* platter，*磁道* track，*扇区* sector（最小读写单位）。
磁盘*块* block/*页* page：连续的若干个扇区，大小通常是 512B-16KB。
示例：一个磁盘有 8 个圆盘，16 个盘面，没个盘面有 2^{16} 和磁道，每个磁道平均有 2^8 个扇区，每个扇区有 2^12 个字节，那么磁盘的容量为 $2^4 \times 2^{16} \times 2^8 \times 2^{12} = 2^{40}$ 字节。
一个磁盘页面 (page/block) 的访问时间包括:

- *寻道时间*：移动磁头至磁道
- *旋转时间*：页面旋转
- *传输时间*：数据传输

寻道时间和旋转时间最长，顺序访问比随机访问更快。

存储访问

为了避免磁盘访问，一个直观的做法是：尽可能多的把页面放在内存中。
缓冲池 (buffer pool)：内存中用来缓存磁盘页面的区域。
缓冲池管理器 (buffer manager)：管理内存和磁盘之间的数据传输。
缓冲区管理器的目标：**减少磁盘总访问次数（缩短磁盘总访问时间）**
注意：数据在被操作（读/写）之前，须先加载到内存；缓冲区大小受到内存容量的限制。



操作系统中常用的内存页面替换策略是最近最少使用策略 least recently used (LRU)。
LRU 在一些访问模式下性能不佳：两个表 *r* 和 *s* 通过二层循环 (nested loop) 进行连接，*s* 反复被访问。
最佳方案：最近最多使用 most recently used (MRU)。

7.1.2　记录的组织方式

数据库之数据组织

- 一个数据库以一组文件 (files) 的形式进行存储
- 一个文件是包含多条记录 (records) 的序列
- 一条记录是包含多个字段 (fields) 的序列

最简单常见的数据组织方式：每条记录是定长的，每个文件只存储一种类型的记录，不同表格用不同文件存储。

记录的表示

记录表示为字节序列 (sequence of bytes)。
DBMS 根据记录所在关系的模式，将记录的字节序列表示翻译为记录的全部属性值。
DBMS 的系统目录 (catalog) 记录关系的模式定义。
记录表示包含两部分：记录头 (header)、记录数据 (data)。

记录头包含记录的*元数据* (metadata)：
- 指向记录所在关系的模式定义的指针
- 记录的长度
- 记录的最后修改时间
- 记录的可见性（与并发控制相关）
- 记录的哪些属性值非空，用位图 (bitmap) 表示

记录数据由所有属性值拼接而成：
- 通常按定义关系模式时指定的属性顺序存储属性值
- 每个属性值在记录中存储位置的偏移量是 4 字节或 8 字节的倍数

变长记录的布局

定长属性值和变长属性值分别置于记录两端；记录头后紧跟指针数组，指向每个属性值。

页布局

一个页包含两个部分：*页头* (page header) 和*页数据* (page data)。
页头记录页的元数据，例如页的大小、页的校验和、页的可见性等。
页数据用*分槽页* (slotted page) 组织记录。*槽数组* (slot array) 存储第 i 个槽的起止位置的偏移量，每条记录占一个槽。

7.1.3　文件的组织方式

不同的 DBMS 采用不同的文件组织方式。
文件组织要考虑更新（增、删、改）和检索需求。
- 更新将涉及数据存储空间的扩展与回收问题
- 检索将涉及扫描整个数据库的问题、大批量处理数据问题
- 不同的需求要求不同的文件组织方法

无序文件组织

特点：记录可存储于任意有空间的位置，磁盘上存储的记录是无序的。
新记录总插入到文件尾部；删除记录时，可以直接删除该记录，所在位置的内容，也可以在该记录前标记「删除标记」。
在前者基础上，新增记录可以利用那些标记为「删除标记」的记录空间。
频繁删增记录时会造成空间浪费，所以需要周期性重新组织数据库。
数据库*重组* (Reorganization) 是通过移走被删除的记录使有效记录连续存放，从而回收那些由删除记录而产生的未利用空间。
无序文件中页的组织方式可以是基于链表的组织方式，也可以是基于页目录的组织方式。

有序文件组织

特点：记录按某属性或属性组值的顺序插入，磁盘上存储的记录是有序的。
用于存储排序的属性通常称为*排序字段* (ordering field)，通常，排序字段使用关系中的主码，所以又称*排序码* (ordering key)。

当按排序字段进行检索时，速度得到很大提高；但当按非排序字段检索时，速度可能不会提高很多。
在更新时要移动其他记录，为插入记录留出空间。
改进措施是可为将来有可能插入的记录预留空间（这可能造成空间浪费），或者再使用一个临时的无序文件（被称为*溢出文件*）保留新增的记录。
当采取溢出文件措施时，检索操作既要操作主文件，又要操作溢出文件，所以需要周期性重新组织数据库。
数据库重组是将溢出文件合并到主文件中，并恢复主文件中的记录顺序。

