



《数据库概论》学习笔记

Introduction to Database Systems

作者：VectorPikachu

组织：EECS, Peking University

时间：June 16, 2025



课程教材《数据库系统概念》[7]

目录

第一章 数据库系统简介	1
1.1 数据独立性	1
1.2 数据管理发展阶段	1
1.3 数据模型	2
1.4 数据库系统的构成	4
第二章 实体-联系模型	5
2.1 E-R 模型基本概念	5
2.2 实体的码 (Key)	6
2.3 属性	6
2.4 联系	7
2.5 ER 设计实例	9
2.5.1 基于业务描述的 ER 设计实例	9
2.5.2 由业务单据生成 ER 模型	10
2.6 其他概念术语	11
2.7 扩展 ER 特性	12
2.8 概念数据库设计过程	14
2.9 ER 模型向关系模式的转换	14
2.10 关系模式向 ER 的转换	14
第三章 关系模型	15
3.1 关系基本概念	15
3.2 关系模型三要素	15
3.2.1 数据结构	15
3.2.2 数据操作	16
3.2.3 数据完整性	17
3.3 关系代数运算	18
3.3.1 基本关系代数运算	18
3.3.1.1 一元运算	18
3.3.1.2 多元运算	19
3.3.2 扩展关系代数运算	20
3.3.3 关系代数更新运算	24
3.4 关系代数查询实例	25
3.5 关系演算	27
3.5.1 元组关系演算	27
3.5.2 域关系演算	28
3.6 关系系统	28
3.6.1 全关系系统的十二条准则	28
第四章 SQL	30
4.1 数据定义	30
4.1.1 数据模式定义	31

4.1.2 数据类型	34
4.1.3 索引	37
4.1.4 视图定义	39
4.1.5 临时表和内存表	41
4.1.6 公用表表达式 CTE	41
4.1.7 分区表	41
4.2 数据查询	42
4.2.1 空值	44
4.2.2 连接运算	45
4.2.3 集合运算	48
4.2.4 聚集函数 (Group function)	49
4.2.5 分组运算	49
4.2.6 嵌套子查询	52
4.2.7 字符串与文本操作	54
4.3 数据更新	55
4.4 服务器端脚本语言	56
4.4.1 SQL 脚本语法成分	56
4.4.2 服务器端 SQL 脚本形式	57
第五章 完整性与安全性	59
5.1 数据完整性	59
5.2 数据库安全性	62
第六章 关系中的非关系数据	67
6.1 递归查询	67
6.1.1 层次结构的关系表示	67
6.1.2 递归查询	67
6.1.3 层次结构典型查询问题	69
6.2 XML	69
6.3 JSON	72
6.4 向量	74
第七章 关系规范化	76
7.1 关系模式的设计问题	76
7.2 函数依赖	76
7.3 码的定义 (使用函数依赖)	77
7.4 范式	78
7.4.1 1NF	78
7.4.2 2NF	79
7.4.3 3NF	79
7.4.4 BCNF	80
7.4.5 多值依赖	80
7.4.6 4NF	81
7.4.7 PJNF	81
7.5 Armstrong 公理系统	82
7.6 闭包计算	83

7.7 候选码计算	84
7.8 函数依赖的等价和覆盖	85
7.9 函数依赖和多值依赖的推理规则	86
7.10 模式分解	86
7.10.1 保持函数依赖分解	87
7.10.2 保持无损连接分解	87
7.10.3 关系模式分解算法	88
7.10.3.1 达到 BCNF 无损连接分解算法	88
7.10.3.2 达到 4NF 无损连接分解算法	89
7.10.3.3 达到 3NF 保持函数依赖的分解	90
7.10.3.4 同时保持函数依赖和无损连接的分解算法	90
7.11 模式调优	91
7.11.1 垂直划分	91
第八章 事务	92
8.1 SQL 中的事务	92
8.1.1 事务基本特性 ACID	92
8.2 事务调度	93
8.2.1 并发调度中的不一致现象	94
8.3 事务隔离性级别	94
8.4 快照隔离	95
8.5 事务可串行化判定	95
8.5.1 冲突可串行化	95
8.5.2 视图可串行化	96
8.6 保存点	98
第八章 练习	99
第九章 并发控制	101
9.1 基于锁的协议	101
9.1.1 封锁类型	101
9.1.2 两阶段封锁协议	102
9.1.3 先读后写: 锁转换	103
9.1.4 先读后写: 更新锁	103
9.1.5 封锁粒度	104
9.1.6 码范围锁	105
9.1.7 一些具体系统中的特殊锁模式	107
9.1.8 封锁带来的问题	108
9.1.8.1 阻塞	108
9.1.8.2 死锁 (Deadlock)	108
9.1.8.3 活锁 (Live lock)	109
9.1.9 其他锁相关概念	110
9.2 基于时间戳的协议	110
9.3 基于有效性检查的协议	111
9.4 MVCC(Multi-Version Concurrency Control)	112
9.4.1 MySQL MVCC 在不同隔离性级别的读视图	113

