



# 《数据库概论》学习笔记

## Introduction to Database Systems

作者：VectorPikachu

组织：EECS, Peking University

时间：May 31, 2025



# 目录

<b>第一章 关系模型</b>	<b>1</b>
1.1 关系基本概念 . . . . .	1
1.2 关系模型三要素 . . . . .	1
1.2.1 数据结构 . . . . .	1
1.2.2 数据操作 . . . . .	2
1.2.3 数据完整性 . . . . .	2
1.3 关系代数运算 . . . . .	3
1.3.1 基本关系代数运算 . . . . .	3
1.3.1.1 一元运算 . . . . .	3
1.3.1.2 多元运算 . . . . .	4
1.3.2 扩展关系代数运算 . . . . .	5
<b>第二章 SQL</b>	<b>10</b>
2.1 数据模式定义 . . . . .	10
2.1.1 数据类型 . . . . .	14
<b>第三章 关系中的非关系数据</b>	<b>15</b>
3.1 递归查询 . . . . .	15
3.1.1 层次结构的关系表示 . . . . .	15
3.1.2 递归查询 . . . . .	15
3.1.3 层次结构典型查询问题 . . . . .	17
3.2 XML . . . . .	17
3.3 JSON . . . . .	20
3.4 向量 . . . . .	22
<b>第四章 关系规范化</b>	<b>24</b>
4.1 关系模式的设计问题 . . . . .	24
4.2 函数依赖 . . . . .	24
4.3 码的定义（使用函数依赖） . . . . .	25
4.4 范式 . . . . .	25
4.4.1 1NF . . . . .	26
4.4.2 2NF . . . . .	27
4.4.3 3NF . . . . .	27
4.4.4 BCNF . . . . .	28
4.4.5 多值依赖 . . . . .	28
4.4.6 4NF . . . . .	29
4.4.7 PJNF . . . . .	29
4.5 Armstrong 公理系统 . . . . .	30
4.6 闭包计算 . . . . .	31
4.7 候选码计算 . . . . .	32
4.8 函数依赖的等价和覆盖 . . . . .	33
4.9 函数依赖和多值依赖的推理规则 . . . . .	34

---

4.10 模式分解 . . . . .	34
4.10.1 保持函数依赖分解 . . . . .	35
4.10.2 保持无损连接分解 . . . . .	35
4.10.3 关系模式分解算法 . . . . .	36
4.10.3.1 达到 BCNF 无损连接分解算法 . . . . .	36
4.10.3.2 达到 4NF 无损连接分解算法 . . . . .	37
4.10.3.3 达到 3NF 保持函数依赖的分解 . . . . .	38
4.10.3.4 同时保持函数依赖和无损连接的分解算法 . . . . .	38
4.11 模式调优 . . . . .	39
4.11.1 垂直划分 . . . . .	39
<b>第五章 事务</b>	<b>40</b>
5.1 SQL 中的事务 . . . . .	40
5.1.1 事务基本特性 ACID . . . . .	40
5.2 事务调度 . . . . .	41
5.2.1 并发调度中的不一致现象 . . . . .	42
5.3 事务隔离性级别 . . . . .	42
5.4 快照隔离 . . . . .	42
5.5 事务可串行化判定 . . . . .	43
5.5.1 冲突可串行化 . . . . .	43
5.5.2 视图可串行化 . . . . .	44
5.6 保存点 . . . . .	46
<b>第六章 并发控制</b>	<b>47</b>
6.1 基于锁的协议 . . . . .	47
<b>第七章 恢复控制</b>	<b>48</b>
7.1 故障类型 . . . . .	48
7.2 故障恢复 . . . . .	48
7.2.1 备份 . . . . .	48
7.2.2 日志 . . . . .	50
7.2.3 WAL, Write Ahead Log . . . . .	53
<b>第八章 数据库存储</b>	<b>54</b>
8.1 存储介质 . . . . .	54
<b>参考文献</b>	<b>55</b>

# 第一章 关系模型

## 1.1 关系基本概念

### 定义 1.1 (域 (Domain))

具有相同数据类型的一组值的集合. 如整数集合、字符串集合、全体学生集合.



### 定义 1.2 (笛卡尔积 (Cartesian Product))

一组域  $D_1, D_2, \dots, D_n$  的 **笛卡尔积** 为:

$$D_1 \times D_2 \times \dots \times D_n = \{(d_1, d_2, \dots, d_n) | d_i \in D_i, i = 1, 2, \dots, n\}.$$

笛卡尔积的元素  $(d_1, d_2, \dots, d_n)$  称作  $n$  元组 (tuple).

元组的每一个值  $d_i$  被称作分量 (component).

若  $D_i$  的基数为  $m_i$ , 则笛卡尔积的基数为  $\prod_{i=1}^n m_i$ .



### 定义 1.3 (关系)

笛卡尔积  $D_1 \times D_2 \times \dots \times D_n$  的子集称作在域  $D_1, D_2, \dots, D_n$  上的 **关系**. 用  $R(D_1, D_2, \dots, D_n)$  表示.  $R$  是关系的名字,  $n$  是关系的度或目.

关系是笛卡尔积中**有意义**的子集.



关系的性质:

1.  $P_1$ : 列是同质的, 是同一类型的数据, 即每一列中的分量来自同一域.
2.  $P_2$ : 不同的列可以来自同一域, 每列必须有不同的属性名. (一元联系、类型相同的属性)
3.  $P_3$ : 行列的顺序无关紧要.
4.  $P_4$ : 任意两个元组不能完全相同 (集合内不能有相同的两个元素)
5.  $P_5$ : 每一分量必须是不可再分的数据, 称其为作满足第一范式 (1NF) 的关系.

## 1.2 关系模型三要素

关系模型的三要素:

1. 数据结构.
2. 数据操作.
3. 数据完整性.

### 1.2.1 数据结构

关系模型的数据结构就是 **关系**: 实体集和联系都表示为关系.

### 定义 1.4 (候选码 (Candidate Key))

关系中的一个属性组, 其值能唯一标识一个元组. 若从属性组去掉任何一个属性, 它就不具有这一性质了, 这样的属性组称为候选码.



### 定义 1.5 (主属性)

任何一个候选码中的属性被称为主属性.



**定义 1.6 (主码 (Priamry Key, PK))**

进行数据库设计时, 从一个关系的多个候选码中选定一个作为主码.

**定义 1.7 (外码 (Foreign Key, FK))**

关系  $R$  中的一个属性组, 它不是  $R$  的码, 但它与另一个关系  $S$  的码相对应, 称这个属性组为  $R$  的外码.

**定义 1.8 (关系模式)**

关系的描述, 记为  $R(A_1, A_2, \dots, A_n)$ , 包括:

1. 关系名、关系中的属性名.
2. 属性向域的映像, 通常说明为属性的类型、长度等.
3. 属性间的数据依赖关系, 比如在特定的时间和教室只能安排一门课.

关系模式是稳定的.

**定义 1.9 (关系)**

关系是某一时刻对应某个关系模式的内容 (元组的集合). 关系是某一时刻的值, 是随时间不断变化的.

**定义 1.10 (关系型数据库)**

1. 型: 关系模式的集合, 数据库描述. 数据库的内涵 (Intension).
2. 值: 是某一时刻关系的集合. 数据库的外延 (Extension).



## 1.2.2 数据操作

关系操作是集合操作. 操作的对象及结果都是集合. 是一次一集合 (Set-at-a-time) 的方式.

非关系型的数据操作方式是一次一记录 (Record-at-a-time).

关系数据语言的特点:

1. 一体化: 对象单一, 都是关系, 因此操作符也单一
2. 非过程化: 用户只需提出“做什么”, 无须说明“怎么做”. 存取路径的选择和操作过程由系统自动完成
3. 面向集合的存取方式: 一次一关系.

抽象的关系模型查询语言:

1. 关系代数. 过程查询语言.
2. 关系演算: 元组关系演算、域关系演算. 非过程查询语言.

SQL(介于关系代数和关系演算之间, by IBM)、QUEL(基于 Codd 提出元组关系演算语言 ALPHA)、QBE.

## 1.2.3 数据完整性

**定义 1.11 (关系模型完整性)**

关系模型完整性由三部分组成:

1. 实体完整性
2. 参照完整性
3. 用户定义完整性

**定义 1.12 (实体完整性)**

关系的主码中的属性值不能为空值. (保证其实体存在.)



**定义 1.13 (参照完整性)**

如果关系  $R_2$  的外码  $F_k$  与关系  $R_1$  的主码  $P_k$  相对应, 则  $R_2$  中每个元组的  $F_k$  值或者等于  $R_1$  中某个元组的  $P_k$  值, 或者为空值.

如果关系  $R_2$  的某个元组  $t_2$  参照了关系  $R_1$  的某个元组  $t_1$ , 则  $t_1$  必须存在, 也即必须与客观存在的实体发生联系.

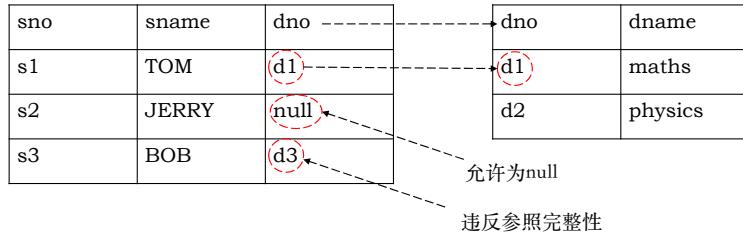


图 1.1: 参照完整性

**定义 1.14 (用户完整性)**

用户针对具体应用环境定义的完整性约束条件.



实体完整性和参照完整性由系统自动支持, 系统提供定义和检验用户定义的完整性的机制.

## 1.3 关系代数运算

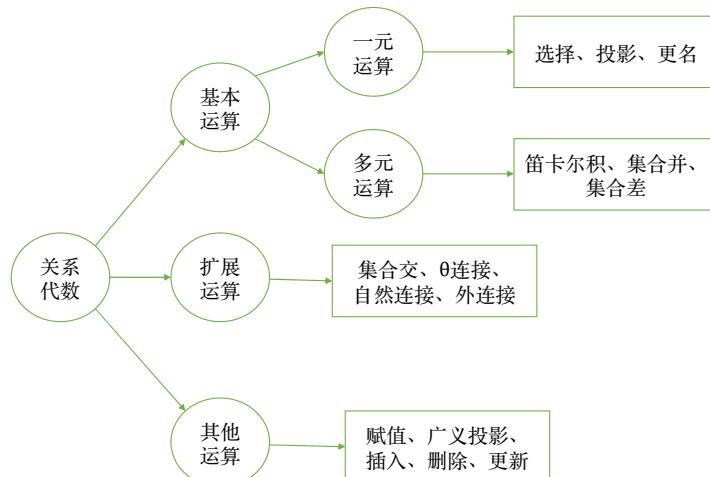


图 1.2: 关系代数示意图

### 1.3.1 基本关系代数运算

#### 1.3.1.1 一元运算

**定义 1.15 (选择运算)**

在关系中选择给定条件的元组 (行角度):

$$\sigma_F(R) = \{t | t \in R, F(t) = \text{true}\}.$$

$F$  由逻辑运算符连接算术表达式而成.



**定义 1.16 (投影运算)**

从关系中取若干列组成新的关系 (从列的角度):

$$\Pi_A(R) = \{t[A] | t \in R\}, A \subseteq R.$$



投影的结果要去掉相同的行:

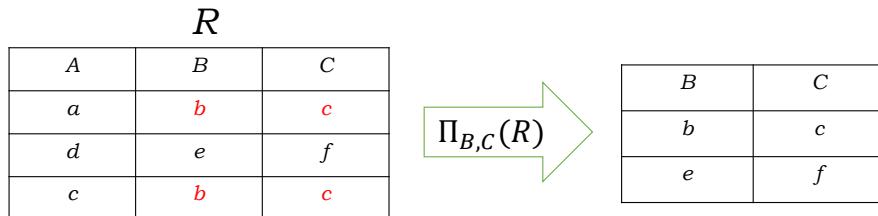


图 1.3: 投影运算要去掉相同的行

**定义 1.17 (更名运算)**

将关系  $R$  更名为  $S : \rho_S(R)$ ; 将计算表达式  $E$  更名为关系  $S : \rho_{S(A_1, A_2, \dots, A_n)}(E)$ .

1. 将更名运算施加到关系上, 得到具有不同名字的同一关系
2. 当同一关系多次参与同一运算时需要更名

**1.3.1.2 多元运算****定义 1.18 (并运算)**

$$R \cup S = \{r | r \in R \vee r \in S\}.$$

关系  $R$  和  $S$  进行并运算的前提是它们必须是相容的.

1. 关系  $R$  和  $S$  必须是同元的, 其属性数目必须相同.
2. 对  $\forall i, R$  的第  $i$  个属性和  $S$  的第  $i$  个属性的域必须相同.



例题 1.1 选修了 001 号或 002 号课程的学生号.

$$\begin{aligned} &\Pi_{\text{sno}}(\sigma_{\text{cno}=001 \vee \text{cno}=002}(\text{SC})) \\ &\Pi_{\text{sno}}(\sigma_{\text{cno}=001}(\text{SC}) \vee \sigma_{\text{cno}=002}(\text{SC})) \end{aligned}$$

**定义 1.19 (差运算)**

$$R - S = \{r | r \in R \wedge r \notin S\}.$$



例题 1.2 求选修了 001 号但未选修 002 号课程的学生号.

$$\begin{aligned} &\Pi_{\text{sno}}(\sigma_{\text{cno}=001}(\text{SC})) - \Pi_{\text{sno}}(\sigma_{\text{cno}=002}(\text{SC})) \\ &\Pi_{\text{sno}}(\sigma_{\text{cno}=001 \wedge \text{cno} \neq 002}(\text{SC})) \end{aligned}$$

**定义 1.20 (连串 (Concatenation))**

$r = (r_1, r_2, \dots, r_n), s = (s_1, s_2, \dots, s_m)$ ,  $r$  与  $s$  的连串定义为:

$$\hat{rs} = (r_1, r_2, \dots, r_n, s_1, s_2, \dots, s_m).$$



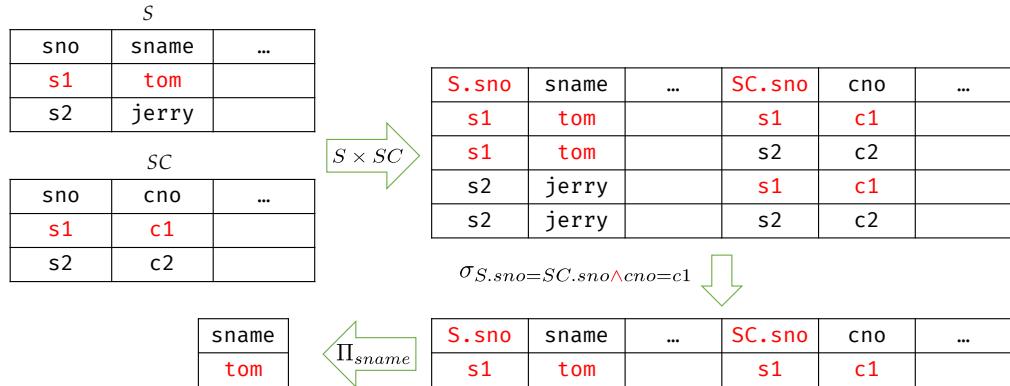
## 定义 1.21 (笛卡尔积)

$$R \times S = \{\hat{rs} | r \in R \wedge s \in S\}.$$

- $R \times S$  的度为  $R$  和  $S$  的度之和.
- $R \times S$  的元组个数为  $R$  和  $S$  的元组个数之积.



例题 1.3 求选修 c1 课程的学生姓名.



$$\Pi_{sname}(\sigma_{S.sno=SC.sno \wedge cno=c1}(S \times SC))$$

图 1.4: 笛卡尔积的使用

例题 1.4 求数学成绩比王红同学高的学生姓名.

$$\Pi_{S.姓名}(\sigma_{R.姓名=王红 \wedge R.课程=数学 \wedge S.课程=数学 \wedge R.成绩 < S.成绩}(R \times \rho_S(R)))$$

### 1.3.2 扩展关系代数运算

## 定义 1.22 (交运算)

$$R \cap S = \{r | r \in R \wedge r \in S\}.$$

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



例题 1.5 求同时选修了 001 号和 002 号课程的学生号.

$$\begin{aligned} &\Pi_{sno}(\sigma_{cno=001}(SC)) \cap \Pi_{sno}(\sigma_{cno=002}(SC)) \\ &\Pi_{sno}(\sigma_{cno=001 \wedge cno=002}(SC)) \end{aligned}$$

定义 1.23 ( $\theta$  连接)

从两个关系的广义笛卡儿积中选取给定属性间满足一定条件的元组:

$$R \bowtie_{A\theta B} S = \{\hat{rs} | r \in R \wedge s \in S \wedge r[A]\theta s[B]\} = \sigma_{r[A]\theta s[B]}(R \times S).$$



$A, B$  为  $R$  和  $S$  上度数相等且可比的属性列,  $\theta$  为算术比较符.