散列文件组织

特点：可以把记录按某属性或属性组的值，依据一个散列函数来确定记录存储在散列文件的哪个页。
用于进行散列函数计算的属性通常称为*散列字段* (hash field)，散列字段通常也采用关系中的主码，所以又称*散列码* (hash key)。

7.1.4　补充……

面向行的存储 (row-oriented storage) / 面向列的存储 (column-oriented storage)
适合联机分析处理 (OLAP) / 不适用于联机事务处理 (OLTP)

7.2　数据库索引

7.2.1　索引概述

索引的目的是从数据库中高效获取满足条件的记录。
索引通过存储（查找键，数据位置）来快速定位记录。

索引的分类

按照物理存储分类　索引按照物理存储分为*聚集索引*和*非聚集索引*。

- *聚集索引* (clustered index)：数据按照查找键排序
  - 按照查找键连续存储，范围查询效率高，一张表只能有一个
  - 维护增删时需要移动数据文件中的记录，开销稍大
- *非聚集索引* (Nonclustered Index)：辅助索引、二级索引
  - 一张表可以有多个，数据不一定按照索引列顺序存储

按照索引粒度分类　索引按照索引粒度分为*稠密索引*和*稀疏索引*。

- *稠密索引*：索引到数据记录（非聚集索引一定是稠密索引）
- *稀疏索引*：索引到文件页面（聚集索引通常是稀疏索引）

按照层数分类　索引按照层数分为*单级索引*（如哈希）和*多级索引*（索引文件上再建索引，如 B+ 树）。

按照字段特性分类　索引按照字段特性分为*主键索引*、*唯一索引*和*普通索引*。

- *主键索引*
  - 建立在主键字段上的索引
  - 一张表最多只有一个主键索引

  - 索引列的值不允许有空值
- 唯一索引
  - 建立在 `UNIQUE` 字段上的索引
  - 一张表可以有多个唯一索引
  - 索引列的值必须唯一，但是允许有空值

按照字段个数分类　索引按照字段个数分为*单列索引*（建立在单个字段上的索引）和*多列索引*（建立在多个字段上的索引）。

索引组织表 (index organization table)

索引组织表 = 聚集索引 + 数据文件，在聚集索引的索引项中存储数据本身，叶子节点为数据页面。
索引是否回表（通过索引找到页面记录才返回结果）：

- Index only scan 不回表：仅查找索引列，根据索引就可以返回结果，如学号是否存在
- Index scan 回表：根据索引列查找其他列的内容，如根据学号查找姓名

数据文件	数据记录指针	代表
索引组织表	主键	MySQL, SQLite
堆表	(页面号, 槽号)	PostgreSQL, DB2

索引组织表中，按照主键组织数据，数据记录会经常移动，物理位置改变。

索引类型

不同类型索引支持不同的查询类型。

查询类型	索引类型
点查询	哈希索引、B+ 树索引
范围查询	B+ 树索引
多列条件查询	位图索引
空间范围查询、KNN	多维索引（R 树、Quadtree、KD 树）

7.2.2　B+ 树索引

索引结构

结构：平衡多叉查找树，根到所有叶子节点路径长度相同。
扇出 m：每个节点最多分叉数。子节点数目范围 $\left[\left\lceil\frac{m}{2}\right\rceil, m\right]$，查找键数目范围 $\left[\left\lceil\frac{m}{2}\right\rceil - 1, m-1\right]$。
除了根节点，每个节点至少半满。
根的特殊情况：如果根不是叶子节点，至少 2 个子节点；如果根是叶子节点，有 $[0, m-1]$ 个查找键。
节点表示页面（磁盘索引），节点内指针指向索引页面号（内部节点）或者 (数据记录页面号, 数据记录槽号)（叶子节点）。
扇出 m 由页面大小和键值大小决定，通常是 200~300，10 亿条记录仅需要 4~5 层，I/O 次数少。

查找算法

点查询 (point query)

- 从根节点逐层加载索引页面到缓冲区，直到叶子节点
- 注意只判断存在性时，也要访问到叶子节点
- 内部节点存在的键，叶子节点中不一定存在

区间查询 (range query) $[L, U]$

- 叶节点兄弟之间有指针
- 首先根据查找键查找左端点（具有大于等于 L 的最小键的索引项）
- 然后按照节点指针向右线性扫描，如果键 $\leq U$ 则输出，否则终止