第十章 恢复控制	116
10.1 故障类型	116
10.2 备份	116
10.3 日志	118
10.4 WAL, Write Ahead Log	121
10.5 故障恢复	122
10.5.1 事务故障恢复	122
10.5.2 系统故障恢复	122
10.5.3 介质故障恢复	122
10.6 检查点 (Checkpoint)	122
10.6.1 一致性检查点	123
10.6.2 模糊检查点	123
10.7 ARIES 恢复算法	125
10.7.1 Buffer Manager(BM) 的实现策略	125
10.7.2 日志类型	126
10.7.3 ARIES 恢复算法中的数据结构	127
10.7.4 ARIES 恢复算法实现过程	128
第十一章 数据库存储	130
11.1 存储介质	130
11.2 廉价磁盘冗余阵列 (RAID)	131
11.2.1 RAID 0	132
11.2.2 RAID 1	132
11.2.3 RAID 3: 位交叉奇偶校验	132
11.2.4 RAID 4: 块交叉奇偶校验	133
11.2.5 RAID 5: 块交叉的分布奇偶校验	133
11.2.6 RAID 6	133
11.2.7 RAID 总结	133
11.2.8 选择合适的 RAID 级别	133
11.3 缓冲区	133
11.4 数据库存储结构	134
11.4.1 页面与区间	135
11.4.2 GAM - 全局分配位图 (bitmap)	135
11.4.3 PFS - 空闲页空间	135
11.4.4 IAM - 索引分配位图	135
11.4.5 基本页结构	135
11.5 索引	137
11.5.1 B+ 树索引	137
11.5.2 散列索引	138
11.5.3 位图索引	139
11.5.3.1 位片索引 (Bit-sliced Index)	140
11.5.3.2 关系表的行式存储和列式存储	140
11.5.4 多维索引	141
11.5.4.1 k-d tree	141
11.5.4.2 四叉树	142

11.5.4.3 R-tree	142
11.5.5 LSM 树	143
11.5.5.1 LSM 树的设计思想	143
11.5.5.2 LSM 树的增删查改	143
11.5.5.3 LSM-tree 读写放大	145
第十二章 查询处理	146
12.1 查询处理流程	146
12.1.1 查询处理步骤	146
12.1.2 查询优化	146
12.1.3 代价估算	147
12.1.4 统计信息	148
12.2 执行计划	148
12.2.1 数据查找	148
12.2.2 各种访问方法	149
12.3 算子实现	149
12.3.1 选择运算	150
12.3.2 排序	151
12.3.3 集合运算	151
12.3.4 去重	151
12.3.5 连接运算	151
12.3.6 多表连接	154
12.3.7 算子执行方式	154
12.3.8 算子封装形式: 迭代器	154
第十三章 考试题型	155
参考文献	156

第一章 数据库系统简介

期末考试提纲

- 数据独立性
- 文件系统在数据管理方面的不足
- 数据库系统在数据管理方面的特性
- 数据模型的概念、种类、特性比较
- 数据库模式：三级模式、两级映像
- 数据库是如何保证数据独立性的
- DBMS 各项功能、DBA 职责

1.1 数据独立性

定义 1.1 (数据结构)

按照逻辑关系组织起来的一批数据，按一定的存储方法把它存储在计算机中，并在这些数据上定义了一个运算的集合。

1. 逻辑结构：数据之间存在的逻辑关系。e.g., 表、树、图。
2. 物理结构：数据在计算机内的存储方式。e.g., 顺序方式、链接方式。



定义 1.2 (数据独立性)

当数据结构发生变化时，通过系统提供的映象（转换）功能，使应用程序不必改变。

1. 数据的物理独立性：当数据存储结构发生变化时，使应用程序不必改变。
2. 数据的逻辑独立性：当数据逻辑结构发生变化时，使应用程序不必改变。



数据定义：逻辑结构、物理结构。

数据操作：查询 + 更新。

数据约束：对客观事物的合理反映，数据一致性。

1.2 数据管理发展阶段

人工管理 → 文件系统 → 数据库系统 → 大数据时代

文件系统管理的不足：

1. 数据定义独立性弱：
 - (a). 数据与程序紧密结合；
 - (b). 数据的语义信息只能由程序来解释；
 - (c). 数据分散管理；
 - (d). 数据共享困难；
2. 数据完整性难于维护（多副本）；
3. 数据查询困难（文件系统眼中的数据只是字符流）

数据库系统眼中的数据：**结构化数据**。

数据库系统保证**数据独立性**的举措：

1. 把数据库定义和描述从应用程序中分离出去；
2. 数据描述是分级的（全局逻辑、局部逻辑、存储）；
3. 数据存取由系统管理，用户不必考虑存取路径等细节，从而简化了应用程序（SQL）。

数据库具有统一的**数据控制**能力：

1. 完整性控制
2. 安全性控制
3. 并发控制
4. 恢复控制

数据库系统的特点:

1. 面向全组织, 有结构的数据, 记录之间无联系, 结构化数据
2. 冗余度小, 集中管理, 易扩充性
3. 高数据独立性
4. 统一的数据控制功能: 安全性控制 (security), 并发控制 (concurrency), 完整性控制 (integrity), 恢复控制 (recovery)

1.3 数据模型

定义 1.3 (数据模型)

是数据库系统中用于提供信息表示和操作手段的形式构架.