例题 1.6 求数学成绩比王红同学高的学生姓名.

$$\Pi_{S.\text{姓名}} \left( \sigma_{\text{课程}=\text{数学} \wedge \text{姓名}=\text{王红}}(R) \bowtie_{R.\text{成绩} < S.\text{成绩}} \sigma_{\text{课程}=\text{数学}} \rho_S(R) \right)$$

**定义 1.24 (等值连接)**

$\theta$  为等号的时候为等值连接.

**定义 1.25 (自然连接)**

从两个关系的广义笛卡儿积中选取在相同属性列  $B$  上取值相等的元组，并去掉重复的列:

$$R \bowtie S = \{\hat{r}s[\bar{B}] | r \in R \wedge s \in S \wedge r[B] = s[B]\}.$$

自然连接与等值连接不同：自然连接中相等的分量必须是相同的属性组，并且要在结果中去掉重复的属性，而等值连接则不必.



**例题 1.7** 求出 001 号学生所在系的名称:

$$\Pi_{\text{dname}}(\sigma_{\text{sno}=001}(S \bowtie \text{Dept}))$$

$$\Pi_{\text{dname}}(\sigma_{\text{sno}=001}(S) \bowtie \text{Dept})$$

$A$	$B$	$C$	$D$	
$\alpha$	1	$\alpha$	a	
$\beta$	2	$\gamma$	a	
$\gamma$	4	$\beta$	b	
$\alpha$	1	$\gamma$	a	
$\delta$	2	$\beta$	b	

⋈

$B$	$D$	$E$	
1	a	$\alpha$	
3	a	$\beta$	
1	a	$\gamma$	
2	b	$\delta$	
3	b	$\epsilon$	

=

$A$	$B$	$C$	$D$	$E$
$\alpha$	1	$\alpha$	a	$\alpha$
$\alpha$	1	$\alpha$	a	$\gamma$
$\alpha$	1	$\gamma$	a	$\alpha$
$\alpha$	1	$\gamma$	a	$\gamma$
$\delta$	2	$\beta$	b	$\delta$

图 1.5: 自然连接例子

**例题 1.8** 关系  $R(A, B), S(A, C)$ ,  $R$  与  $S$  中元组个数分别为 10, 15, 试填写下表.

条件	表达式	最小元组数	最大元组数
无任何条件	$R \bowtie S$	0	150
	$\Pi_A(R) \cup \Pi_A(S)$	1	25
A 是 R 的主码	$R \bowtie S$	0	15
	$\Pi_A(R) \cup \Pi_A(S)$	10	25
A 是 R 的主码 A 是 S 的外码	$R \bowtie S$	15	15
	$\Pi_A(R) \cup \Pi_A(S)$	10	10

表 1.1: 不同条件下的表达式及其元组数范围

自然连接的问题: 因失配而发生信息丢失.

**定义 1.26 (外连接)**

为避免自然连接时因失配而发生的信息丢失, 可以假定往参与连接的一方表中附加一个取值全为空值的行, 它和参与连接的另一方表中的任何一个未匹配上的元组都能匹配, 称之为外连接.

外连接 = 自然连接 + 未匹配元组 (悬挂元组).

外连接的形式:

1. 左外连接 = 自然连接 + 左侧表中未匹配元组.  $\bowtie_l$ .



2. 右外连接 = 自然连接 + 右侧表中未匹配元组.  $\bowtie$ .
3. 全外连接 = 左外连接 + 右外连接.  $\bowtie$ .

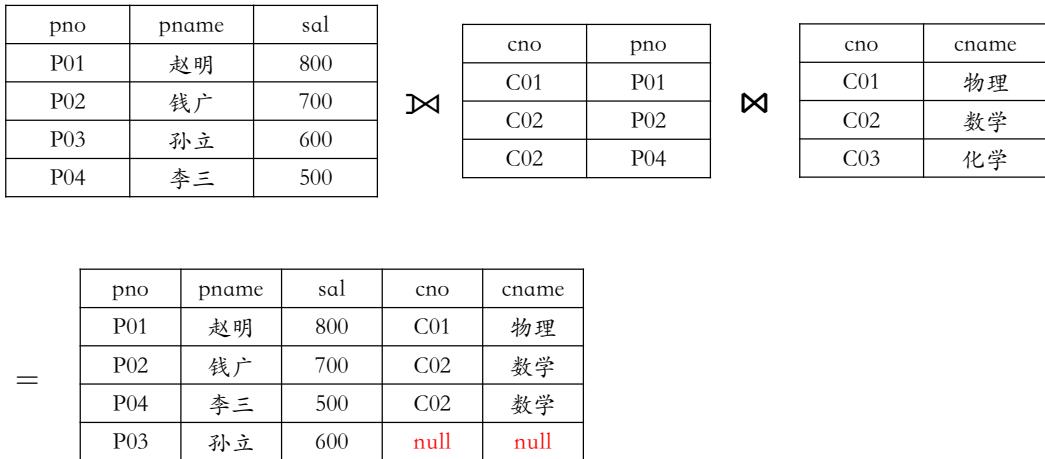


图 1.6: 左外连接示意图

外连接结合律不成立的反例:

R1		R2		R3	
A	B	B	C	A	C
1	2	2	3	4	5

$(R1 \bowtie R2) \bowtie R3$		
A	B	C
1	2	3
4	null	5

$R1 \bowtie (R2 \bowtie R3)$		
A	B	C
1	2	null
null	2	3
4	null	5

图 1.7: 外连接结合律不成立的反例

#### 定义 1.27 (半连接)

半连接 (Semi-join) 是一种用于优化查询的操作, 特别是在分布式数据库系统中. 它主要用于减少数据传输量, 提高查询效率. 半连接操作的目标是从两个关系 (表) 中返回第一个关系中的那些元组 (行), 这些元组与第二个关系中的至少一个元组匹配.  $\bowtie$ .

简单来说, 半连接操作会从一个表 (称为外部表或左表) 中选择记录, 并检查这些记录是否在另一个表 (称为内部表或右表) 中有对应的记录. 如果有, 则保留该记录; 如果没有, 则丢弃. 但是, 与普通连接不同的是, 结果集只包含来自外部表的列, 而不包含内部表的任何列.



半连接可以通过 SQL 查询中的 EXISTS 或 IN 子查询来实现.

#### 定义 1.28 (反半连接)

在半连接操作中, 我们会从第一个表 (外部表或左表) 中选出那些在第二个表 (内部表或右表) 中有匹配记录的行. 而反半连接则恰恰相反, 它的目的是找出那些在第一个表中存在但在第二个表中没有匹配记录的所有行.  $\bowtie$ .



在 SQL 中, 反半连接通常可以通过 NOT EXISTS 或者 LEFT JOIN 加上 IS NULL 的方式来实现.

$R$			$S$		
$A$	$B$	$C$	$B$	$C$	$D$
$a$	b	c	b	c	d
$d$	b	c	b	c	e
b	b	f	e	b	a
c	a	d	a	d	b

$R \bowtie S$			$S \bowtie R$		
$A$	$B$	$C$	$B$	$C$	$D$
$a$	b	c	b	c	d
$d$	b	c	b	c	e
c	a	d	a	d	b

$R \overline{\bowtie} S$			$R$ anti join $S$
$A$	$B$	$C$	
b	b	f	

图 1.8: 半连接示意图

### 定义 1.29 (外部并)

外部并操作的目标是在保持这种兼容性的同时合并这些关系. 不过, 与内部并 (Inner Union) 不同, 外部并也会保留那些在其中一个关系中存在但在另一个关系中不存在的属性值.

$$R \cup_{\text{outer}} S$$

$R$		$S$				
A	B	B	C	A	B	C
a	b	b	c	a	b	c
c	d	a	d	null	d	null

$R$			$S$							
A	B	C	B	C	D	A	B	R.C	S.C	D
a1	b1	c1	b1	c3	d1	a1	b1	c1	c3	d1
a2	b2	c2				a2	b2	c2	null	null

图 1.9: 外部并示意图

### 定义 1.30 (象集 (Image Set))

对于关系  $R(X, Z)$ ,  $X, Z$  是属性组,  $x$  是  $X$  上的取值, 定义  $x$  在  $R$  中的象集为:

$$Z_x = \{t[Z] | t \in R \wedge t[X] = x\}$$

注 象集实际是是: (1) 先从  $R$  中选出在  $X$  上取值为  $x$  的元组; (2) 只保留  $Z$  属性.

例题 1.9 如何求得选修了全部课程的学生?

思路一: 判断每个学生的课程象集是否包含了整个课程集合.

$$\{u | r \in SC \wedge u = r[\text{姓名}] \wedge \text{课程名}_u \supseteq C\}$$

思路二：判断学生与课程集合构成的笛卡尔积是否完全包含在选课集合中。

$$\{u \mid r \in SC \wedge u = r[\text{姓名}] \wedge \forall c \in C, (u, c) \in SC\}$$

### 定义 1.31 (除法)

除法通常用来找出在第一个关系中与第二个关系中的所有元素都有匹配的那些元组。它的定义式为：

$$R(X, Y) \div S(Y) = \{x \mid r \in R \wedge x = r[X] \wedge Y_x \supseteq S\},$$

$$R(X, Y) \div S(Y) = \{u \mid u \in \Pi_X(R) \wedge \forall v \in S, \widehat{uv} \in R\}.$$

除法的计算表达式为：

$$R(X, Y) \div S(Y) = \Pi_X(R) - \Pi_X(\Pi_X(R) \times \Pi_Y(S) - R).$$



## 第二章 SQL

SQL: Structured Query Language

SQL 语言的特点:

1. 语言简洁, 易学易用。
2. 面向集合的操作方式, 一次一集合。
3. 高度非过程化。用户只需提出“做什么”, 无须告诉“怎么做”
4. 一体化。单一的结构——关系, 带来了数据操作符的统一。
5. 两种使用方式, 统一的语法结构。
  - (a). 既是自含式(用户使用)的
  - (b). 又是嵌入式的(程序员使用)

SQL 功能	操作符
数据定义	create, alter, drop
数据查询	select
数据修改	insert, update, delete
数据控制	grant, revoke

表 2.1: SQL 主要操作符

text2SQL: 建立自然语言与结构化数据之间的关系。

### 2.1 数据模式定义

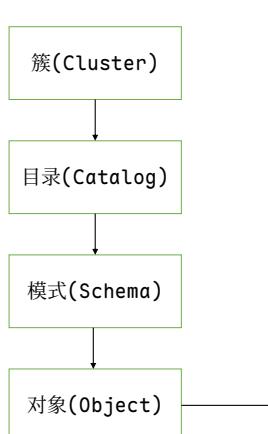


图 2.1: 标准 SQL 中的数据定义对象

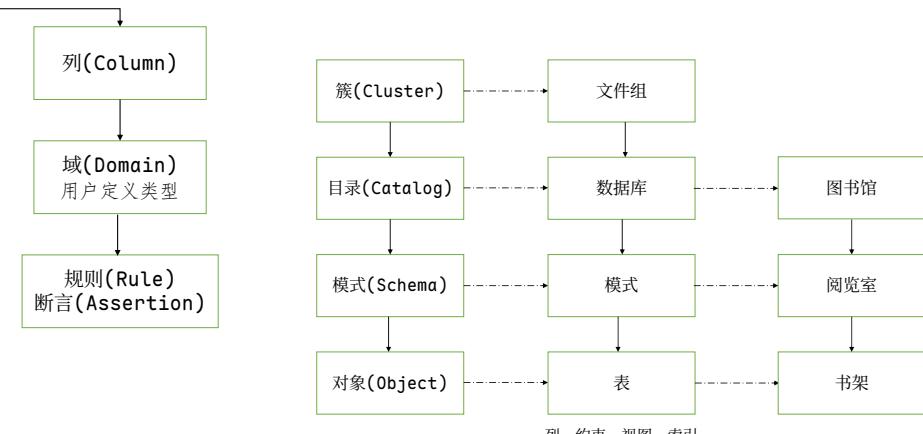


图 2.2: 实际数据库(SQL Server)中的定义对象

SQL Server: 模式把对象和用户分离开来.

对象命名: <数据库>.<模式>.<表>.

创建模式:

```
create schema <模式名>
create schema University.Library
```

数据库定义: SQL Server

```
create database <数据库名>
[on [primary] <文件描述> <文件组> ...]
[log on <文件描述> <文件组> ...]
```

最简单的创建数据库的命令: `create database University.`

`use` 命令指定当前要使用的数据库: `use University.`

```
create database demoDB1
on primary
(
    name = demo_dat1,
    filename = 'D:\SQL_Practice\demodata1.mdf',
    size = 10,
    maxsize = 50
)
log on
(
    name = demo_log1,
    filename = 'D:\SQL_Practice\demodata1.ldf',
    size = 5,
    filegrowth = 5
)
```

数据库定义: MySQL

```
create database <数据库名>
[ default character set utf8
  default collate utf8_Chinese_ci ]
```

`create database` 等于 `create schema`.

MySQL 表空间:



图 2.3: MySQL 表空间

```
create tablespace myTs 'ts1.ibd' engine = innodb
create tablespace myTs add datafile
  'F:\\test_mysql_tablespace\\first.ibd'
create table myTb (...) tablespace myTs
```

创建基本表的语法命令:

```
create table <表名> (
  <列名> <数据类型> [default <缺省值>] [not null] [unique]
  [, <列名> <数据类型> [default <缺省值>] [not null] [unique]]
  ...
)
```

```
[, primary key (<列名> [, <列名>] ...)]
[, foreign key (<列名> [, <列名>] ...)
  references <表名> (<列名> [, <列名>] ...)]
[, check(<条件>)]
)
```

下面是创建表的一些例子:

```
create table student
( sno char(8),
  sname char(8) not null default '佚名',
  age tinyint,
  sex char(1),
  primary key (sno),
  check (sex = 'M' or sex='F')
)
```

```
create table course
( cno char(8) primary key,
  cname char(8) not null unique,
  pcno char(8) foreign key references C(cno),
  credit tinyint
)
```

```
create table SC
( sno char(8) foreign key references S(sno),
  cno char(8) foreign key references C(cno),
  grade tinyint,
  primary key (sno, cno),
  check((grade is null) or grade between 0 and 100)
)
```

修改基本表: 更改、添加、除去列和约束.

```
alter table <表名>
[add column <子句>]
[add constraint <子句>]
[drop <子句>]
[alter column <子句>]
```

```
-- 在student表age列之后加入addr
alter table student add column addr CHAR(30) after age;
-- 把addr列重命名为address
alter table student change addr address CHAR(50) not null;
-- 试修改teacher表中的salary列的数据类型为bigint
alter table teacher modify salary bigint;
-- 重命名一个表中的列名从sal到salary
alter table rename sal to salary
```

删除基本表:

```
drop table <表名>;
```

删除表定义及该表的所有数据、索引、触发器、约束和权限规范.

drop table 不能删除被 foreign key 约束所引用的表, 必须先除去 foreign key 约束或引用表.

任何引用已删除表的视图或存储过程必须通过 drop view 或 drop procedure 语句显式除去.

标准 SQL 中的信息视图:

```
INFORMATION_SCHEMA.SCHEMATA
INFORMATION_SCHEMA.TABLES
INFORMATION_SCHEMA.COLUMNS
INFORMATION_SCHEMA.CHECK_CONSTRAINTS
INFORMATION_SCHEMA.VIEWS
INFORMATION_SCHEMA.DOMAINS
```

MySQL 中的信息视图查询:

```
select schema_name from information_schema.schemata;
select table_name from information_schema.tables;
select column_name from information_schema.columns where table_name = 'student';
```

sysobjects		
列名	数据类型	描述
name	sysname	对象名
Id	int	对象标识号
xtype	char(2)	对象类型
uid	smallint	所有者对象的用户 ID
crdate	datetime	对象的创建日期
schema_ver	int	版本号, 该版本号在每次表的架构更改时都增加

表 2.2: 表定义相关的字典表: SQL Server

syscolumns		
列名	数据类型	描述
name	sysname	列名或过程参数的名称
id	int	该列所属的表对象 ID
xtype	tinyint	systypes 中的物理存储类型
xusertype	smallint	扩展的用户定义数据类型 ID
length	smallint	systypes 中的最大物理存储长度
offset	smallint	该列所在行的偏移量; 如果为负, 表示可变长度行
type	tinyint	systypes 中的物理存储类型
usertype	smallint	systypes 中的用户定义数据类型 ID
isnullable	int	表示该列是否允许空值

表 2.3: 表定义相关的字典表: SQL Server

SQL 中, 任何时候都可以执行一个数据定义语句, 随时修改数据库结构.

## 2.1.1 数据类型

数据类型	范围	<b>unsigned</b> 范围	存储字节数
tinyint	$-2^7 \sim 2^7 - 1$	$0 \sim 2^8 - 1$	1 字节
smallint	$-2^{15} \sim 2^{15} - 1$	$0 \sim 2^{16} - 1$	2 字节
mediumint	$-2^{23} \sim 2^{23} - 1$	$0 \sim 2^{24} - 1$	3 字节
int	$-2^{31} \sim 2^{31} - 1$	$0 \sim 2^{32} - 1$	4 字节
bigint	$-2^{63} \sim 2^{63} - 1$	$0 \sim 2^{64} - 1$	8 字节

表 2.4: MySQL 整数数据类型及其范围和存储字节数

```
create table test_int (
    a(6) tinyint zerofill,
    b(6) tinyint unsigned );
insert into test_int values (1, 111);
select a, b from test_int;
-- a 000001 b 111
select a - b from test_int;
-- ERROR 1690 (22003): BIGINT UNSIGNED value is out of range
```

宽松模式: `set sql_mode = 'ANSI'`. 对于违反数据约束的有一些默认操作.