插入算法

- 找到应插入的叶子节点 L
- 将索引记录插入到 L 中
- 如果 L 有多余空间，结束
- 否则，需要分裂 L
  - 将节点内容均分为两部分，假设 L 中间键为 K'
  - L 中小于 K' 的保存在 L'，大于等于 K' 的移动到 L''
  - 将 K' 插入 L 父节点，并更新指向 L' 和 L'' 的指针
  - 如果 L 父节点溢出，继续分裂
- 内部节点（非叶结点）分裂
  - 中间键 K' 不放在新节点（L' 和 L''）中，仅插入父节点

删除算法

- 找到待删除索引记录所在叶节点
- 从叶节点中删除索引记录
- 如果节点中键数目大于等于 $\left\lceil \frac{m}{2} \right\rceil - 1$，则结束
- 如果节点中键数目小于等于 $\left\lceil \frac{m}{2} \right\rceil - 2$，发生下溢
  - 尝试从节点相邻的兄弟节点借一个索引项，使两者均至少半满，并更新父节点
  - 如果借不到，则将该节点与其兄弟节点合并
    - 如果节点 L 与左侧兄弟节点 L_1 合并，则从 L 的父节点中删除指向 L 的指针及相应的键
    - 如果节点 L 与右侧兄弟节点 L_2 合并，则从 L 的父节点中删除指向 L 的指针及相应的键
    - 如果 L 的父节点 N 至少半满，则完成合并；否则处理 N，使 N 至少半满
      - 如果 N 是根节点，且 N 中只有一个指针，则删除 N
      - 如果 N 是内节点，则尝试从 N 相邻的兄弟节点借一个指针和键，使两者均至少半满；如果借不到，则将 N 与其兄弟节点合并

7.2.3 哈希索引

- 静态哈希：哈希童数不变
  - 闭哈希：当插入哈希值满时，寻找空位置（线性探测、平方探测）
  - 开哈希：当插入哈希值满时，新建哈希桶，通过哈希桶链表链接

- 动态哈希：哈希桶数动态变化
  - 可扩展哈希表
  - 线性哈希表

哈希函数

哈希函数：给定查找键，返回一个整数表示（哈希值）。哈希值用于确定查找键在哈希表中的位置。

- MD5
- SHA256
- CRC-64
- PostgreSQL Lookup3Hash
- MurmurHash
- Google CityHash
- Facebook RocksDB XXHash

可扩展哈希表

一个可扩展哈希表包含 2^i 个桶，i 是*全局深度*。
键值为 K 的索引项属于编号等于 hash(K) 的前 i 位的桶。
每个桶中存放一个指针，指向存储该桶中索引项的页。
每个桶均没有*溢出页* (overflow page)，如果容纳得下，多个相邻桶中的全部索引项可以存入同一个页。
每个页记录一个*局部深度* (local depth) j, 该页中的全部索引项的 hash(K) 的前 j 位相同，用于标识这些索引项都存于这个页。

性质

- 桶数 $= 2^i$
- 全局深度 $\geq$ 每个页的局部深度
- 一个页被多个桶共享当且仅当这个页的局部深度小于全局深度
- 设一个页的局部深度为 j，则页中全部索引项的 hash(K) 的前 j 位相同

可扩展哈希表查找　确定索引项所属的桶，然后在桶指向的页中查找索引项。

可扩展哈希表插入　找到索引项被插入的页 P，如果 P 中有足够的空间空间，则将索引项插入 P；否则，分裂 P。
如果 P 溢出且 P 的局部深度小于全局深度

- 将 P 的局部深度 j 加 1
- 创建一个新页 P'，令 P 和 P' 的局部深度相同
- 根据键的哈希值的前 j 位，将 P 中索引项在 P 和 P' 中重新分配
- 更新指向 P 的桶中的指针

如果 P 溢出且 P 的局部深度等于全局深度

- 将全局深度加 1，即桶的数量翻倍
- 更新每个桶中的页指针
- 对于 P，使用前面介绍的方法分裂 P

可扩展哈希表删除　找到索引项所在的页，并从页中删除索引项。

可扩展哈希表的问题　哈希表每次容量加倍时需要阻塞对整个哈希表的访问，降低了系统性能。
如果少量桶多次溢出，则需要频繁对哈希表进行扩容，造成一部分桶空，导致空间浪费（例如所有元组哈希值前 n 位都相同，则需要频繁扩充）。【谁叫你选这么差劲的哈希函数，用 SHA 就不会有这个问题，千万不要自己造轮子!】
解决方案：线性哈希表

- 允许在页面后链接存放溢出数据的页面，更加节省空间
- 能够实现在哈希表扩容时仅需要对个别桶进行分裂，不会大规模阻塞对哈希表的访问

第八章　查询处理与优化

懒得抄了，自己看看课件算了。