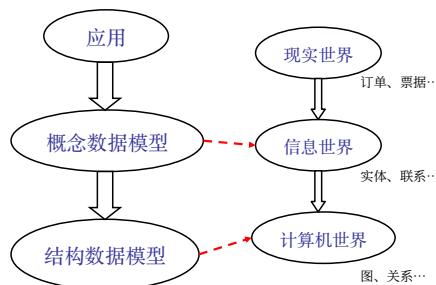


图 1.1: 数据模型

概念模型: 强调语义表达能力, 给人看

1. E-R(实体-联系模型, entity-relationship model).
2. 基于对象的数据模型 (object-based data model)

结构数据模型的概念、种类、特性比较 (强调数据结构, 计算机实现, 形式化定义, 给程序看.):

1. 层次模型: 使用树状结构表示数据之间的关系, 每个记录只有一个父记录.
2. 网络模型: 扩展了层次模型, 允许一个记录有多个父记录.
3. 关系模型: 基于关系理论, 使用表格形式组织数据, 是目前最常用的数据模型.
4. 面向对象模型: 将数据及其处理方法封装在一起, 支持复杂数据类型.

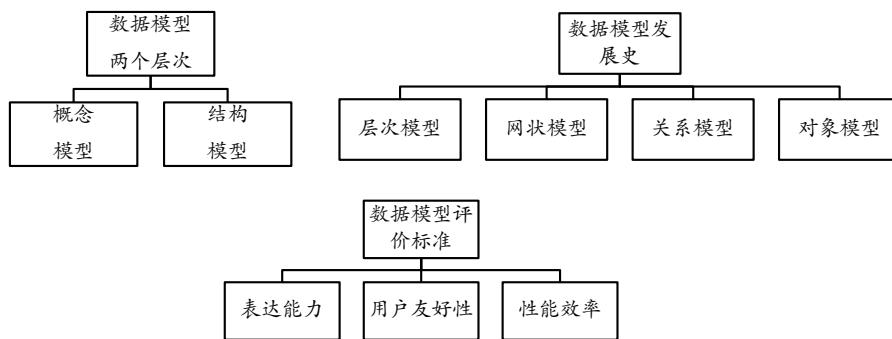


图 1.2: 数据模式总结

定义 1.4 (元数据 (metadata))

元数据是描述数据的数据.



所谓**数据库模式** 实际上是一种类型, 而它的值就是数据库当前的一个快照.

数据库系统的架构通常遵循**三级模式**和**两级映像**的原则.

1. 外模式 (视图层): 用户看到并与之交互的数据视图.
2. 模式 (逻辑层): 描述数据库中数据的整体逻辑结构.
3. 内模式 (物理层): 定义数据的实际存储方式和访问路径.
4. 两级映像指的是外模式/模式映像和模式/内模式映像, 用于保证数据独立性
 - 外模式/模式映像: 定义某个外模式和模式之间的对应关系
 - 模式/内模式映像: 定义数据逻辑结构与存储结构之间的对应关系

我的理解: 所谓外模式映像就是定义这样一个映射 f 把当前的 $view$ 映射到某个表上, 内模式映像则是一个映射 g 将某个 $table$ 映射到存储结构上.

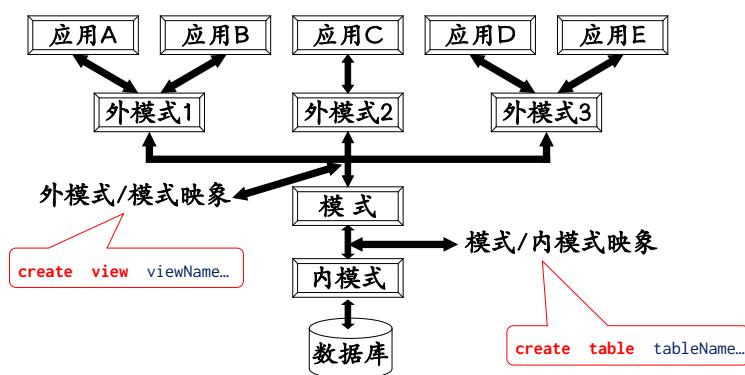


图 1.3: 数据库的三种模式

注 数据库的逻辑独立性来自于: 外模式/模式映像. 数据库的物理独立性来自于: 模式/内模式映像.

逻辑模式

sno	sname	age
s01	Tom	18
s02	Jerry	19
s03	Bob	20

sno	cno	grade
s01	c01	88
s02	c01	90
s01	c02	86

内模式

sno	sname	age
s01	Tom	18
s02	Jerry	19
s03	Bob	20

sno	sname	age
s01	Tom	18
s02	Jerry	19
s03	Bob	20

列式存储

sno	cno	grade
s01	c01	88
s02	c01	90
s01	c02	86

混合式存储

图 1.4: 数据库内模式的例子

1.4 数据库系统的构成

定义 1.5 (数据库)

数据的集合. 由 DBMS 统一管理, 多用户共享.



定义 1.6 (数据库管理系统 DBMS)

系统软件, 对数据库进行统一管理和控制.



定义 1.7 (数据库系统)

带有数据库的整个计算机系统, 包括硬件、软件、数据、人员.



DBMS(数据库管理系统) 的功能包括但不限于:

- 数据定义: 创建、修改和删除数据库中的对象.
- 数据操作: 插入、更新、删除和查询数据库中的数据.
- 数据库事务管理: 确保数据库操作的原子性、一致性、隔离性和持久性 (ACID 属性).
- 数据库备份与恢复: 提供机制以防止数据丢失, 并在必要时恢复数据.