严格模式: `set sql_mode = 'traditional'`. 直接报错.

# 第三章 关系中的非关系数据

大数据分为:

1. 结构化数据. 指关系型数据表.
2. 半结构化数据. 指关系结构与内容混合在一起的数据类型.
3. 非结构化数据. 文档、视频、音频、图片.

## 3.1 递归查询

### 3.1.1 层次结构的关系表示

邻接表: `Adjacent(child, parent)`.

物化路径: `material_path(node, path)`. 从起点出发的路径.

嵌套集合: `nestedSet(node, left_value, right_value)`. 嵌套集模型是根据树遍历来对节点进行编号, 遍历会访问每个节点两次, 按访问顺序分配数字, 并在两次访问中都分配。这将为每个节点留下两个数字, 它们作为节点两个属性存储。这使得查询变得高效: 通过比较这些数字来获得层级结构关系。但是更新数据将需要给节点重新分配数字, 因此变得低效。

### 3.1.2 递归查询

层次结构的传递闭包: 使用递归查询.

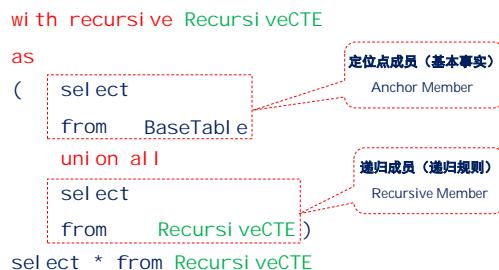


图 3.1: MySQL 中的递归查询

```
with recursive nums(n)
as
(
    select 1
    union all
    select n+1
    from nums
    where n < 100
)
select sum(n) from nums
```

```
with recursive Components ( part, subpart )
as
(
    select part, subpart
    from Assembly
    union all
    select A.part, C.subpart
    from Components A
    join Components C
    on A.subpart = C.part
)
```

```
from Assembly A, Components C
where A.subpart = C.part )
select * from Components where part = 'trike'
```

**例题 3.1** 利用递归查询计算经理下属. 表结构: emp(empid, ename, mgrid).

```
declare @root = 1 as int
-- 定义递归起点编号
with recursive SubsCTE
as ( select empid, ename, 0 as lvl
      from emp
      where empid = @root
      -- 初始查询
      union all
      select C.empid, C.ename, P.lvl + 1
      from SubsCTE as P join emp AS C on C.mgrid = P.empid
      -- 查询当前层级的表: SubsCTE
    )
select * from SubsCTE
```

```
declare @lvl = 0 as int
create temporary table Subs( empid int, level int )
-- 插入根节点
insert into Subs( empid, level )
  select empid, @lvl from emp
  where empid = @root;
while found_rows() > 0

-- 递归查找下属
begin
  set @lvl = @lvl + 1
  insert into Subs( empid, level )
    select C.empid, @lvl
    from Subs as P join emp as C
    on P.lvl = @lvl - 1 and C.mgrid = P.empid
end
```

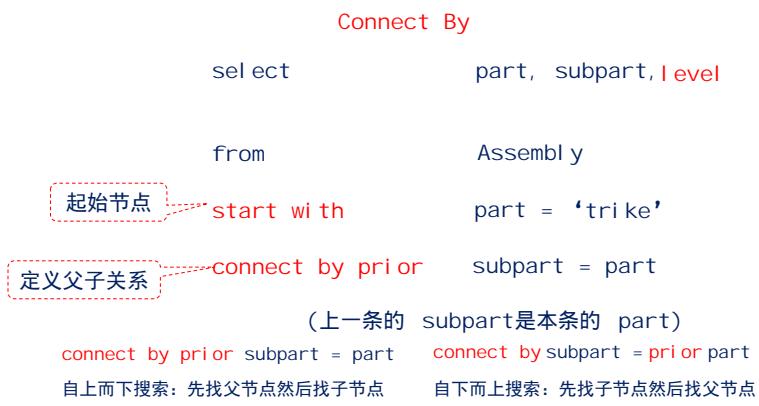


图 3.2: Oracle 中的递归查询

### 3.1.3 层次结构典型查询问题

树形结构（数据存放在邻接表中）

1. 返回给定节点的所有下属，并标注层级
  2. 返回给定节点的所有上级，并标注层级
  3. 移动子树，将一颗子树的根节点换成另外一个
  4. 将邻接表转换为物化路径，并存放在一个表中
  5. 基于物化路径表，查询指定节点的所有子节点
  6. 基于物化路径表，查询指定节点的所有父节点
  7. 将邻接表转换为嵌套集合，并存放在一个表中
  8. 基于嵌套集合表，查询指定节点的所有子节点
  9. 基于嵌套集合表，查询指定节点的所有父节点
- 图
1. 计算传递闭包
  2. 计算最短路径
  3. 环路检测

## 3.2 XML

1. 标签 (tag): 定义数据成为一个元素.
2. 文本 (text): 说明元素属性.
3. 元素 (elements): < 起始标签 > 元素属性 < 结束标签 > 的组合.  
格式正确的 (well-formed)XML 文档

  1. 只能有唯一的根元素
  2. 所有元素都必须有起始标签和结束标签
  3. 大小写一致, XML 区分大小写
  4. 子元素必须被上层元素完全包含
  5. 属性值必须被双引号或单引号括起来
  6. 元素内的属性不能被重复使用

```
<books>
  <book id="0001">
    <bookname> 数据库技术</bookname>
  </book>
  <book id="0001">
    <bookname> 数据仓库技术</bookname>
  </book>
</books>
```

HTML: HyperText Markup Language. 描述数据的显示格式

XML: eXtensible Markup Language. 描述数据的内容

XML: 半结构化模型的资源描述框架. RDF (Resource Description Framework)

存储 RDF 使用三元组: < 标识符, 属性名, 属性值 >. 或者 <Subject, Property, Object>.

基于三元组表的 RDF 查询: Implementation Techniques for Main Memory Databases 喜欢什么和不喜欢什么?

```
SELECT C.object, D.object
FROM triples A, triples B, triples C, triples D
```

```
-- 四张相同的表triples, 别名分别为A, B, C, D
WHERE    A.subject = B.object
AND      A.property = "title"
AND      A.object = "Implementation Techniques for Main Memory Databases"
-- A中找到标题为给出标题的对象.
AND      B.property = "authorOf"
-- B中找到authorOf A找出的书, 也就是找出作者
AND      B.subject = C.subject
AND      C.property = "likes"
-- C中找出这个作者喜欢的东西
AND      C.subject = D.subject
AND      D.property = "dislikes"
-- D中找出这个作者不喜欢的东西
```

三元组的单属性表表示: 过于细碎, 导致太多的连接操作.

三元组的宽表表示: 太多列 + 稀疏.

DTD(Document Type Definition) 是 XML 的核心机制之一, 用于定义 XML 文档的结构和规则. 它的主要作用是:

1. 验证 XML 文档的合法性: 确保 XML 文档的元素、属性、内容等符合预定义的结构规则.
2. 约束文档格式: 通过定义元素的嵌套关系、属性的取值范围等, 保证数据的一致性和正确性.
3. 支持数据交换: 为不同系统之间共享数据提供统一的结构规范.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note [
  <!ELEMENT note (to,from,heading,body)>
  <!ELEMENT to (#PCDATA)>
  <!ELEMENT from (#PCDATA)>
  <!ELEMENT heading (#PCDATA)>
  <!ELEMENT body (#PCDATA)>
]>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

每个 DTD 规则表达式是一个有限状态自动机.

```
<!ELEMENT books (book+)>
<!ELEMENT book (title, author+, year, price)>
<!ATTLIST book id ID #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ENTITY company "MyPublisher">
```

XML 数据模型 (DOM, Document Object Model) 是 W3C(万维网联盟) 制定的一种标准接口规范, 用于动态访问和操作 XML 或 HTML 文档的内容、结构和样式. 它是通过将文档解析为树形结构 (即 DOM 树), 并以对象的形式表示每个节点 (如元素、属性、文本等), 从而允许程序或脚本以编程方式操作文档.

XPath (XML Path Language) 是一种用于在 XML 或 HTML 文档中定位和选择节点的查询语言。它通过路径表达式 (Path Expressions) 和条件筛选 (谓语) 来导航 XML 文档的树形结构，从而提取特定的数据或节点。

表达式	含义	示例
/	从根节点开始选取	/bookstore/book
//	从当前节点开始，选取文档中任意位置的节点	//title
.	当前节点	.//price
..	父节点	../author
@	选取属性	@id

表 3.1: 路径表达式

表达式	含义	示例
*	选取所有子节点	/* (根节点的所有子元素)
@*	选取所有属性	//@lang
text()	选取文本节点	//price/text()

表 3.2: 节点选择

谓语 (条件筛选):

- 索引筛选: [n] 选择第 n 个节点 (从 1 开始计数):

/bookstore/book[1] % 第一个 book 节点

- 属性筛选: [@ 属性名 =' 值 ']:

//book[@id='b001'] % id 为 b001 的 book 节点

- 文本筛选: [text()=' 值 '] 或 contains(text(), ' 部分值 '):

//title[text()='XML权威指南'] % 文本完全匹配

//title[contains(text(), '指南')] % 文本包含 "指南"

轴	含义	示例
child::	子元素 (默认可省略)	child::book (等价于 book)
parent::	父节点	parent::bookstore
ancestor::	所有祖先节点	ancestor::bookstore
descendant::	所有后代节点	descendant::title
following-sibling::	所有后续兄弟节点	following-sibling::author

表 3.3: 轴 (Axis)

MySQL 中的 XML 函数: ExtractValue(xml\_target, xpath\_expr), 从 XML 数据中提取特定节点的值。

```
-- 提取 <to> 节点的值
SELECT ExtractValue('<note><to>Tove</to></note>', '/note/to');
-- 结果: Tove
```

```
-- 提取多个节点的值（使用 `|` 分隔）
SELECT ExtractValue('<a><b>1</b><c>2</c></a>', '//b/text() | //c/text()');
-- 结果: 1 2
```

### 3.3 JSON

JSON 的基本成分为:

1. 对象. {属性名: 属性值, 属性名: 属性值}. E.g., {'name': 'Tom', 'hobby': ['sing', 'dance']}.
2. 数组. [value, value, value ...]. [{name: 'Tom', age: 12}, {...}].

MySQL 中生成 JSON 的函数:

1. `JSON_ARRAY` 函数. 创建一个 JSON 数组.

```
JSON_ARRAY(val1, val2, ..., valN)
```

```
SELECT JSON_ARRAY(1, 'apple', NULL, TRUE);
-- 输出: [1, "apple", null, true]
```

2. `JSON_OBJECT` 函数. 将键值对列表返回为 JSON 对象.

```
JSON_OBJECT('key1' VALUE val1, 'key2' VALUE val2, ...)
```

```
SELECT JSON_OBJECT('id' VALUE 1, 'name' VALUE 'Alice');
-- 输出: {"id": 1, "name": "Alice"}
```

MySQL 中的 JSON 数据类型:

```
CREATE TABLE testJSON (
    a JSON,
    b INT
);
INSERT INTO testJSON VALUES
('[3, 10, 5, "x", 44]', 33),
('[3, 10, 5, 17, [22, "y", 66]]', 0);
-- 提取嵌套值
SELECT a->"$[3]", a->"$[4][1]" FROM testJSON;
-- 结果分别为: ["x", 17], [NULL, "y"]
```

JSON 的检索操作:

```
create table user (
    id int not null primary key auto_increment,
    info json );
insert into user(info) values (
    '{"name":"wangming",
     "age":18,
     "address":{"province":"sichuan","city":"chengdu"},
     "hobby" :["sing", "dance"]}' );
```

```
-- 提取多层嵌套值
SELECT json_extract('[10, 20, [30, 40]]', '$[2][*]');
-- 输出: [30, 40]

-- 提取 JSON 对象中的字段
SELECT json_extract(info, '$.address.city') FROM user;
-- 输出: "chengdu"

-- 提取多个字段
SELECT json_extract(info, '$.name', '$.hobby') FROM user;
-- 输出: ["wangming", ["sing", "dance"]]
```

将 JSON 展开为平面表:

```
-- 展开 JSON 数组为行
SELECT *
FROM json_table(
  '[{"a":3}, {"a":2}, {"b":1}, {"a":0}, {"a":[1,2]}]' ,
  "$[*]"
  COLUMNS(
    rowid FOR ORDINALITY,
    ac VARCHAR(100) PATH ".$.a" DEFAULT '111' ON EMPTY DEFAULT '999' ON ERROR,
    aj JSON PATH ".$.a" DEFAULT '{"x": 333}' ON EMPTY,
    bx INT EXISTS PATH ".$.b"
  )
) AS tt;
```

上面展开后的结果为:

rowid	ac	aj	bx
1	3	"3"	0
2	2	2	0
3	111	{"x": 333}	1
4	0	0	0
5	999	[1, 2]	0

将 JSON 展开为平面表: 数组的 unwind 操作.

```
-- 展开 JSON 中的嵌套数组 (类似 unwind 操作)
SELECT *
FROM json_table(
  '[{"a": 1, "b": [11, 111]}, {"a": 2, "b": [22, 222]}, {"a": 3}]',
  "$[*]"
  COLUMNS(
    a INT PATH ".$.a",
    NESTED PATH ".$.b[*]" COLUMNS(b INT PATH "$")
  )
) AS jt
WHERE b IS NOT NULL;
```

将平面表转换为 JSON:

```
-- 将平面表转换为 JSON 数组
CREATE TABLE score (
    sname CHAR(10),
    cname CHAR(10),
    score INT
);

INSERT INTO score VALUES
('张三', '数学', 96),
('张三', '语文', 99),
('李四', '数学', 98),
('李四', '语文', 88);

-- 转换为 JSON 数组
SELECT CONCAT(
    '[',
    GROUP_CONCAT(
        json_object('sname' VALUE sname, 'cname' VALUE cname, 'score' VALUE score)
    ),
    ']'
) AS scores
FROM score;
```

## 3.4 向量

向量嵌入使得我们具有处理非结构化数据的能力.

向量数据库中的核心问题: 近似最近邻查找. ANN (Approximate nearest neighbor search)

多维索引使用 k-d tree: [k-d 树文档](#)

近似最近邻搜索算法 ANNOY: [\[1\]](#).

高维数据的搜索: HNSW: Hierarchical Navigable Small World [\[2\]](#).

基于 PostgreSQL 的向量插件: pgvector (基于 IVFFlat 索引), pg\_embedding (基于 HNSW 索引).

```
-- 1. 启用插件
CREATE EXTENSION vector;

-- 2. 创建表
CREATE TABLE items (
    id bigserial PRIMARY KEY,
    embedding vector(3)
);

-- 3. 插入数据
INSERT INTO items (embedding) VALUES
    ('[1,2,3]'),
    ('[4,5,6]');

-- 4. 创建索引 (IVFFlat)
```

```
CREATE INDEX idx_items_embedding_ivfflat
ON items
USING ivfflat (embedding vector_12_ops)
WITH (lists = 100);

-- 5. 查询相似向量
SELECT *
FROM items
ORDER BY embedding <=> '[3,1,2]'
LIMIT 5;
```

## 第四章 关系规范化

### 4.1 关系模式的设计问题

信息在关系模式中的表示完全取决于主码。

职工	级别	工资
赵明	4	500
钱广	5	600
孙志	6	700
李开	5	600
周祥	6	700

表 4.1: 职工信息表

1. 信息的不可表示问题。

(a). 插入异常: 如果没有职工具有 8 级工资, 则 8 级工资的工资数额就难以插入。

(b). 删除异常: 如果仅有职工赵明具有 4 级工资, 删掉赵明则会将有关 4 级工资的工资数额信息也一并删除。

2. 信息的冗余问题.

(a). 数据冗余: 职工很多, 工资级别有限, 每一级别的工资数额反复存储多次。

(b). 更新异常: 如果将 5 级工资的工资数额调为 620, 则需要找到每个具有 5 级工资的职工, 逐一修改。

$\boxed{\text{职工}} \rightarrow \boxed{\text{级别}} \rightarrow \boxed{\text{工资}}$ 。分解为:  $\boxed{\text{职工}} \rightarrow \boxed{\text{级别}} + \boxed{\text{级别}} \rightarrow \boxed{\text{工资}}$ 。

### 4.2 函数依赖

#### 定义 4.1 (函数依赖)

设  $R(U)$  是属性集  $U$  上的关系模式,  $X, Y \subseteq U$ ,  $r$  是  $R(U)$  上的任意一个关系, 如果成立:

$$\text{对 } \forall t, s \in r, \text{ 若 } t[X] = s[X], \text{ 则 } t[Y] = s[Y]$$

则称“ $X$  函数决定  $Y$ ”或者“ $Y$  函数依赖于  $X$ ”, 记作  $X \rightarrow Y$ . 称  $X$  为决定因素.

例如,  $sno \rightarrow sname$ ,  $(sno, cno) \rightarrow grade$ .

#### 定义 4.2 (函数依赖的双重否定形式的定义)

不存在  $t, s \in r$ ,  $t[X] = s[X]$ , 但  $t[Y] \neq s[Y]$ .