DBA(数据库管理员) 的职责:

- 建库方面: 确定模式、外模式、存储结构、存取策略; 负责数据的整理和装入.
- 用库方面:
 - 定义完整性约束条件
 - 规定数据的保密级别、用户权限
 - 监督和控制数据库的运行情况
 - 制定后援和恢复策略, 负责故障恢复
- 改进方面:
 - 监督分析系统的性能 (空间利用率, 处理效率)
 - 数据库重组织, 物理上重组织, 以提高性能
 - 数据库重构, 设计上较大改动, 模式和内模式修改

第二章 实体-联系模型

期末考试提纲

- 实体、联系、属性
- 超码、候选码、主码
- 联系的种类、联系的势

- 弱实体、特化、概化、聚集
- E-R 图概念模型设计应用实例
- E-R 图和关系模式的相互转换

2.1 E-R 模型基本概念

E-R 模型: Entity-Relation Model.

- 世界是由一组称为实体的基本对象和这些对象之间的联系构成的.
- **实体 (Entity):** 客观存在并可相互区分的事物叫实体.
- 属性 (Attribute): 实体具有的某一特性称为属性. 用椭圆表示.
- 域 (Domain): 属性的取值范围称为域.
- 实体型 (Entity Type): 实体名 + 属性名集合. e.g. 学生 (学号, 姓名, 年龄, 性别, 系, 年级)
- 实体集 (Entity Set): 同类型的实体的集合. 用矩形表示.
- 联系 (Relationship): 实体之间的相互关联. 联系也可以有属性. 用联系表示.
- 联系的元 (Degree): 参与联系的实体集的个数.
 - 学生与学生的班长联系: 一元. 只有一个实体集.
 - 有时称一元联系为递归联系.
 - 联系是发生在实体集之间的还是实体型之间的? **实体集**.
- 实体集属性中作为主码的一部分的属性用下划线来标明.

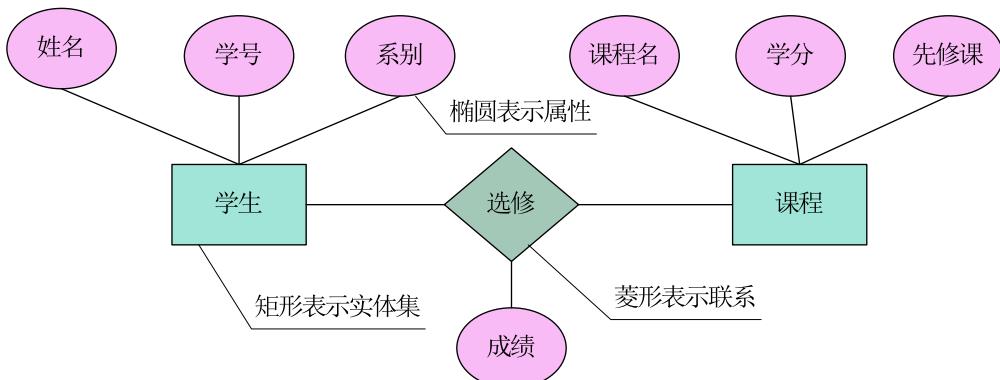


图 2.1: E-R 图

表达一元联系:

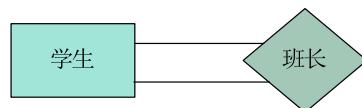


图 2.2: 一元联系

2.2 实体的码 (Key)

定义 2.1 (超码 (Superkey))

能唯一标识实体的属性或者属性组. 超码的任意超集也是超码.



定义 2.2 (候选码 (candidate key))

其任意真子集都不能成为超码的最小超码.



定义 2.3 (主码 (primary key))

选定的候选码作为实体的标识码.



定义 2.4 (替代码)

除去主码的其他候选码. (可以替代主码的候选码)



定义 2.5 (代理码)

人为添加的属性作为主码. e.g. ssn(单纯的随机码)



定义 2.6 (自然码)

e.g. 不重名的情况下, 姓名可以作为候选码. “真正的”候选码.



定义 2.7 (智能码)

经过编码的标识符. e.g. 学号, 电话号码, 身份证



2.3 属性

属性包括: 复合、多值、派生、NULL.

定义 2.8 (简单属性)

不可再分的属性. e.g. 学号, 姓名



定义 2.9 (复合属性 (Composite))

可以划分为更小的属性. e.g. 电话号码 = 区号 + 号码, 地址 = 省 + 市 + 区 + 街道 + 门牌号, 日期 = 年 + 月 + 日.



定义 2.10 (单值属性)

每一个特定的实体在该属性上的取值唯一.



定义 2.11 (多值属性)

多个一个的取值 (取值是一个集合. 必须展开.)



定义 2.12 (派生属性 (Derived))

可以从其他属性或者实体派生出来的值. e.g., 绩点可以通过成绩计算出来.

**定义 2.13 (NULL 属性)**

值存在但是目前没有获得其信息./ 当实体在某个属性上没有值时设为 NULL.

实体完整性: 主码取值不能为 NULL.

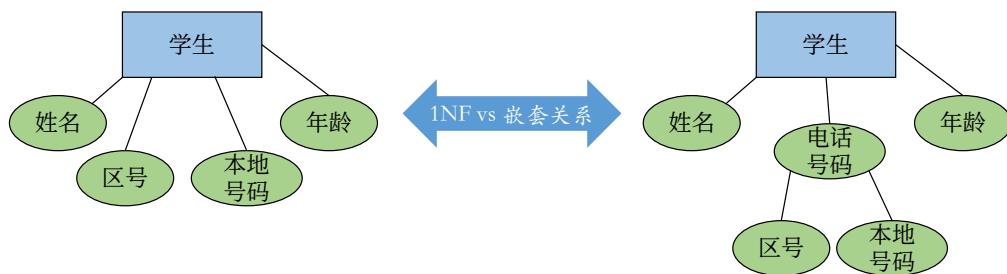
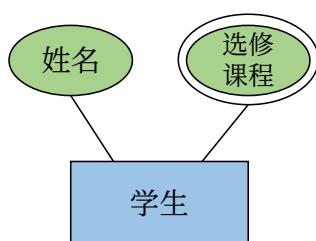


图 2.3: 简单属性 vs 复合属性

多值属性用**双椭圆**表示



派生属性用**虚椭圆**表示

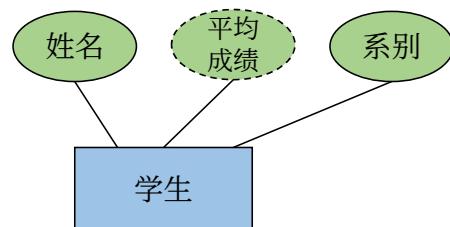


图 2.4: 多值属性和派生属性的表示

2.4 联系

注: 在多方实体和联系之间的线段上标注字母; 在单方实体和联系之间的线段上标注数字 1.

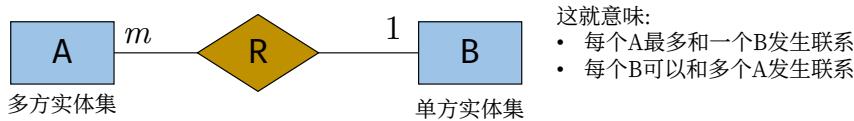
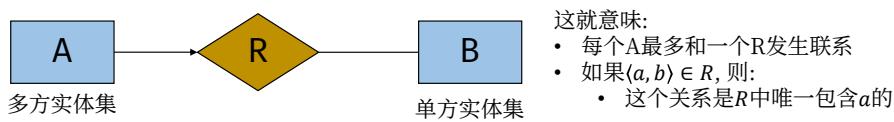
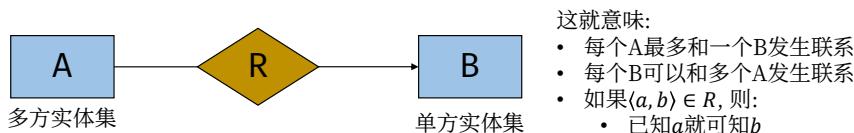


图 2.5: 联系种类在 E-R 图中的表示

定义 2.14 (联系的数量)

实体之间的**联系的数量**, 即一个实体通过一个联系集能与另一实体集相关联的实体的数目.

- 一对-的 (1:1)
- 一对多的 (1:m)
- 多对多的 (m : n)



用箭头或线段来表示联系的种类, 箭头指向单方实体集.

二元联系的种类:

定义 2.15 (一对-联系)

两个实体集 E_1, E_2 之间的一对-联系:

- E_1 中的一个实体与 E_2 中至多一个实体相联系;
- E_2 中的一个实体与 E_1 中至多一个实体相联系.

一对-不是一一对应!!!

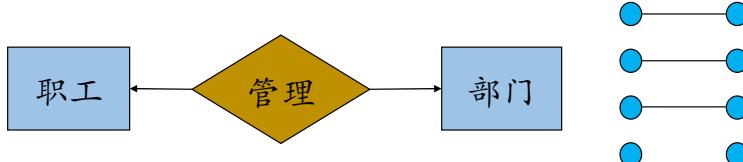


图 2.6: 一对-联系

定义 2.16 (一对多联系)

两个实体集 E_1, E_2 之间的一对多联系:

- E_1 中的一个实体与 E_2 中 $n(n \geq 0)$ 个实体相联系;
- E_2 中的一个实体与 E_1 中至多一个实体相联系.

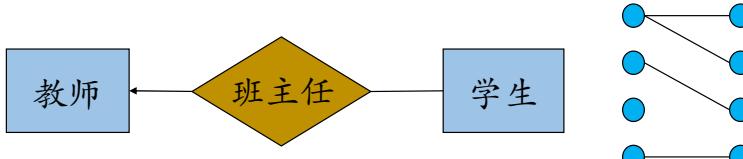


图 2.7: 一对多联系

定义 2.17 (多对多联系)

两个实体集 E_1, E_2 之间的多对多联系:

- E_1 中的一个实体与 E_2 中 $n(n \geq 0)$ 个实体相联系;
- E_2 中的一个实体与 E_1 中 $m(m \geq 0)$ 个实体相联系.