#### 定义 4.3 (平凡的函数依赖)

如果  $X \rightarrow Y$ ,  $Y \subseteq X$ , 则称其为平凡的函数依赖. 否则称为非平凡的函数依赖.

如,  $(sno, sname) \rightarrow sname$  是平凡的函数依赖.

一个关系模式有  $n$  个属性, 在它上面成立的所有可能的函数依赖有多少个? 非平凡的函数依赖有多少个?

**Answer.** 需要计算  $X \rightarrow Y$  的个数. 其中  $X$  和  $Y$  都是非空子集, 这就可以知道答案为  $(2^n - 1)^2$ . 先计算出平凡的函数依赖个数. 也就是  $Y \subseteq X$  的个数,  $\sum_{k=1}^n \binom{n}{k} (2^k - 1) = 3^n - 2^n$ . 接着减去这个数, 得到:  $2^{2n} - 2^n - 3^n + 1$ .

**定义 4.4 (完全函数依赖)**

如果  $X \rightarrow Y$ , 且对于任意  $X$  的真子集  $X'$ , 都有  $X' \not\rightarrow Y$ , 则称  $Y$  对  $X$  完全函数依赖, 记作  $X \xrightarrow{f} Y$ . 否则称  $Y$  对  $X$  部分函数依赖, 记作  $X \xrightarrow{p} Y$ .

**定义 4.5 (传递函数依赖)**

在  $R(U)$  中, 如果:  $X \rightarrow Y, Y \rightarrow Z, Y \not\rightarrow X$ , 且  $Z \not\subseteq Y$ , 则称  $Z$  对  $X$  传递函数依赖.

**注** 定义4.5中  $Y \not\rightarrow X$  意味着必须通过  $Y$  来传递, 而  $Z \not\subseteq Y$  说明并非平凡依赖, 否则可以直接导出  $X \rightarrow Z$ .

**主要问题:** 如果关系模式设计不当, 把本来彼此没有依赖关系的两个属性放在同一个关系模式中, 所造成的对候选码的部分依赖和传递依赖是在现实中不存在的, 从而会出现异常。E.g., 学号  $\rightarrow$  系号, 系号  $\rightarrow$  系主任, 这样学号  $\rightarrow$  系主任, 这是不合理的。

## 4.3 码的定义（使用函数依赖）

**定义 4.6 (超码)**

设  $K$  为  $R(U, F)$  的属性或属性组, 若  $K \rightarrow U$ , 则称  $K$  为  $R$  的超码.

**例题 4.1**  $R(ABC; \{A \rightarrow B, B \rightarrow C\})$  有多少超码?

超码:  $\{A, AB, AC, ABC\}$ .

**定义 4.7 (候选码)**

设  $K$  为  $R(U, F)$  的超码, 若  $K \xrightarrow{f} U$ , 则称  $K$  为  $R$  的候选码.

**例题 4.2**  $R(ABC; \{A \rightarrow B, B \rightarrow C\})$  有多少候选码?  $\{A\}$ .

**定义 4.8 (主属性)**

包含在任意候选码中的属性, 称作主属性.

**例题 4.3**  $R(ABC; \{AB \rightarrow C, C \rightarrow AB\})$ , 计算  $R$  的主属性. 候选码:  $\{AB, C\}$ , 主属性:  $\{A, B, C\}$ .

**定义 4.9 (全码)**

关系模式  $R(U, F)$  的码由整个属性集  $U$  构成.

**注** 一个全码的关系模式不存在非平凡的函数依赖, 否则就会有更小的码.

## 4.4 范式

接下来考虑这个关系模式:  $S(\underline{sno}, sname, dno, dean, \underline{cno}, grade)$ .

函数依赖为:

$$(sno, cno) \xrightarrow{f} grade$$

$$sno \rightarrow sname$$

$$sno \rightarrow dno$$

$$dno \rightarrow dean$$

数据在表4.2中.

sno	sname	dno	dean	cno	grade
S01	杨明	D01	思齐	C01	90
S02	李婉	D01	思齐	C01	87
S01	杨明	D01	思齐	C02	92
S03	刘海	D02	述圣	C01	95
S04	安然	D02	述圣	C02	78
S05	乐天	D03	省身	C01	82

表 4.2: 学生数据表

**定义 4.10 (范式)**

范式是对关系的不同数据依赖程度的要求.

**定义 4.11 (规范化)**

通过模式分解将一个低级范式转换为若干个高级范式的过程称作规范化.

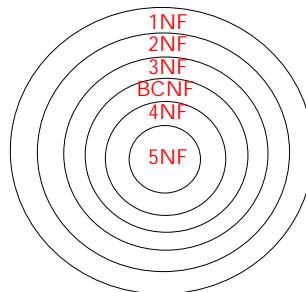


图 4.1: 范式

**4.4.1 1NF****定义 4.12 (1NF)**

关系中每一分量 **不可再分**. 也即不能以集合、序列等作为属性值.

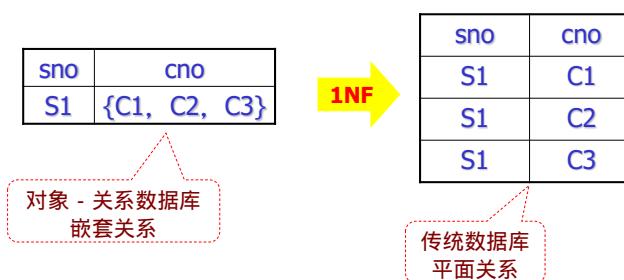


图 4.2: 1NF

**注** 1NF 与应用对属性粒度的处理需求有关.

较细的原子粒度有助于标准化，施加约束避免输入错误，从而提高数据质量。

**1NF 关系模式的不良特性**, 我们考察表4.2.

1. 插入异常: 如果学生没有选课, 关于他的个人信息及所在系的信息就无法插入。
2. 删除异常: 如果删除学生的选课信息, 则他的个人信息及所在系的信息也随之删除。

3. 更新异常: 如果学生转系, 若他选修了  $k$  门课, 则需要修改  $k$  次。

4. 数据冗余: 如果一个学生选修了  $k$  门课, 则有关他的所在系的信息重复  $k$  次。

这些不良特性意味着非主属性对码存在着部分依赖:  $(sno, cno) \xrightarrow{p} sname, (sno, cno) \xrightarrow{p} dno, (sno, cno) \xrightarrow{p} dean$ .

从而我们提出了 2NF 来消除非主属性对码的部分依赖。

#### 4.4.2 2NF

##### 定义 4.13 (2NF)

若  $R \in 1NF$ , 且每个非主属性完全依赖于码, 则称  $R \in 2NF$ .



**注** 之前的表4.2存在非主属性对码的部分依赖, 不是 2NF.

**例题 4.4** 关系模式  $R(A, B, C, D)$ , 给出它的一个函数依赖集, 使得码为  $AB$ , 并且  $R$  属于 1NF 而不属于 2NF.

$\{AB \rightarrow C, A \rightarrow D\}$ .

如何把关系模式改进到 2NF? 把非主属性划分为两部分, 一种是完全依赖于码, 一种是部分依赖于码.

那么我们有:  $S_D(sno, sname, dno, dean), S_C(sno, cno, grade)$ .

但是在关系模式  $S_D$  中存在着  $sno \rightarrow dno, dno \rightarrow dean$ , 这就会导致学生和系主任被关联在一起了, 不合理的.

因而我们有 3NF: 消除非主属性对码的传递依赖.

#### 4.4.3 3NF

##### 定义 4.14 (3NF)

关系模式  $R(U, F)$  中, 若不存在这样的码  $X$ , 属性组  $Y$  及非主属性  $Z$  ( $Z \not\subseteq Y$ ), 使得下式成立:

$$X \rightarrow Y, Y \rightarrow Z, Y \not\rightarrow X$$

则称  $R \in 3NF$ .



上面的  $S_D \notin 3NF$ .

**例题 4.5** 关系模式  $R(A, B, C, D)$ , 给出它的一个函数依赖集, 使得码为  $AB$ , 并且  $R$  属于 2NF 而不属于 3NF.

$\{AB \rightarrow C, C \rightarrow D\}$ .

如何将关系改进到 3NF? 碎断函数依赖的传递链.  $R(ABC, \{A \rightarrow B, B \rightarrow C\})$  分解为  $R_1(AB, \{A \rightarrow B\})$  和  $R_2(BC, \{B \rightarrow C\})$ .

**注** 一个全是主属性的关系模式最高一定可以达到 3NF. 3NF 的目的是为了消除非主属性的冗余.

3NF 的问题: 主属性对码的不良依赖.

**例题 4.6** 考虑  $STC(sno, tno, cno)$ , 我们有:

1. 每位老师只教授一门课:  $tno \rightarrow cno$ .
2. 某学生选定一门课, 就对应一位老师:  $(sno, cno) \rightarrow tno$ .

从而候选码为:  $(sno, tno)$  或  $(sno, cno)$ .

一旦没有同学选修, 无法保存一个老师的授课信息.

所以我们考虑 BCNF: 所有属性都由码直接决定.

#### 4.4.4 BCNF

##### 定义 4.15 (BCNF)

关系模式  $R(U, F)$  中, 对于属性组  $X, Y$ , 若  $X \rightarrow Y (Y \not\subseteq X)$ , 那么  $X$  必是码, 则  $R \in \text{BCNF}$ .



**BCNF:** 所有属性都由码直接决定.

$STC \notin \text{BCNF}$ , 因为  $t_{no} \rightarrow c_{no}$ , 但是  $t_{no}$  不是码.

如何将关系模式改造成 BCNF 的? 将属性划归到以决定它的属性作为码的关系模式中.

sno	tno	cno
s1	t1	c1
s2	t2	c2
s3	t3	c2
s3	t1	c1
	t4	c1

sno	tno
s1	t1
s2	t2
s3	t3
s3	t1

tno	cno
t1	c1
t2	c2
t3	c2
t4	c1

图 4.3: 把关系模式改造为 BCNF

**例题 4.7** 设  $(sno, cno, order)$  表示学生选课的名次, 假设存在函数依赖  $(sno, cno) \rightarrow order, (cno, order) \rightarrow sno$ , 请问它属于 BCNF 吗?

首先不存在对码的传递依赖和部份依赖, 是 3NF. 同时非平凡依赖左边一定是码也是对的. 所以是 BCNF.

##### 定义 4.16 (3NF)

关系模式  $R$  中的函数依赖  $X \rightarrow Y$ , 满足下述条件之一:

- $X \rightarrow Y$  是平凡的函数依赖.
- $X$  是  $R$  的码.
- $Y$  是主属性.



3NF vs. BCNF 存储成本与性能的平衡: 现代存储成本较低, 冗余带来的空间问题可能不如查询性能重要.

#### 4.4.5 多值依赖

##### 定义 4.17 (多值依赖的描述型定义)

对于关系模式  $R(U), X, Y, Z \subseteq U, Z = U - X - Y$ . 多值依赖  $X \rightarrow\rightarrow Y$  成立当且仅当: 对  $R(U)$  的任一关系  $r$ , 给定一对  $(x, z)$  值对应有一组  $Y$  的值, 这组  $Y$  值仅仅决定于  $x$  值而与  $z$  值无关. 也就是  $\forall x, z, Y_{xz} = Y_x$ .



$cno \rightarrow\rightarrow tno$

cno	tno	bno
C1	T1	B1
C1	T1	B2
C1	T2	B1
C1	T2	B2

图 4.4: 多值依赖的例子

##### 定义 4.18 (多值依赖的形式化定义)

关系模式  $R(U), X, Y, Z \subseteq U, Z = U - X - Y$ . 对  $R(U)$  的任一关系  $r$ , 若存在行  $t_1, t_2$ , 使得  $t_1[X] = t_2[X]$ , 那么就必然存在行  $t_3, t_4$ , 使得:

$$t_3 = (t_1[X], t_1[Y], t_2[Z])$$

$$t_4 = (t_2[X], t_2[Y], t_1[Z])$$

则称  $Y$  多值依赖于  $X$ , 记作  $X \rightarrow\rightarrow Y$ .



$t_1$	<table border="1"><tr><td>c1</td><td>t1</td><td>b1</td></tr></table>	c1	t1	b1	$t_3$	<table border="1"><tr><td>c1</td><td>t1</td><td>b2</td></tr></table>	c1	t1	b2
c1	t1	b1							
c1	t1	b2							
$t_2$	<table border="1"><tr><td>c1</td><td>t2</td><td>b2</td></tr></table>	c1	t2	b2	$t_4$	<table border="1"><tr><td>c1</td><td>t2</td><td>b1</td></tr></table>	c1	t2	b1
c1	t2	b2							
c1	t2	b1							

图 4.5: 多值依赖的形式化定义

多值依赖的基本性质:

1. 多值依赖具有对称性: 若  $X \rightarrow\rightarrow Y$ , 则  $X \rightarrow\rightarrow Z$ , 其中  $Z = U - X - Y$ .
2. 函数依赖是多值依赖的特例. 若  $X \rightarrow Y$ , 则  $X \rightarrow\rightarrow Y$ .
3. 平凡的多值依赖. 若  $X \rightarrow\rightarrow Y$ ,  $U - X - Y = \emptyset$ , 称  $X \rightarrow\rightarrow Y$  为平凡的多值依赖.

多值依赖与函数依赖有效性范围的不同:

- $X \rightarrow Y$  的有效性仅决定于  $X, Y$  属性集上的值. 它在任何属性集  $W (XY \subseteq W \subseteq U)$  上都成立.
- $X \rightarrow\rightarrow Y$  在属性集  $W (XY \subseteq W \subseteq U)$  上成立, 但在  $U$  上不一定成立.

#### 定义 4.19 (嵌入式多值依赖)

若  $X \rightarrow\rightarrow Y$  在属性集  $W (XY \subseteq W \subseteq U)$  上成立, 则称  $X \rightarrow\rightarrow Y$  为  $R(U)$  的嵌入式多值依赖.



注  $X \rightarrow\rightarrow Y$  在  $U$  上成立  $\Rightarrow X \rightarrow\rightarrow Y$  在属性集  $W (XY \subseteq W \subseteq U)$  上成立. 这是全集!  $A \rightarrow\rightarrow B$  在  $ABCD$  上成立, 则在  $ABC$  上也成立.

若  $X \rightarrow\rightarrow Y$  在  $R(U)$  上成立, 则对于  $\forall Y' \subseteq Y$ , 不能确定  $X \rightarrow\rightarrow Y'$  是否成立.  $A \rightarrow\rightarrow BC$  成立,  $A \rightarrow\rightarrow B$  未必成立.

多值依赖可以保证无损连接:  $A \rightarrow\rightarrow B, A \rightarrow\rightarrow C \Leftrightarrow r = \Pi_{AB}(r) \bowtie \Pi_{AC}(r)$ .

#### 定理 4.1 (多值依赖成立)

$A \rightarrow\rightarrow B$  成立当且仅当  $R = \Pi_{AB}(R) \bowtie \Pi_{AC}(R)$ .



### 4.4.6 4NF

#### 定义 4.20 (4NF)

关系模式  $R(U) \in 1NF$ , 对于非平凡的多值依赖  $X \rightarrow\rightarrow Y (Y \not\subseteq X)$ ,  $X$  含有码, 则称  $R \in 4NF$ .



非 4NF 的主要弊端: 冗余大. 如果一门课  $c_i$  有  $m$  个教员,  $n$  本参考书, 则  $c_i$  在关系中一共有  $mn$  行.

如何将关系模式改造为 4NF 的? 多值属性单独放在独立的关系模式中.

### 4.4.7 PJNF

#### 定义 4.21 (连接依赖)

$R_1(U_1), R_2(U_2), \dots, R_n(U_n)$  是  $R(U)$  的一个分解,  $r$  是  $R(U)$  上的一个关系, 若  $r = \bowtie_{i=1}^n \Pi_{R_i}(r)$ , 则称  $r$  满足连接依赖  ${}^*(R_1, R_2, \dots, R_n)$ .



连接依赖  ${}^*(R_1, R_2, \dots, R_n)$  中, 若有某个  $R_i$  等于  $R$ , 则称之为平凡的连接依赖.

连接依赖  ${}^*(R_1, R_2)$  等价于多值依赖  $R_1 \cap R_2 \rightarrow\rightarrow R_1, \alpha \rightarrow\rightarrow \beta \Leftrightarrow {}^*(\alpha \cup (R - \beta), \alpha \cup \beta)$ .

**定义 4.22 (PJNF)**

若  $R \in \text{PJNF}$ , 则对于  $R$  的任一连接依赖  ${}^*(R_1, R_2, \dots, R_n)$  必是下述情况之一:

1.  ${}^*(R_1, R_2, \dots, R_n)$  是平凡的连接依赖
2. 每个  $R_i$  是  $R$  的超码



## 4.5 Armstrong 公理系统

**定义 4.23 (逻辑蕴含)**

关系模式  $R(U, F)$ ,  $F$  是其函数依赖集,  $X, Y \subseteq U$ . 如果从  $F$  的函数依赖能够推出  $X \rightarrow Y$ , 则称  $F$  逻辑蕴含  $X \rightarrow Y$ , 记作  $F \vdash X \rightarrow Y$ .