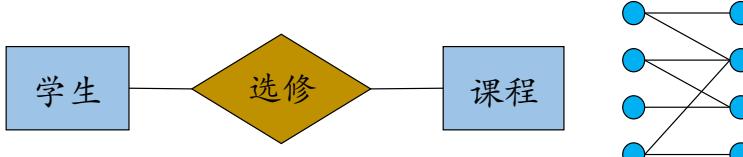


图 2.8: 多对多联系

一个实体集内的递归联系:

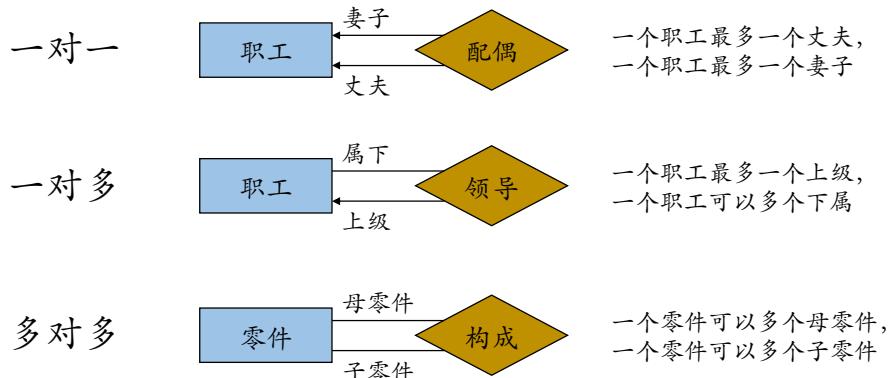


图 2.9: 一个实体集内的递归联系

定义 2.18 (多元联系)

当一个联系涉及三个或更多实体集时，我们称之为多元联系。

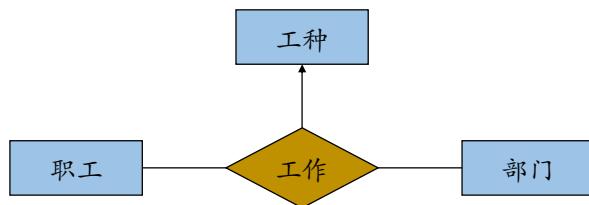


图 2.10: 多元联系

定义 2.19 (联系的势)

势表达了一个实体出现在联系中的次数。

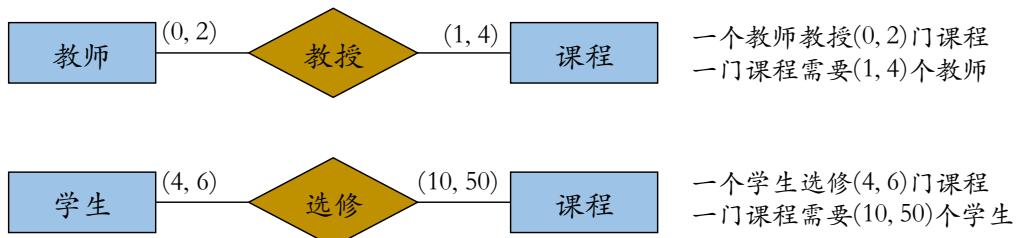


图 2.11: 联系的势

2.5 ER 设计实例

2.5.1 基于业务描述的 ER 设计实例

例题 2.1 考虑一个学校数据库，它要存储以下信息：

- 教师: 教工号、教工名、职称;
- 项目: 项目号、项目名称、起始年份、资助额;

- 学生: 学号、学生名、年龄、学位;
- 一个教工可以负责多个项目;
- 每个项目只能有一个负责人;
- 一个老师可以参与多个项目;
- 一个学生只能参与一个项目;
- 一个项目可以有多个学生和老师参与.

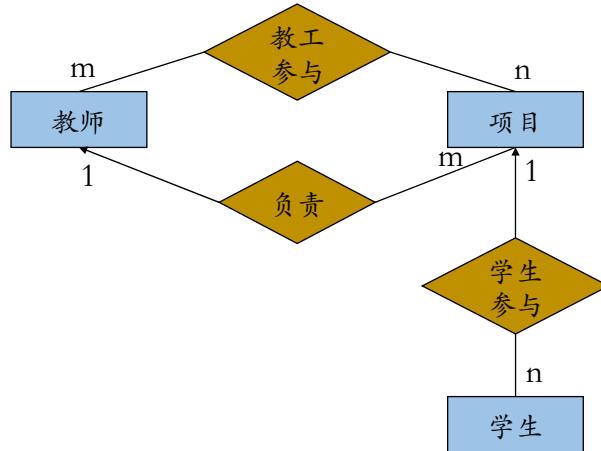


图 2.12: 基于业务描述的 ER 设计实例

2.5.2 由业务单据生成 ER 模型

卖家信息							
昵称: 米开朗旗舰店	真实姓名: 义乌市美智朗工艺品有限公司	城市: 浙江 金华					
联系电话:	邮件: m***	支 付 宝: 4***					
订单信息							
订单编号: 1149473762988781	支付宝交易号: 2015071021001001130292410113						
成交时间: 2015-07-10 16:25:05	付款时间: 2015-07-10 16:25:56	确认时间: 2015-07-21 08:51:32					
宝贝	宝贝属性	状态	单价(元)	数量	优惠	商品总价(元)	运费(元)
米开朗 纯净的理想 木质成人拼图1000片拼图儿童益智玩具情人节礼物	-	已确认收货	128.00	1	-	215.00 2860天猫积分抵28.60元	0.00 (快递)
米开朗木质成人拼图500/1000/1500/2000片静谧的祝福儿童益智玩具	颜色分类: 晚安 静静地祝福木质500片	已确认收货	176.00	1	-		
(可获返商城积分151点) ?							
实付款: 186.40 元							
物流信息							
收货地址: 陈立军, 13051836828, , 北京 北京市 海淀区 北京大学理科一号楼1628, 100871							
运送方式: 快递							
物流公司: 圆通速递							