**定义 4.24 (闭包)**

被  $F$  所逻辑蕴含的函数依赖的全体所构成的集合称作  $F$  的闭包, 记作  $F^+ = \{X \rightarrow Y \mid F \vdash X \rightarrow Y\}$ .

**定理 4.2 (Armstrong 公理系统)**

- 自反律 (reflexivity): 若  $Y \subseteq X$ , 则  $X \rightarrow Y$ .
- 增广律 (augmentation): 若  $X \rightarrow Y$ , 则  $XZ \rightarrow YZ$ .
- 传递律 (transitivity): 若  $X \rightarrow Y, Y \rightarrow Z$ , 则  $X \rightarrow Z$ .

**定理 4.3 (正确性)**

设  $A = \{f \mid$  用 Armstrong 公理系统从  $F$  中导出的函数依赖  $f\}$ ,  
 $B = \{f \mid$  被  $F$  所逻辑蕴含的函数依赖  $f\}$ . 那么正确性就是:  $A \subseteq B$ .



**证明** 设  $r$  是  $R(U, F)$  上的任一关系,  $t, s \in r$ .

1. 检查自反律. 现在我们设  $t[X] = s[X]$ , 由于  $Y \subseteq X$ , 那么  $t[Y] = s[Y]$ , 那么也就是  $X \rightarrow Y$ . 也就是  $X \rightarrow Y$  可以被  $F$ (其实  $\emptyset$  也可以蕴含出) 所逻辑蕴含.
2. 检查增广律. 现在我们设  $t[XZ] = s[XZ]$ , 那么  $t[X] = s[X]$ . 结合上  $X \rightarrow Y$ , 那么有  $t[Y] = s[Y]$ . 同时  $t[XZ] = s[XZ]$ , 得到  $t[Z] = s[Z]$ . 最后结合  $t[Y] = s[Y], t[Z] = s[Z]$ , 得到  $t[YZ] = s[YZ]$ , 从而得到  $XZ \rightarrow YZ$ .
3. 检查传递律. 现在我们设  $t[X] = s[X]$ , 由于  $X \rightarrow Y$ , 得到  $t[Y] = s[Y]$ . 由于  $Y \rightarrow Z$ , 得到  $t[Z] = s[Z]$ . 这样就得到了  $X \rightarrow Z$ .

综上所述, Armstrong 公理系统的正确性得证.

下面是由 Armstrong 公理系统推导出的推理规则:

1. 合并律 (union rule): 若  $X \rightarrow Y, X \rightarrow Z$ , 则  $X \rightarrow YZ$ .
2. 分解律 (decomposition rule): 若  $X \rightarrow YZ$ , 则  $X \rightarrow Y, X \rightarrow Z$ .<sup>1</sup>
3. 伪传递律 (pseudotransitivity rule): 若  $X \rightarrow Y, WY \rightarrow Z$ , 则  $WX \rightarrow Z$ .

<sup>1</sup>一个更强的推论: 若  $X \rightarrow A_1A_2\dots A_n$ , 则  $X \rightarrow A_i$ .

## 4.6 闭包计算

### 定义 4.25 (属性集的闭包)

设  $F$  为属性集  $U$  上的一组函数依赖,  $X \subseteq U$ ,

$$X_F^+ = \{A | X \rightarrow A \text{ 能由 } F \text{ 根据 Armstrong 公理系统推出}\},$$

称  $X_F^+$  为属性集  $X$  关于函数依赖集  $F$  的闭包.



### 算法 4.1: 属性集 $X$ 关于函数依赖集 $F$ 的闭包 $X_F^+$ 的计算

**Input:** 属性集  $X$ , 函数依赖集  $F$

**Output:**  $X_F^+$

```

1  $X_F^+ \leftarrow X;$ 
2 repeat
3   foreach 函数依赖  $A \rightarrow B \in F$  do
4     if  $A \subseteq X_F^+$  then
5        $X_F^+ \leftarrow X_F^+ \cup B;$ 
6     end
7   end
8 until  $X_F^+$  不再发生变化;

```

算法4.1的正确性:

$$A \subseteq X_F^+ \Rightarrow X \rightarrow A, A \rightarrow B \Rightarrow X \rightarrow B \Rightarrow B \in X_F^+.$$

算法最多  $|U - X|$  步终止 (每次都加入一个属性).

### 定理 4.4 (闭包的封闭性)

$$(X^+)^+ = X^+.$$



**证明** 我们设  $A \in (X^+)^+$ , 那么就有  $X^+ \rightarrow A$ , 同时我们有  $X \rightarrow X^+$ , 那么就有  $X \rightarrow A$ . 那么  $A \in X^+$ , 这样就导出了  $(X^+)^+ \subseteq X^+$ . 同时原本就有  $X^+ \subseteq (X^+)^+$ , 那么  $(X^+)^+ = X^+$ .

### 定义 4.26 (属性集的封闭性)

如果  $X^+ = X$ , 则称  $X$  是封闭的.



如何判断  $X \rightarrow Y$  是否可以由 Armstrong 公理系统导出?

1. 计算出  $F^+$ , 再判断  $X \rightarrow Y$  是否属于  $F^+$ . 计算很复杂!
2. 判断  $Y \subseteq X_F^+$  是否成立. 简单. 判断依据来自于下面的定理.

### 定理 4.5

$$X \rightarrow Y \text{ 能由 Armstrong 公理系统导出} \Leftrightarrow Y \subseteq X_F^+.$$



同时我们现在也可以借助属性集的闭包来说明 Armstrong 公理系统的完备性.

### 定理 4.6 (完备性)

设  $A = \{f | \text{用 Armstrong 公理系统从 } F \text{ 中导出的函数依赖 } f\}$ ,

$B = \{f | \text{被 } F \text{ 所逻辑蕴含的函数依赖 } f\}$ . 那么正确性就是:  $B \subseteq A$ .



**证明** 我们使用 反证法.

若存在函数依赖  $X \rightarrow Y$  被  $F$  逻辑蕴含, 但  $X \rightarrow Y$  不能用 Armstrong 公理系统从  $F$  中导出.

则存在  $Y$  的子集不属于  $X$  的闭包, 也即  $Y - X_F^+ \neq \emptyset, U - X_F^+ \neq \emptyset$ .

下面我们构造一个  $R(U)$  上的关系  $r$ :

$r$	$X_F^+$	$U - X_F^+$
$t$	1	0
$s$	1	1

下面证明和我们的假设互相矛盾的两条结论:

1.  $r$  满足  $F$ .

设  $W \rightarrow V$  是  $F$  中的任一个函数依赖. 我们设  $t[W] = s[W]$ , 那么  $W \subseteq X_F^+$ , 在另一边不可能会有  $t[W] = s[W]$ . 这样就得到  $X \rightarrow W$ , 利用传递性得到  $X \rightarrow V$ , 从而  $V \subseteq X_F^+$ , 从而  $t[V] = s[V]$ . 所以  $r$  满足函数依赖  $W \rightarrow V$ , 也即  $r$  是  $R(U, F)$  上的关系.

2.  $r$  不满足  $X \rightarrow Y$ .

$Y - X_F^+ \neq \emptyset$ , 所以存在  $A \notin X_F^+$ . 但是有  $t[X] = s[X], t[A] \neq s[A]$ , 所以  $t[Y] \neq s[Y]$ , 也即  $X \rightarrow Y$  不成立. 那么假设就不成立, 完备性得到证明.

## 4.7 候选码计算

### 定义 4.27 (左部属性)

左部属性, 只出现在  $F$  左边的属性.



### 定义 4.28 (右部属性)

右部属性, 只出现在  $F$  右边的属性.



### 定义 4.29 (双部属性)

双部属性, 出现在  $F$  两边的属性.



### 定义 4.30 (外部属性)

外部属性, 不出现在  $F$  中的属性.



1. 左部属性一定出现在任何候选码中.

2. 右部属性一定不出现在任何候选码中.

3. 外部属性一定出现在任何候选码中.

候选码的构成: 左部属性 + 外部属性 + [可能出现的双部属性].

**例题 4.8** 设  $U = \{C, T, H, R, S\}, F = \{C \rightarrow T, HR \rightarrow C, HT \rightarrow R, HS \rightarrow R\}$ , 给出  $R$  的所有候选码, 判断其范式级别.

左部属性为  $\{H, S\}$ , 双部属性为  $\{C, T, R\}$ . 注意到  $(HS)_F^+ = HSRCT$ , 所以候选码为  $HS$ .

**算法 4.2:** 寻找一个候选码的一般算法

---

**Input:** 关系模式  $R(U, F)$   
**Output:** 一个候选码  $K$

```

1  $K := U;$ 
2 构造一个 FD:  $K \rightarrow T$ , 其中  $T \notin U$ ;
3 for 每一个属性  $A \in K$  do
4   if  $\text{Membership}(F \cup \{K \rightarrow T\}, (K - A) \rightarrow T)$  then
5      $K := K - A;$ 
      $\triangleright \text{Membership}(F, X \rightarrow Y)$  判断是否有  $X \rightarrow Y \in F^+$ 
6   end
7 end
8 return( $K$ );

```

---

**算法 4.3:** 寻找全部候选码的算法

---

**Input:** 关系模式  $R(U, F)$   
**Output:**  $R(U, F)$  的全部候选码

```

1  $K :=$  找到一个候选码;
2  $K$  入队列  $Q$ ;
3 while 队列  $Q \neq \emptyset$  do
4    $K :=$  队列  $Q$  的头;
5    $W := W \cup \{K\}$ ;
6    $D := K$  中全部双部属性;
7   while  $D \neq \emptyset$  do
8      $A := D$  中一个双部属性;
9      $D := D - \{A\}$ ;
10    for 每一个  $X \rightarrow Y \in F$  do
11      if  $A \in Y$  then
12         $K' := (K - A) \cup \{X\}$ ;
13        if  $K' \notin Q$  then
14           $K'$  入队列  $Q$ ;
15        end
16      end
17    end
18  end
19 end
20 return( $W$ );

```

---

## 4.8 函数依赖的等价和覆盖

**定义 4.31 (函数依赖集的等价性)**

对于函数依赖集  $F, G$ , 若  $F^+ = G^+$ , 则称  $F$  与  $G$  等价.



定义4.31实际上要求我们检验:  $F \subseteq G^+ \wedge G \subseteq F^+$ .

**定义 4.32 (函数依赖集的最小覆盖)**

对于函数依赖集，它的最小覆盖  $F$  满足下面的三个条件：

1. **单属性化**. 对于  $F$  中任一函数依赖  $X \rightarrow A$ ,  $A$  必是单属性.
2. **无冗余化**.  $F$  中不存在这样的函数依赖  $X \rightarrow A$ , 使得  $F$  与  $F - \{X \rightarrow A\}$  等价.
3. **既约化**.  $F$  中不存在这样的函数依赖  $X \rightarrow A$ , 在  $X$  中有真子集  $Z$ , 使得  $F$  与  $F - \{X \rightarrow A\} \cup \{Z \rightarrow A\}$  等价.



下面给出求解函数依赖集  $F$  的最小覆盖  $F_{min}$  的算法：

1. **单属性化**: 逐个检查  $F$  中各函数依赖  $FD_i : X \rightarrow Y$ , 若  $Y = A_1A_2\dots A_k$ , 则用诸  $X \rightarrow A_i$  代替  $Y$ .
2. **无冗余化**: 逐个检查  $F$  中各函数依赖  $X \rightarrow A$ , 令  $G = F - \{X \rightarrow A\}$ , 若  $A \in X_G^+$ , 则从  $F$  中去掉该函数依赖.
3. **既约化**: 逐个检查  $F$  中各函数依赖  $X \rightarrow A$ , 设  $X = B_1B_2\dots B_m$ , 逐个考察  $B_i$ , 若  $A \in (X - B_i)_F^+$ , 则  $X - B_i$  取代  $X$ .

**例题 4.9** 已知  $F = \{A \rightarrow B, B \rightarrow A, A \rightarrow C, B \rightarrow C\}$ , 求  $F_{min}$ .

检查  $A \rightarrow B$ ,  $G = F - \{A \rightarrow B\} = \{B \rightarrow A, A \rightarrow C, B \rightarrow C\}$

$A_G^+ = \{A, C\} \Rightarrow B \notin A_G^+ \Rightarrow A \rightarrow B \notin G^+$ .

检查  $A \rightarrow C$ ,  $G = F - \{A \rightarrow C\} = \{A \rightarrow B, B \rightarrow A, B \rightarrow C\}$

$A_G^+ = \{A, B, C\} \Rightarrow C \in A_G^+ \Rightarrow A \rightarrow C \in G^+$ .

$F_{min} = \{A \rightarrow B, B \rightarrow A, B \rightarrow C\}$  或者  $F_{min} = \{A \rightarrow B, B \rightarrow A, A \rightarrow C\}$ .

## 4.9 函数依赖和多值依赖的推理规则

1. 自反律: 若  $Y \subseteq X$ , 则  $X \rightarrow Y$ .
2. 增广律: 若  $X \rightarrow Y$ , 则  $XZ \rightarrow YZ$ .
3. 传递律: 若  $X \rightarrow Y, Y \rightarrow Z$ , 则  $X \rightarrow Z$ .
4. 复制律: 若  $X \rightarrow Y$ , 则  $X \rightarrow \rightarrow Y$ .
5. 补充律: 若  $X \rightarrow \rightarrow Y$ , 则  $X \rightarrow \rightarrow R - X - Y$ .
6. 多值增广律: 若  $X \rightarrow \rightarrow Y, Z \subseteq R, W \subseteq Z$ , 则  $XZ \rightarrow \rightarrow YW$ .
7. 多值传递律: 若  $X \rightarrow \rightarrow Y, Y \rightarrow \rightarrow Z$ , 则  $X \rightarrow \rightarrow Z - Y$ .
8. 联合律: 若  $X \rightarrow \rightarrow Y, Z \subseteq Y$ , 且存在  $W$ , 使得  $W \subseteq R, W \cap Y = \emptyset, W \rightarrow Z$ , 则  $X \rightarrow Z$ .

## 4.10 模式分解

**定义 4.33 (函数依赖在属性集上的投影)**

函数依赖集  $F$  在属性集  $U_i$  上的投影定义为：

$$F_i = \{X \rightarrow Y | X \rightarrow Y \in F^+ \wedge XY \subseteq U_i\}.$$



**注** 要判断  $X \rightarrow Y \in F^+$ , 只需要判断  $Y \in X_F^+$ .

**例题 4.10** 求  $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow D\}$  在  $S(ACD)$  上的投影.

首先计算出:  $A_F^+ = ABCD, C_F^+ = CD, D_F^+ = D$ . 那么从而得到投影为  $\{A \rightarrow C, A \rightarrow D, C \rightarrow D\}$ . 现在考虑右侧可以组合成的  $CD$ ,  $(CD)_F^+ = CD$ , 从而投影确实为  $\{A \rightarrow C, A \rightarrow D, C \rightarrow D\}$ .

**例题 4.11** 计算下面的函数依赖集在  $S(ABC)$  上的投影:

1.  $F = \{AB \rightarrow DE, C \rightarrow E, D \rightarrow C, E \rightarrow A\}$ ;
2.  $F = \{A \rightarrow D, BD \rightarrow E, AC \rightarrow E, DE \rightarrow B\}$ ;

3.  $F = \{AB \rightarrow D, AC \rightarrow E, BC \rightarrow D, D \rightarrow A, E \rightarrow B\}$ .

#### 定义 4.34 (模式分解)

关系模式  $R(U, F)$  的一个分解是指

$$\rho = \{R_1(U_1, F_1), R_2(U_2, F_2), \dots, R_n(U_n, F_n)\},$$

其中  $U = \bigcup_{i=1}^n U_i$ , 并且没有  $U_i \subseteq U_j$ ,  $1 \leq i, j \leq n$ ,  $F_i$  是  $F$  在  $U_i$  上的投影.



### 4.10.1 保持函数依赖分解

#### 定义 4.35 (保持函数依赖分解)

设关系模式  $R(U, F)$  的一个分解是

$$\rho = \{R_1(U_1, F_1), R_2(U_2, F_2), \dots, R_n(U_n, F_n)\},$$

如果  $F^+ = (\bigcup_{i=1}^n F_i)^+$ , 则称  $\rho$  是保持函数依赖的分解.



保持函数依赖  $\Leftrightarrow F^+ = (\bigcup_{i=1}^n F_i)^+ \Leftrightarrow F \subseteq (\bigcup_{i=1}^n F_i)^+ \wedge F_i \subseteq F^+$ .

### 4.10.2 保持无损连接分解

#### 定义 4.36 (无损连接分解)

关系模式  $R(U, F)$ ,  $U = \bigcup_{i=1}^n U_i$ ,  $\rho = \{R_1(U_1, F_1), R_2(U_2, F_2), \dots, R_n(U_n, F_n)\}$ ,  $r$  是  $R$  的任意一个关系实例, 定义

$$m_\rho(r) = \bowtie_{i=1}^n \Pi_{U_i}(r).$$

若  $m_\rho(r) = r$ , 则称  $\rho$  是  $R$  的一个无损分解.

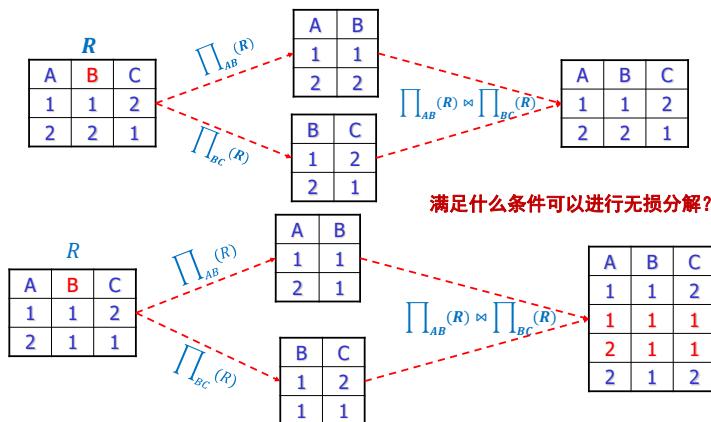


图 4.6: 无损分解的例子

无损连接分解的判别算法.

对于  $U = \{A_1, A_2, \dots, A_n\}$ ,  $\rho = \{R_1(U_1, F_1), R_2(U_2, F_2), \dots, R_k(U_k, F_k)\}$ .

1. 建立一个  $k$  行  $n$  列的矩阵(行是子模式  $U_i$ , 列是  $U$  中属性  $A_j$ ), 其中:

$$TB = \{C_{ij} | \text{若 } A_j \in U_i, C_{ij} = a_j, \text{ 否则 } C_{ij} = b_{ij}\}.$$

2. 对  $F$  中的每一个函数依赖  $X \rightarrow Y$ , 若  $TB$  中存在元组  $t_1, t_2$ , 使得  $t_1[X] = t_2[X], t_1[Y] \neq t_2[Y]$ . 则对每一个  $A_i \in Y$ :

(a). 若  $t_1[A_i], t_2[A_i]$  中有一个等于  $a_i$ , 则另一个也改为  $a_i$ ;

(b). 若 (a). 不成立, 则取  $t_1[A_i] = t_2[A_i]$  ( $t_2$  的行号小于  $t_1$ ).

3. 反复执行 2., 直至:

(a). TB 中出现一行全为  $a_1, a_2, \dots, a_n$  的一行, 此时  $\rho$  为无损分解;

(b). TB 不再发生变化, 且没有一行为  $a_1, a_2, \dots, a_n$ , 此时  $\rho$  为有损分解.

例题 4.12 判断下面的分解是否是无损分解.

$$U = \{A, B, C, D, E\}, F = \{AB \rightarrow C, C \rightarrow D, D \rightarrow E\}, \rho = \{ABC, CD, DE\}.$$

图 4.7: 无损连接分解判别算法示例

例题 4.13 判断下面的分解是否是无损分解.

$$U = \{A, B, C, D, E\}, F = \{A \rightarrow C, B \rightarrow C, C \rightarrow D, DE \rightarrow C, CE \rightarrow A\}, \rho = \{AD, AB, BE, CDE, AE\}.$$

图 4.8: 无损连接分解判别算法示例

分解为两个关系模式的无损分解判定算法:

$$\rho = \{U_1, U_2\} \text{ 是无损连接分解} \Leftrightarrow U_1 \cap U_2 \rightarrow U_1 - U_2 \text{ 或者 } U_1 \cap U_2 \rightarrow U_2 - U_1.$$

### 4.10.3 关系模式分解算法

#### 4.10.3.1 达到 BCNF 无损连接分解算法

给定关系模式  $R(U, F)$

1. 令  $\rho = R(U, F)$ ;
2. 检查  $\rho$  中各关系模式是否属于 BCNF, 若是, 则算法终止;

3. 设  $\rho$  中  $R_i(U_i, F_i)$  不属于 BCNF.

则存在函数依赖  $X \rightarrow A \in F_i^+$ , 且  $X$  不是  $R_i$  的码.

我们将  $R_i$  分解为  $\sigma = \{S_1(U_1), S_2(U_2)\}$ , 其中  $U_1 = XA, U_2 = U_i - A$ , 我们以  $\sigma$  代替  $R_i$ , 返回到 2.

### 定理 4.7

上述算法得到的分解是无损连接分解.



**证明** 上述算法中出现的分解操作在第 3 步, 只要证明这个分解是无损连接分解, 那么整个算法得到的分解都是无损连接分解.

根据判断分解为两个关系模式的无损分解判定方法, 我们发现:  $U_1 \cap U_2 = X, U_1 - U_2 = A$ , 而已知  $X \rightarrow A$ , 那么就有  $U_1 \cap U_2 \rightarrow U_1 - U_2$ , 从而是无损连接分解.

那么上述算法得到的是无损连接分解.

### 定理 4.8

上述算法分解得到的每个关系模式都是 BCNF 的.



**证明** 因为每次都会进行一次分解, 那么至多进行  $|F^+|$  次分解, 最后得到的一定是 BCNF.

**例题 4.14**  $U = \{A, B, C, D, E\}, F = \{A \rightarrow B, B \rightarrow C, AD \rightarrow E\}$ .

码是  $AD, A \rightarrow B, B \rightarrow C$  违反了 BCNF.

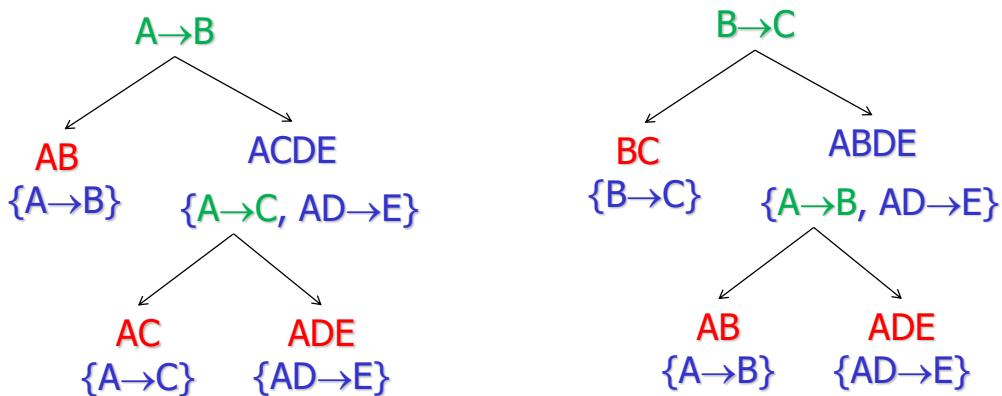


图 4.9: 达到 BCNF 无损连接分解算法

**例题 4.15** 如何构造一个有  $N$  种 BCNF 分解结果的关系模式?

$R(A_0 A_1 \dots A_n; \{A_0 \rightarrow A_n, A_1 \rightarrow A_n, A_2 \rightarrow A_n, \dots, A_{n-1} \rightarrow A_n\})$ .

**例题 4.16**  $R(A_1 A_2 \dots A_n; \{A_1 \rightarrow A_2, A_2 \rightarrow A_3, \dots, A_{n-1} \rightarrow A_n\})$  有多少种 BCNF 分解结果?

可以把 BCNF 分解算法改进为: 使用  $X^+$ .

若要求分解保持函数依赖, 那么分解后的模式总可以达到 3NF, 但不一定能达到 BCNF.

### 4.10.3.2 达到 4NF 无损连接分解算法

给定关系模式  $R(U, F)$ ,

1. 令  $\rho = R(U, F)$ ;
2. 检查  $\rho$  中各关系模式是否属于 4NF, 若是, 则算法终止;
3. 设  $\rho$  中  $R_i(U_i, F_i)$  不属于 4NF,

存在非平凡多值依赖  $X \rightarrow\rightarrow A$ , 且  $X$  不是  $R_i$  的码,

将  $R_i$  分解为  $\sigma = \{S_1(U_1), S_2(U_2)\}$ , 其中  $U_1 = XA, U_2 = U_i - A$ ,

以  $\sigma$  代替  $R_i$ , 返回到 2.

例题 4.17  $U = \{A, B, C, D, E, G\}$ ,  $F = \{A \rightarrow BCG, B \rightarrow AC, C \rightarrow G\}$ , 码为  $BDE$ .

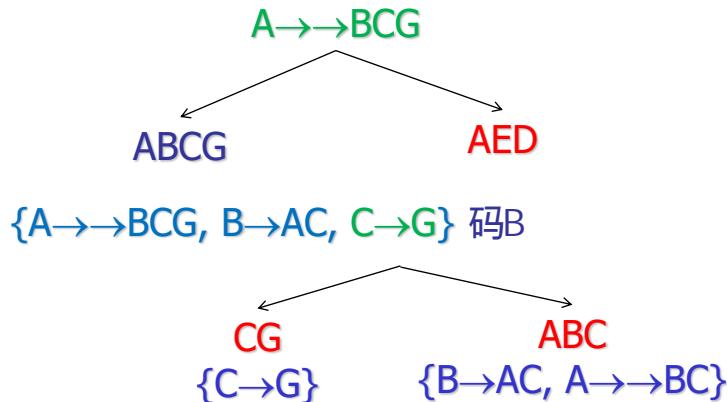


图 4.10: 达到 4NF 无损连接分解算法

#### 4.10.3.3 达到 3NF 保持函数依赖的分解

1. 求  $F$  的最小覆盖  $F_{min}$ ;
2. 找出不在  $F_{min}$  中出现的属性, 将它们构成一个关系模式, 并从  $U$  中去掉它们(剩余属性仍记为  $U$ );
3. 若有  $X \rightarrow A \in F_{min}$ , 且  $XA = U$ ,  $\rho = \{R\}$ , 算法终止;
4. 对  $F_{min}$  按具有相同左部的原则进行分组(设为  $k$  组), 每一组函数依赖所涉及的属性全体为  $U_i$ , 令  $F_i$  为  $F_{min}$  在  $U_i$  上的投影, 则  $\rho = \{R_1(U_1, F_1), R_2(U_2, F_2), \dots, R_k(U_k, F_k)\}$  是  $R(U, F)$  的一个保持函数依赖的分解, 并且每个  $R_i(U_i, F_i) \in 3NF$ .  
 » 3NF 分解算法的第 3 步, 如何确定此时关系模式已经是 3NF 的了?  
 » 此时第 2 步去掉的属性不依赖留下的属性, 互相也不依赖, 必然是 3NF 的. 而且第 3 步  $X \rightarrow A \in F_{min}$ , 而  $F_{min}$  中的依赖是单属性化和无冗余化的, 从而属性  $A$  不可能传递依赖于  $X$ (否则就会违反了无冗余的性质), 同时  $X$  中的属性都是主属性, 所以这时关系模式已经是 3NF 的了.

#### 4.10.3.4 同时保持函数依赖和无损连接的分解算法

设  $\rho = \{R_1(U_1, F_1), R_2(U_2, F_2), \dots, R_k(U_k, F_k)\}$  是  $R(U, F)$  的一个保持函数依赖的 3NF 分解, 设  $\text{X}$  是  $R(U, F)$  的码.

设若有某个  $U_i$ ,  $\text{X} \subseteq U_i$ , 则  $\rho$  即为所求; 否则令  $\tau = \rho \cup \{R^*(X, F_X)\}$ ,  $\tau$  即为所求.

##### 定义 4.37 (悬挂元组)

$R$  分解为  $R_1, R_2, \dots, R_n$ ,  $r_1 \bowtie r_2 \bowtie \dots \bowtie r_n$  称为泛关系.

在  $r_i$  中出现, 但是在  $\Pi_{R_i}(r_1 \bowtie r_2 \bowtie \dots \bowtie r_n)$  中没有出现的元组, 称为悬挂元组.



悬挂元组代表了不完整的信息.

<table border="1" style="border-collapse: collapse; width: 100%; height: 100%;"> <tr> <th style="width: 50px;">A</th><th style="width: 50px;">B</th></tr> <tr> <td>a1</td><td>b1</td></tr> <tr> <td>a2</td><td>b2</td></tr> <tr> <td>a3</td><td>b3</td></tr> </table>	A	B	a1	b1	a2	b2	a3	b3	$\bowtie$	<table border="1" style="border-collapse: collapse; width: 100%; height: 100%;"> <tr> <th style="width: 50px;">B</th><th style="width: 50px;">C</th></tr> <tr> <td>b1</td><td>c1</td></tr> <tr> <td>b2</td><td>c2</td></tr> </table>	B	C	b1	c1	b2	c2	$=$	<table border="1" style="border-collapse: collapse; width: 100%; height: 100%;"> <tr> <th style="width: 50px;">A</th><th style="width: 50px;">B</th><th style="width: 50px;">C</th></tr> <tr> <td>a1</td><td>b1</td><td>c1</td></tr> <tr> <td>a2</td><td>b2</td><td>c2</td></tr> </table>	A	B	C	a1	b1	c1	a2	b2	c2
A	B																										
a1	b1																										
a2	b2																										
a3	b3																										
B	C																										
b1	c1																										
b2	c2																										
A	B	C																									
a1	b1	c1																									
a2	b2	c2																									

图 4.11: 悬挂元组

## 4.11 模式调优

- 分解通常使得对复杂查询的回答的效率更差,因为在查询求值期间必须执行额外的连接.
- 分解使得对简单查询的回答更有效,因为这种查询通常涉及相同关系的一小部分属性.
- 分解通常使得简单的更新事务更有效.
- 分解能降低存储空间的要求,因为它一般能消除冗余数据.
- 如果冗余级别低,则分解会增加存储的需求.

### 4.11.1 垂直划分

$R(XYZ)$  还是  $R_1(XY)$  和  $R_2(XZ)$ ?

» 一般情况下  $R$  好于  $R_1$  和  $R_2$ ,但是下面的情况除外:

1. 大多数用户的存取分别在两个集合上;
2. 属性  $Y$  和  $Z$  的值占用很大空间.

事务-属性交叉矩阵 (Transaction-Attribute Cross Matrix) 是数据库物理设计与数据挖掘中,用于表示“哪个事务访问了哪些属性”的一种二元 (0/1) 矩阵结构. 该矩阵的行对应系统中的各个事务 (Transaction), 列对应数据的各个属性 (Attribute), 矩阵元素若为 1, 则表示该事务访问 (读或写) 了该属性, 否则为 0. 通过对该矩阵进行聚类或分区, 可以识别在同一事务集中频繁共同访问的属性, 从而指导垂直分区或矩阵聚类, 优化磁盘 I/O 与事务并发性能.

属性关联矩阵 → 属性带权关联图 → 图分割.

# 第五章 事务

## 定义 5.1 (事务)

事务是由一系列操作序列构成的执行单元, 这些操作要么都做, 要么都不做, 是一个不可分割的工作单位.



## 5.1 SQL 中的事务

1. 事务以`begin transaction`开始, 以`commit transaction`或`rollback transaction`结束.
2. `commit transaction`表示提交, 事务正常结束.
3. `rollback transaction`表示事务非正常结束, 撤消事务已做的操作, 回滚到事务开始时状态.

```
create table accounts ( userId char(4) primary key,
                      amounts int check ( amounts >= 0 ) );
insert into accounts values ('A',1000), ('B', 2000);
set transaction isolation level read committed;
start transaction;
update accounts set amounts = amounts - 50 where userId='A';
update accounts set amounts = amounts + 50 where userId='B';
commit;
```

事务的执行模式:

1. 显式事务: 以`begin transaction`开始, 以`commit`或`rollback`结束.
2. 隐含事务 (SQL Server): 事务自动开始, 直到遇到`commit`或`rollback`时结束.  
`set implicit_transactions {ON | OFF}.`
3. 自动事务 (MySQL): 每个数据操作语句作为一个事务. `set autocommit = {1 | 0}.`

事务中的错误检查:

1. SQL Server: `set XACT_ABORT ON.`
2. MySQL: `declare exit handler for SQLEXCEPTION rollback.`

### 5.1.1 事务基本特性 ACID

1. 原子性 (Atomicity).
  - (a). 事务中包含的所有操作要么全做, 要么全不做.
  - (b). 原子性由恢复机制实现.
2. 一致性 (Consistency).
  - (a). 事务的隔离执行必须保证数据库的一致性.
  - (b). 事务开始前, 数据库处于一致性状态; 事务结束后, 数据库必须仍处于一致性状态.
  - (c). 数据库一致性状态由用户来负责.
3. 隔离性 (Isolation).
  - (a). 系统必须保证事务不受其它并发执行事务的影响.
  - (b). 对任何一对事务  $T_1, T_2$ , 在  $T_1$  看来,  $T_2$  要么在  $T_1$  开始之前已经结束, 要么在  $T_1$  完成之后再开始执行:  
 $T_1 \rightarrow T_2$  or  $T_2 \rightarrow T_1$ .
  - (c). 隔离性通过并发机制实现.
4. 持久性 (Durability).
  - (a). 一个事务一旦提交之后, 它对数据库的影响必须是永久的.

(b). 系统发生故障不能改变事务的持久性.

(c). 持久性通过恢复机制实现.