图 2.13: 订单

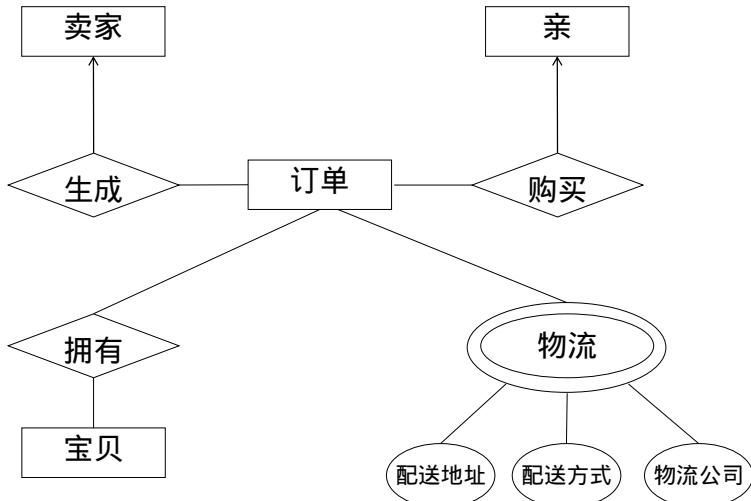


图 2.14: 由业务单据生成 ER 模型

2.6 其他概念术语

定义 2.20 (参与 (Participation))

实体集之间的关联称为参与, 即实体参与联系.

- E 全部参与 R : 实体集 E 中的每个实体都参与到联系集 R 中的至少一个联系.
- E 部分参与 R : 实体集 E 中只有部分实体参与到联系集 R 中的联系.

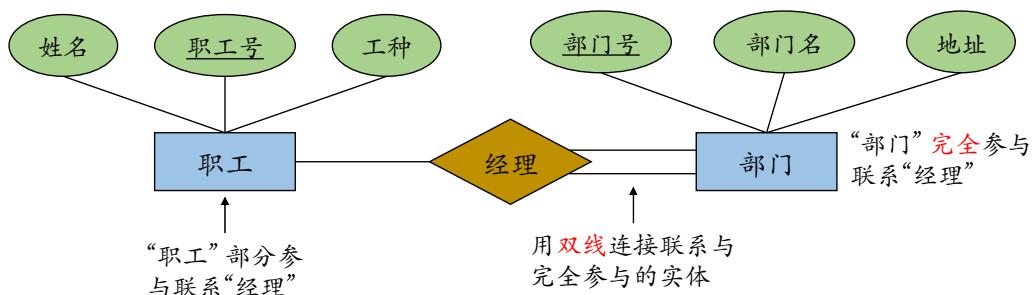


图 2.15: 参与的 ER 图表示

定义 2.21 (角色 (Role))

实体在联系中的作用称为实体的角色. 对于一元联系, 为区别各实体参与联系的方式, 需要显式指明其角色. 如图2.9.

定义 2.22 (存在依赖 (Existence Dependency))

x 存在依赖于 y 意味着实体 x 的存在依赖于 y , y 称为支配实体, x 称为从属实体.

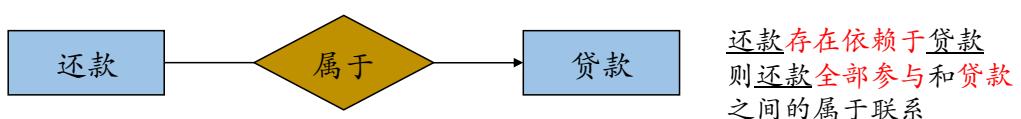


图 2.16: 存在依赖

复合实体. 一个 M:N 联系分解成两个 1:M. (了解即可.)

2.7 扩展 ER 特性

定义 2.23 (弱实体集 (weak entity set))

没有足够的属性以形成主码的实体集称为弱实体集.



定义 2.24 (标识性联系 (identifying relationship))

弱实体集与其标识实体集相连的联系称为标识性联系 (identifying relationship). 其实就是弱实体集和强实体集的那个联系.



注 弱实体集必然存在依赖于强实体集, 但是**存在依赖并不总会导致一个弱实体集**, 从属实体集可以有自己的主码.

定义 2.25 (分辨符 (Discriminator))

弱实体集中用于区别依赖于某个特定强实体集的属性集合, 也称作部分码 (partial key).



!!!!!! 弱实体集的主码 = 强实体集的主码 + 弱实体集的分辨符

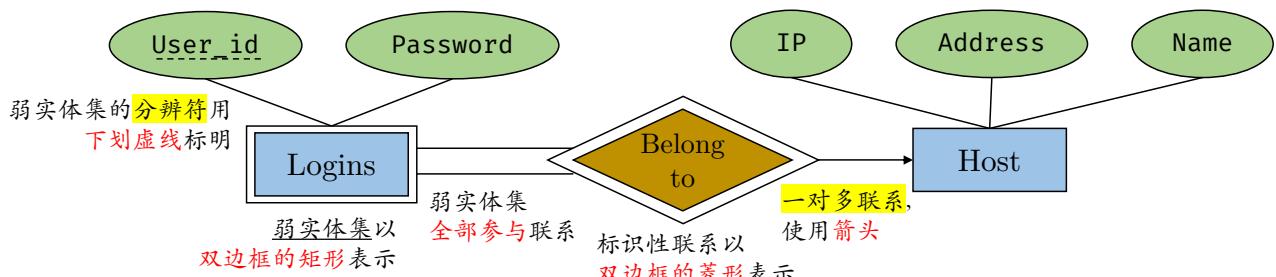


图 2.17: 弱实体集的 ER 表示

注 何时引入弱实体集?