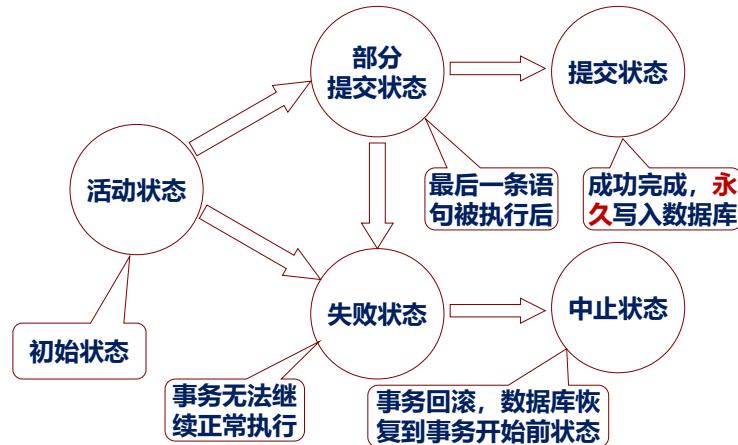


图 5.1: 事务生命周期图

## 5.2 事务调度

### 定义 5.2 (调度)

事务的执行顺序称为一个调度, 表示事务的指令在系统中执行的时间顺序.

一组事务的调度必须保证:

1. 包含了所有事务的操作指令.
2. 一个事务中指令的顺序必须保持不变.



### 定义 5.3 (串行调度)

1. 在串行调度中, 属于同一事务的指令紧挨在一起.
2. 对于有  $n$  个事务的事务组, 可以有  $n!$  个有效调度.



### 定义 5.4 (并行调度)

1. 在并行调度中, 来自不同事务的指令可以交叉执行.
2. 当并行调度等价于某个串行调度时, 则称它是正确的.



**例题 5.1**  $n$  个事务,  $T_i$  有  $k_i$  条指令, 则可能的并发调度有多少个?

调度数为:

$$\frac{(\sum_{i=1}^n k_i)!}{\prod_{i=1}^n k_i!}.$$

### 定义 5.5 (可恢复调度)

对于每对事务  $T_i$  与  $T_j$ , 如果  $T_j$  读取了  $T_i$  所写的数据, 则  $T_i$  必须先于  $T_j$  提交.

1. 一个事务失败了, 应该能够撤消该事务对数据库的影响.
2. 如果有其它事务读取了失败事务写入的数据, 则该事务应该撤消.



### 定义 5.6 (级联调度)

由于一个事务故障而导致一系列事务回滚.



**定义 5.7 (无级连调度)**

对于任意两个事务  $T_i$  和  $T_j$ , 如果  $T_j$  读取了  $T_i$  写入的数据项, 则  $T_i$  的提交操作必须在  $T_j$  的操作之前完成.  
无级联调度必是可恢复调度.



### 5.2.1 并发调度中的不一致现象

丢失修改: 写写不一致. 两个事务  $T_1$  和  $T_2$  读入同一数据并修改,  $T_1$  提交的结果破坏了  $T_2$  提交的结果, 导致  $T_2$  的修改丢失.

读脏数据: 写读不一致. 事务  $T_1$  修改某一数据并将其写回磁盘, 事务  $T_2$  读取同一数据. 此后  $T_1$  由于某种原因被撤消, 其已修改过的数据恢复原值, 造成  $T_2$  读到的数据与数据库中数据不一致, 则  $T_2$  读到的就是脏数据.

不能重复读: 读写不一致. 事务  $T_2$  读取某一数据后, 事务  $T_1$  对其做了修改, 当  $T_2$  再次读取该数据时, 得到与前次不同的值.

发生幻象 (Phantom): 插读不一致. 事务  $T_2$  按一定条件读取某些数据后, 事务  $T_1$  插入一些满足这些条件的数据, 当  $T_2$  再次按相同条件读取数据时, 发现多了一些记录.

解决方案:

1. 丢失修改: 两个事务不能同时修改同一数据项.
2. 读脏数据: 只能读取已提交数据.
3. 不能重复读: 两次读取之间不能有其他事务修改该数据项.
4. 幻象: 两次读取不能插入.

## 5.3 事务隔离性级别

1. read uncommitted: 允许读取未提交的记录.
2. read committed: 只允许读取已提交的记录, 但不要求可重复读.
3. repeatable read: 只允许读取已提交记录, 并且一个事务对同一记录的两次读取之间, 其它事务不能对该记录进行更新.
4. serializable: 调度的执行必须等价于串行调度.

隔离性级别	读脏数据	不能重复读	幻象	丢失修改
Read uncommitted	是	是	是	是
Read committed	否	是	是	是
Repeatable read	否	否	是	否
Serializable	否	否	否	否

表 5.1: 事务隔离性级别

## 5.4 快照隔离

快照隔离 (Snapshot Isolation, SI) 的基本思想: 多版本 + 回滚.

- 对数据库的写发生在提交时, 形成数据项的一个提交版本 (快照).
- 执行时间和访问数据项有交叠的写事务之间会产生冲突, 先提交者赢.
- 读操作访问该读事务开始那一刻的数据项最新版本, 读写相互不会阻塞.

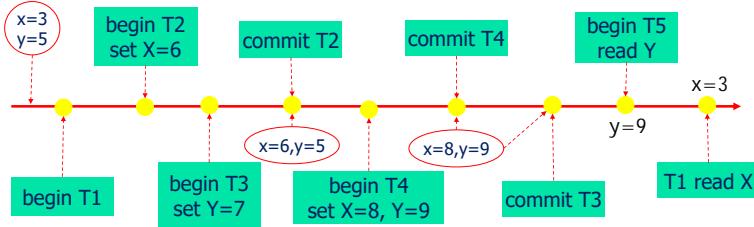


图 5.2: 快照隔离

快照隔离中有不一致的现象.

**例题 5.2** 现在有一致性要求:  $X + Y \geq 0$ , 下面的调度会导致一致性被违反.

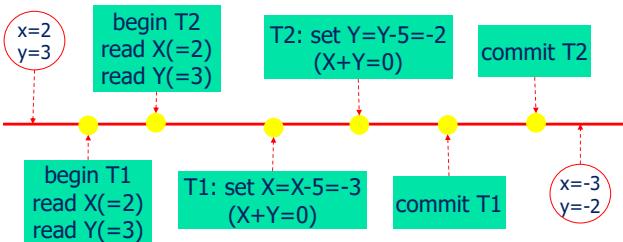


图 5.3: 快照隔离中的不一致现象

SQL Server 中的快照隔离:

1. 事务级快照隔离: 读取操作得到数据项在事务开始时刻最近的已提交版本. 通过回滚解决冲突更新操作.
2. 语句级快照隔离 (Read Committed SI, RCSI): 读取操作得到数据项在语句开始时刻最近的已提交版本. 通过阻塞解决冲突更新操作.

## 5.5 事务可串行化判定

事务可串行化判定, 如何判定两个调度是等价的?

- 冲突可串行化: 冲突指令. 微观角度: 是否可以交换两个相邻指令?
- 视图可串行化: 从读一致性. 宏观视角: 如何保证每个事务在两个调度中是相同的?

### 5.5.1 冲突可串行化

考虑一个调度  $S$  中的两条连续指令 (仅限于 read 和 write)  $I_i$  和  $I_j$ , 分别属于事务  $T_i$  和  $T_j$ .

1.  $I_i = \text{read}(Q), I_j = \text{read}(Q).$
2.  $I_i = \text{read}(Q), I_j = \text{write}(Q).$
3.  $I_i = \text{write}(Q), I_j = \text{read}(Q).$
4.  $I_i = \text{write}(Q), I_j = \text{write}(Q).$

只有上面的情况 1 可交换.

#### 定义 5.8 (冲突指令)

两条指令是不同事务在相同数据项上的操作, 并且其中至少有一个是 write 指令.



#### 定义 5.9 (冲突等价)

如果调度  $S$  可以经过交换一系列非冲突指令转换成调度  $S'$ , 则称调度  $S$  与  $S'$  是冲突等价的.



**定义 5.10 (冲突可串行化)**

当一个调度  $S$  与一个串行调度冲突等价时则称该调度是冲突可串行化的.

**定义 5.11 (优先图)**

调度  $S$  的优先图 (Precedence Graph) 是一个有向图  $G = (V, E)$ ,  $V$  是顶点集,  $E$  是边集. 顶点集由所有参与调度的事务组成, 边集由满足下述条件之一的边  $T_i \rightarrow T_j$  组成:

1. 在  $T_j$  执行  $\text{read}(Q)$  之前,  $T_i$  执行  $\text{write}(Q)$ .
2. 在  $T_j$  执行  $\text{write}(Q)$  之前,  $T_i$  执行  $\text{read}(Q)$ .
3. 在  $T_j$  执行  $\text{write}(Q)$  之前,  $T_i$  执行  $\text{write}(Q)$ .

**定理 5.1**

如果优先图中存在边  $T_i \rightarrow T_j$ , 则在任何等价于  $S$  的串行调度  $S'$  中,  $T_i$  都必须出现在  $T_j$  之前.

**引理 5.1 (有向无环图一定有一个入度为 0 的节点)**

有向无环图一定有一个入度为 0 的节点.



**证明** 我们称入度为 0 的节点为源点.

假设  $n$  个顶点的有向无环图没有源点, 对于顶点  $v_{i1}$ , 由于其不是源点, 那么一定存在一个节点  $v_{i2}$ , 且从  $v_{i2}$  到  $v_{i1}$  有一条有向边. 依此类推, 可以得到一条路径:  $v_{ik} \rightarrow \dots v_{i2} \rightarrow v_{i1}$ .

- 对于  $v_{ik}$  的入边的起始节点  $v_{i,k+1}$ , 如果  $v_{i,k+1} \in \{v_{i1}, v_{i2}, \dots, v_{ik}\}$ , 构成了环, 矛盾.
- 如果  $v_{i,k+1} \notin \{v_{i1}, v_{i2}, \dots, v_{ik}\}$ , 由于顶点数量  $n$  是有限的, 那么可以一直扩展此路径, 直到所有顶点都包含在此路径中, 即  $v_{in} \rightarrow \dots v_{i2} \rightarrow v_{i1}$ . 此时该路径的起始顶点  $v_{in}$  由于不是源点, 一定有另外一个顶点  $v_{im}$  可直达  $v_{in}$ , 而  $v_{im} \in \{v_{i1}, v_{i2}, \dots, v_{in}\}$ , 则也构成回路, 矛盾.

综上所述, 可证明一个有向无环图中至少有一个源点.

**定理 5.2**

如果调度  $S$  的优先图中有环, 则  $S$  是非冲突可串行化的; 如果图中无环, 则是冲突可串行化的.



**证明** 首先, 根据引理 5.1, 考虑到有向无环图中一定存在一个节点入度为 0. 然后计算出拓扑排序即可.

与冲突可串行化等价的串行顺序 = 拓扑排序.

## 5.5.2 视图可串行化

**定义 5.12 (视图等价)**

考虑关于某个事务集的两个调度  $S, S'$ , 若调度  $S, S'$  满足以下条件, 则称它们是视图等价的:

1.  $\llbracket \text{数据库初值} \rrbracket \xrightarrow{S} r_i(Q) \wedge \llbracket \text{数据库初值} \rrbracket \xrightarrow{S'} r_i(Q).$
2.  $\llbracket w_j(Q) \rrbracket \xrightarrow{S} r_i(Q) \wedge \llbracket w_j(Q) \rrbracket \xrightarrow{S'} r_i(Q).$
3.  $w_i(Q) \xrightarrow{S} \llbracket \text{数据库终值} \rrbracket \wedge w_i(Q) \xrightarrow{S'} \llbracket \text{数据库终值} \rrbracket.$



**注** 条件 1 和 2 保证从读一致性, 条件 3 保证两个调度得到最终相同的系统状态.

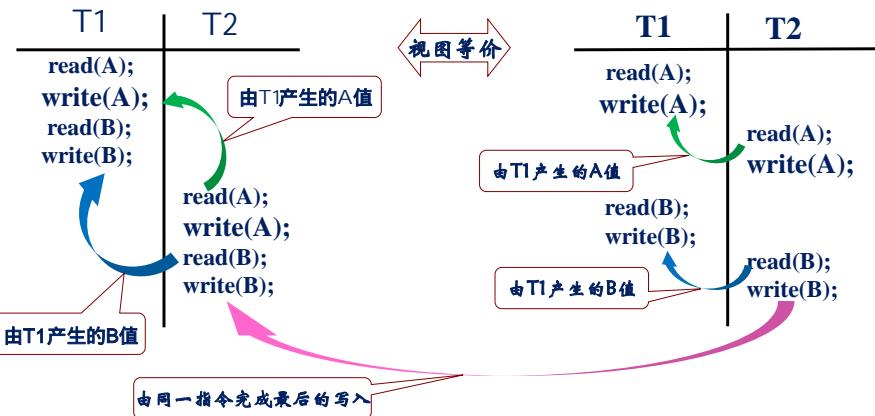


图 5.4: 视图等价

**定义 5.13 (视图可串行化)**

如果某个调度视图等价于一个串行调度，则称该调度是 **视图可串行化的**.

**推论 5.1**

冲突可串行化调度一定是视图可串行化的.

**推论 5.2**

存在视图可串行化但非冲突可串行化的调度.



盲目写操作: write 之前并不执行 read 操作.

无用的写操作: 被覆盖掉的写操作.

带标记的优先图的构造: 设调度  $S = \{T_1, T_2, \dots, T_n\}$ , 构造两个虚事务  $T_b, T_f$ , 其中  $T_b$  为  $S$  中所有  $\text{write}(Q)$  操作,  $T_f$  为  $S$  中所有  $\text{read}(Q)$  操作. 在调度  $S$  的开头插入  $T_b$ , 在调度  $S$  的末尾插入  $T_f$ , 得到新的调度  $S'$ .

1. 如果  $T_j$  读取  $T_i$  写入的数据项的值, 则加入边  $T_i \xrightarrow{0} T_j$ .
2. 如果在优先图中不存在从  $T_i$  到  $T_f$  的通路, 则  $T_i$  是无用事务, 将其删除.
3. 对于每个数据项  $Q$ , 如果  $T_j$  读取  $T_i$  写入的  $Q$  值,  $T_k$  执行  $\text{write}(Q)$  且  $T_k \neq T_b$ , 则: 其实在考虑  $T_i \xrightarrow{0} T_j$  之间插入  $T_k$ .
  - (a). 如果  $T_i = T_b$  且  $T_j \neq T_f$ , 考虑  $T_k$  在优先图中的位置: 插入  $T_j \xrightarrow{0} T_k$ .
  - (b). 如果  $T_i \neq T_b$  且  $T_j = T_f$ , 考虑  $T_k$  在优先图中的位置: 插入  $T_k \xrightarrow{0} T_i$ .
  - (c). 如果  $T_i \neq T_b$  且  $T_j \neq T_f$ , 考虑  $T_k$  在优先图中的位置:  $T_k \xrightarrow{p} T_i$  和  $T_j \xrightarrow{p} T_k$ . 其中  $p$  是一个唯一的, 在前面边的标记中未曾用过的大于 0 的整数. 这实际上是说这两条边二选一.

标号为 0 的边组成底图, 标号为  $p$  的取出一条可以形成两张图, 判定是否有环.

**定理 5.3 (视图可串行化判定准则)**

只要有一个优先图无环, 则调度是视图可串行化的.



存在可串行化但非视图可串行化的调度:

T1	T2	T1	T2
read(A);		read(A);	
A := A - 50		A := A - 50	
<b>write(A);</b>		<b>write(A);</b>	
	read(B);		read(B);
	B := B - 10		B := B + 50
	<b>write(B);</b>		<b>write(B);</b>
<b>read(B);</b>			read(B);
B := B + 50			B := B - 10
<b>write(B);</b>		<b>write(B);</b>	
	<b>read(A);</b>		<b>read(A);</b>
	A := A + 10		A := A + 10
	<b>write(A);</b>		<b>write(A);</b>

图 5.5: 可串行化但非视图可串行化的调度

## 5.6 保存点

平面事务: 一层结构 `begin transaction...commit.`

平面事务的缺点: 不能部分回滚.

需要部分回滚的场合:

1. 非线性流程控制. 比如订票分段, 只需要回滚到没订上的那一段.
2. 批量更新.

解决方法: 保存点.

```
begin
  S1;
  sp1 := create_savepoint();
  ...
  Sn;
  spn := create_savepoint();
  ...
  if (condition) rollback(spi);
  ...
commit();
```

```
start transaction;
insert into test_savePoint values ('sp0');
savepoint sp1;
insert into test_savePoint values ('sp1');
savepoint sp2;
insert into test_savePoint values ('sp2');
savepoint sp3;
insert into test_savePoint values ('sp3');
rollback to sp2;
commit;
```

# 第六章 并发控制

## 6.1 基于锁的协议

### 定义 6.1 (封锁)

封锁就是一个事务对某个数据对象加锁，取得对它一定的控制，限制其它事务对该数据对象使用。

要访问数据项  $R$ , 事务  $T_i$  必须先申请对  $R$  的封锁, 如果  $R$  已经被事务  $T_j$  加了不相容的锁, 则  $T_i$  需要等待, 直至  $T_j$  释放它的封锁.



**封锁性能:** 事务吞吐量, TPC-C.

保证可串行化的一种协议是两阶段封锁协议 (two-phase Locking protocol). 该协议要求每个事务分两个阶段提出加锁和解锁申请。

1. 增长阶段 (growing phase): 一个事务可以获得锁, 但不能释放任何锁.
2. 缩减阶段 (shrinking phase): 一个事务可以释放锁, 但不能获得任何新锁.

起初, 一个事务处于增长阶段. 事务根据需要获得锁. 一旦该事务释放了一个锁, 它就进入了缩减阶段, 并且它不能再发出加锁请求.

# 第七章 恢复控制

## 7.1 故障类型

故障分为三类:

1. **事务故障**: 事务运行没有到达预期的终点就被中止.
2. **系统故障**: 由于系统错误导致的故障.
3. **介质故障**: 由于介质错误导致的故障.

### 定义 7.1 (事务故障)

单个事务的运行没有到达预期的终点就被中止.

分为:

1. **非预期故障**: 不能由事务程序处理的. 如运算溢出, 发生死锁而被选中撤消该事务.
2. **预期故障**: 应用程序可以发现的事务故障, 并且应用程序可以让事务回滚. 如转帐时发现帐面金额不足.



### 定义 7.2 (系统故障)

又称为**软故障**(soft crash). 在硬件故障、软件错误的影响下, 虽引起内存信息丢失, 但未破坏外存中数据. 如 CPU 故障、突然停电. DBMS, OS, 应用程序等异常终止.



### 定义 7.3 (介质故障)

又称为**硬故障**(hard crash). 又称磁盘故障, 破坏外存上的数据库, 并影响正在存取这部分数据的所有事务. 如磁盘损坏、磁带损坏. 磁盘的磁头碰撞, 瞬时的强磁场干扰.



## 7.2 故障恢复

### 定义 7.4 (恢复)

恢复是把数据库从错误状态恢复到某一正确状态的功能, 从而确保数据库的一致性.

恢复的基本原理是**冗余**. 即数据库中任一部分的数据可以根据存储在系统别处的冗余数据来重建.



### 7.2.1 备份

#### 定义 7.5 (转储)

将数据库复制到磁带或另一个磁盘上保存起来的过程.

这些备用数据称为**后备(后援)副本**.



转储分为:

1. 静态转储. 转储期间不允许对数据库进行任何存取、修改活动.
2. 动态转储. 转储期间允许对数据库进行存取或修改.
3. 海量转储. 每次转储全部数据库.
4. 增量转储. 每次只转储自上次转储以来发生变化的数据.

数据库备份 by SQL Server:

```

EXEC sp_addumpdevice 'disk', 'mybackup', 'c:\backup\mybackup.dat';
BACKUP DATABASE mydb TO mybackup
RESTORE DATABASE mydb FROM mybackup

```

```

-- 完整备份. 首次备份或初始化备份, 为后续差异备份提供一个基准点.
backup database LJCHEN to MyBKDB with init
-- 差异备份
backup database LJCHEN to MyBKDB with differential
restore database LJCHEN from MyBKDB with norecovery
restore database LJCHEN from MyBKDB with norecovery

```

MySQL 备份:

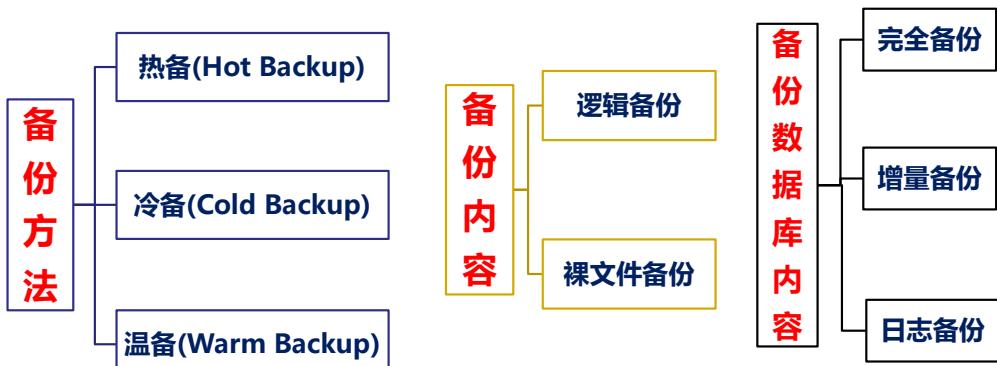


图 7.1: MySQL 备份

```

# 备份 MySQL 中的所有数据库到一个 SQL 文件中
mysqldump -uroot -p --all-databases > /home/mysql/backups/mysqldump_all_databases.sql

# 备份名为 myDb 的单个数据库
mysqldump -uroot -p myDb > /home/mysql/backups/mysqldump_mydb.sql

# 同时备份两个数据库 myDb1 和 myDb2
mysqldump -uroot -p --databases myDb1 myDb2 > /home/mysql/backups/mysqldump_databases_mydb12.sql

# 备份数据库 myDb 中的单张表 myTb
mysqldump -uroot -p myDb myTb > /home/mysql/backups/mysqldump_myTb.sql

# 备份 myDb 数据库中的 myTb 表, 并且只导出 id <= 10 的记录
mysqldump -uroot -p --databases myDb --tables myTb --where="id <= 10" > /home/mysql/backups/
mysqldump_myTb10.sql

# 仅备份 db1 和 db2 的数据库结构 (不包含数据)
mysqldump --no-data --databases db1 db2 > /home/mysql/backups/structure.sql

# 在 MySQL 客户端中使用 source 恢复全库备份
source /home/mysql/backups/mysqldump_all_databases.sql

```

```
# 使用 mysql 命令导入之前备份的 myDb 数据库
mysql -uroot -p myDb < /home/mysql/backups/mysqldump_myDb.sql
```

```
-- 把数据库 my_table 中的数据导出到 data.txt 文件中
select * into outfile 'data.txt'
fields terminated by ','
lines terminated by '\r\n'
from my_table;

-- 把 data.txt 文件中的数据导入到 my_table 中
load data infile 'data.txt'
into table my_table
fields terminated by ','
lines terminated by '\r\n';
```

下面是用于自动备份多个数据库的 Bash 脚本，并设置了定时任务（通过 crontab）每天凌晨 3 点执行一次。

```
#!/bin/bash
cd /usr/local/mysql/backup/scripts/
backup_date=`date +%Y%m%d`
filename=/home/db/mysql/backups/databackup_$backup_date.sql
mysql -e "show databases;" -uroot -proot | grep -E "myDb*" | xargs mysqldump -uroot -proot --databases
> $filename
echo 'databases backup successfully...'
crontab -e
00 03 * * * /usr/local/mysql/backup/scripts/databases_backup.sh
```

## 7.2.2 日志

### 定义 7.6 (日志)

日志文件是以事务为单位用来记录数据库的每一次更新活动的文件，由系统自动记录。

日志内容包括：记录名、旧记录值、新记录值、事务标识符、操作标识符等。

1. 事务  $T_i$  开始时，写入日志： $\langle T_i \text{ start} \rangle$ 。
2. 事务  $T_i$  执行  $\text{write}(X)$  之前，写入日志： $\langle T_i, X, V_1, V_2 \rangle$ ，其中  $V_1$  为更新前的值（前像）， $V_2$  为更新后的值（后像）。
3. 事务  $T_i$  提交时，写入日志： $\langle T_i \text{ commit} \rangle$ 。



根据日志可把事务分为：

1. **圆满事务**：日志文件中记录了事务的  $\text{commit}$  标识。
2. **夭折事务**：日志文件中只有  $\text{start}$  标识，没有记录事务的  $\text{commit}$  标识。

基本的恢复操作：

1. 对圆满事务的更新日志执行  $\text{redo}$  操作，即重新执行该操作，修改对象被赋予新记录值。幂等性： $\text{redo} = \text{redo}^2$ 。
2. 对夭折事务的更新日志执行  $\text{undo}$  操作，即撤销该操作，修改对象被赋予旧记录值。幂等性： $\text{undo} = \text{undo}^2$ 。

其他日志恢复技术：提交日志（Commit Logging）

1. 事务提交之前，其修改结果不会写入磁盘
2. 日志中没有提交标记的事务，其修改结果没有写盘

3. 恢复时只需重做日志中的提交事务
4. OceanBase、Hekaton(SQL Server 内存存储引擎)

无日志恢复技术: Shadow Paging.

1. 被修改的数据会同时存在两份, 一份是原来的数据, 另一份是修改后的数据 (影子, shadow).
2. 通过两个目录结构分别指向修改前的数据和修改后的数据, 最后 Current 指针原子切换到新的目录上, 表示事务提交成功.
3. 当事务提交时, 以一次原子数据写入让整个事务新的修改生效.

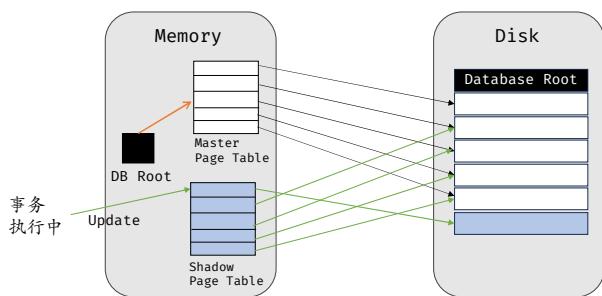


图 7.2: Shadow Paging: 正在修改的事务

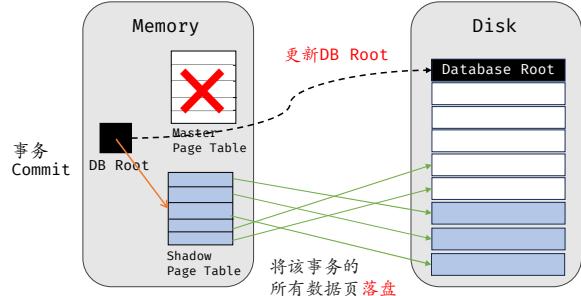


图 7.3: Shadow Paging: 事务 Commit

Undo/Rollback: 删除 shadow pages (table), 啥都不用做.

Redo: 不需要 redo, 因为每次写事务都会将数据落盘.

MySQL 日志文件:

1. 重做日志 (redo log)
2. 回滚日志 (undo log)
3. 二进制日志 (binary log)
4. 错误日志 (error log)
5. 慢查询日志 (slow query log)
6. 一般查询日志 (general log)
7. 中继日志 (relay log)

慢查询日志的例子:

```
-- 开启慢查询日志功能 (1 表示开启)
SET GLOBAL slow_query_log = 1;

-- 设置慢查询日志文件的保存路径 (Windows 系统下路径为 C:\slow_statement.log)
-- 注意: MySQL 进程必须对该路径有写入权限
SET GLOBAL slow_query_log_file = 'C:\slow_statement.log';

-- 设置慢查询时间阈值为 10 秒
-- 所有执行时间超过 10 秒的 SQL 查询会被记录到慢查询日志中
SET GLOBAL long_query_time = 10;

-- 设置慢查询日志输出方式为文件 (FILE), 也可以设置为 TABLE (表)
SET GLOBAL log_output = 'FILE';

-- 测试语句: 执行一个耗时较长的操作, 用于触发慢查询日志记录
-- 此处重复执行 MD5('mysql') 函数 99,999,999 次, 模拟耗时操作
-- 如果该查询执行时间超过 long_query_time (10秒), 则会被记录到慢查询日志中
```

```
SELECT BENCHMARK(99999999, MD5('mysql'));
```

事务日志是自上次备份事务日志后对数据库执行的所有事务记录，它可以将数据库恢复到特定时点或恢复到故障点。

```
-- 1. 对数据库 MyDB 执行完整备份，备份到备份设备 MyDB_1
-- 完整备份是所有其他备份（差异、日志）的基础
BACKUP DATABASE MyDB TO MyDB_1;

-- 2. 对数据库 MyDB 执行事务日志备份，备份到备份设备 MyDB_log1
-- 此时会截断日志，并记录从上一次备份以来的所有事务日志
BACKUP LOG MyDB TO MyDB_log1;

-- 3. 再次对数据库 MyDB 执行事务日志备份，备份到备份设备 MyDB_log2
-- 使用 WITH NO_TRUNCATE 表示不截断事务日志，允许后续继续备份
-- 注意：一般只在紧急情况下使用，避免日志文件无限增长
BACKUP LOG MyDB TO MyDB_log2 WITH NO_TRUNCATE;

-- 4. 恢复完整备份，从备份设备 MyDB_1 还原数据库 MyDB
-- 使用 WITH NORECOVERY 表示数据库仍处于还原状态，等待后续日志应用
RESTORE DATABASE MyDB FROM MyDB_1 WITH NORECOVERY;

-- 5. 应用第一个事务日志备份 MyDB_log1
-- 仍然使用 WITH NORECOVERY，表示还有更多日志需要恢复
RESTORE LOG MyDB FROM MyDB_log1 WITH NORECOVERY;

-- 6. 应用第二个事务日志备份 MyDB_log2
-- 使用 WITH RECOVERY 表示这是最后一次恢复操作，数据库将变为可用状态
RESTORE LOG MyDB FROM MyDB_log2 WITH RECOVERY;
```

with norecovery: 重做所有日志记录。

with recovery: 回滚失败事务日志记录

**例题 7.1** 事务 T 从 A 账户过户 ¥50 到 B 账户：

```
read(A); A := A - 50; write(A)
read(B); B := B + 50; write(B)
commit(T)
```

MyDB\_log1:  $\langle T, A, 100, 50 \rangle$ .

MyDB\_log2:  $\langle T, B, 100, 150 \rangle, \langle T, \text{commit} \rangle$ .

下面各自恢复结果是什么：

```
restore log MyDB from MyDB_log1 with norecovery
restore log MyDB from MyDB_log2 with recovery
```

```
restore log MyDB from MyDB_log1 with recovery
restore log MyDB from MyDB_log2 with recovery
```

第一种情况：应用日志 MyDB\_log1，只执行了对 A 账户的更新操作，B 账户没有更新，事务未提交，所以 B 账户的余额仍然是 100。数据库仍处于 RESTORING 状态。接着应用日志 MyDB\_log2，执行了对 B 账户的更新操作，由于事务已经提交，所以 B 账户的余额被更新为 150。

第二种情况: 应用日志 MyDB\_log1, 只执行了对 A 账户的更新操作, B 账户没有更新, 事务未提交, 所以 B 账户的余额仍然是 100. 现在完成恢复过程并且发布数据库了. 但是是不完整的 log, 需要回滚. 同时已经恢复了, 无法再 restore 了. 最后 A 和 B 都没有改变.

恢复模型: SQL Server

1. 简单恢复模型: 允许将数据库恢复到最新的备份. 数据库备份 + 差异备份 (可选)
2. 完全恢复: 允许将数据库恢复到故障点状态. 数据库备份 + 差异备份 (可选) + 事务日志备份.
3. 大容量日志记录恢复: 允许大容量日志记录操作 (bulk insert...). 数据库备份 + 差异备份 (可选) + 事务日志备份.

### 7.2.3 WAL, Write Ahead Log

WAL 的中文名是预写日志系统, 其核心思想是把用户所有的修改操作 (插入、删除) 先写入日志中, 然后再应用到系统状态里. 一旦成功写完日志, 即可通知用户操作成功. 由于日志是以尾部追加方式写入, 耗时较短, 所以不会长时间阻塞用户线程. 此外为防止意外退出导致数据丢失, 系统重启时还会根据日志重做用户操作, 以保证数据可靠性.

# 第八章 数据库存储

## 8.1 存储介质

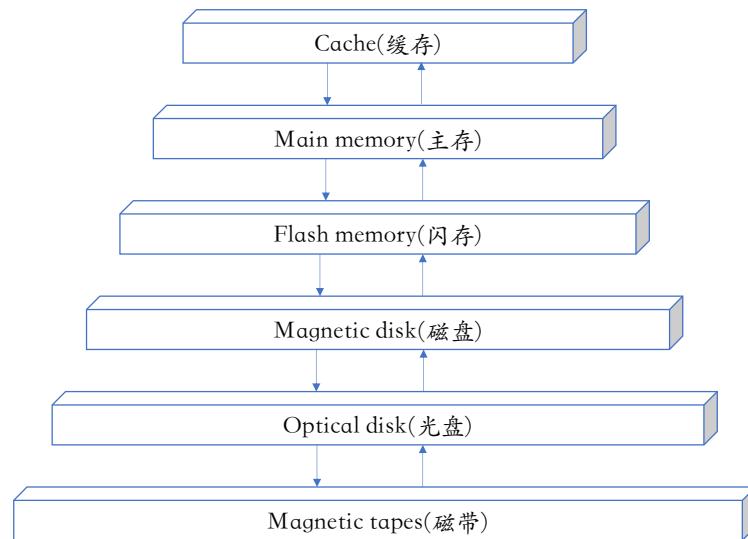


图 8.1: 物理存储介质的层次

局部性原理: CPU 访问存储器时, 无论是存取指令还是存取数据, 所访问的存储单元都趋于聚集在一个较小的连续区域中.

## 参考文献

- [1] Wen Li et al. *Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement (v1.0)*. Oct. 2016. doi: [10.48550/arXiv.1610.02455](https://doi.org/10.48550/arXiv.1610.02455). arXiv: [1610.02455 \[cs\]](https://arxiv.org/abs/1610.02455). (Visited on 05/22/2025).
- [2] Yu A. Malkov and D. A. Yashunin. *Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs*. Aug. 2018. doi: [10.48550/arXiv.1603.09320](https://doi.org/10.48550/arXiv.1603.09320). arXiv: [1603.09320 \[cs\]](https://arxiv.org/abs/1603.09320). (Visited on 05/22/2025).