- 作为层次结构的一部分.
- 实体集的一些多值、复合属性可以抽取出来作为弱实体集.
- 如果弱实体集不但参与和强实体集之间的标识性联系, 而且参与和其它实体集的联系, 或者弱实体集本身含有很多属性, 则将其表述为弱实体集.

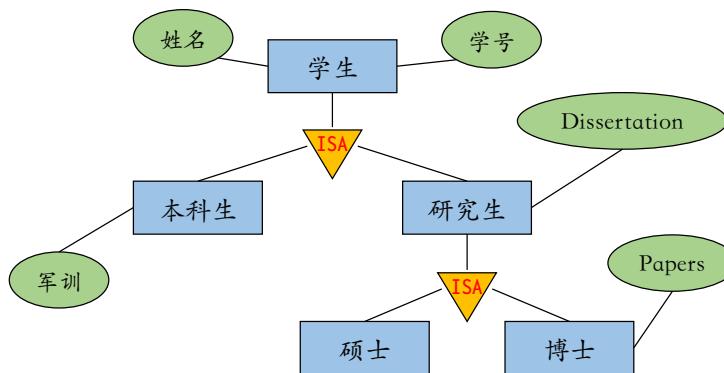


图 2.18: 特化的 ER 表示

定义 2.26 (特化)

实体集中某些子集具有区别于该实体集中其它实体的特性，可以根据这些差异特性对实体集进行分组，这一分组的过程称作特化。标识为 ISA 的三角形。ISA = “is a”。

**定义 2.27 (概化)**

概化是高层实体集与一个或多个底层实体集的包含关系。得到的 E-R 图和特化得到的 E-R 图是一样的，只不过是自底向上。



- 层次结构 (Hierarchy): 实体集作为低层实体集只能参与到一个 ISA 联系中。
- 格结构 (Lattice): 低层实体集可以参与到多个 ISA 联系中。

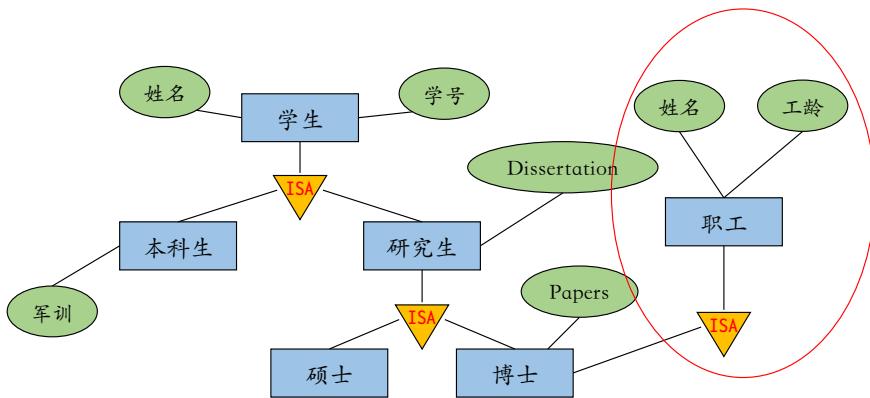


图 2.19: 格结构

概化中的成员身份:

- 不相交的 (Disjoint): 一个实体至多属于一个低层实体集。e.g., 学生只能参加一个项目组，学生就是不相交成员身份。
 - 有重叠的 (Overlapping): 同一实体可同时属于同一概化的多个低层实体集。e.g., 老师可以参加多个项目组，老师就是有重叠成员身份。
- 概化中的全部性约束: 确定高层实体集中一个实体是否必须属于至少一个低层实体集。
- 全部的 (Total): 每个高层实体必须属于一个低层实体集。e.g., 学生必须属于本科生或者研究生其中一种。
 - 部分的 (Partial): 允许一些高层实体不属于任何低层实体集。e.g., 学生可以不属于任何的项目组。

定义 2.28 (聚集)

两个先后动作的序列。

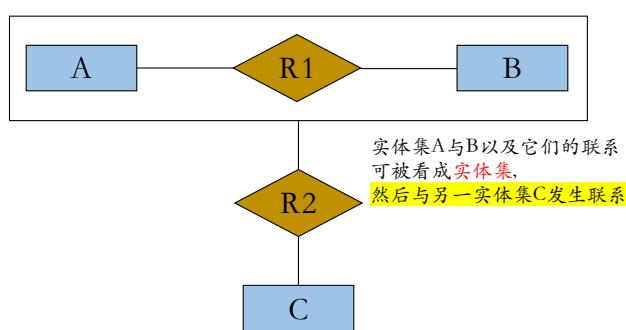


图 2.20: 聚集的 ER 表示

例题 2.2 能否把多元联系转换为若干个二元联系?

新构建一个**标识实体集** E , 构造三个新联系集 R_A, R_B, R_C , 对于每个 $(a_i, b_i, c_i) \in R$, 在 E 中创建一个 e_i , 然后在 R_A, R_B, R_C 中分别加入联系 $(e_i, a_i), (e_i, b_i), (e_i, c_i)$.

这样的转换并无实际意义.

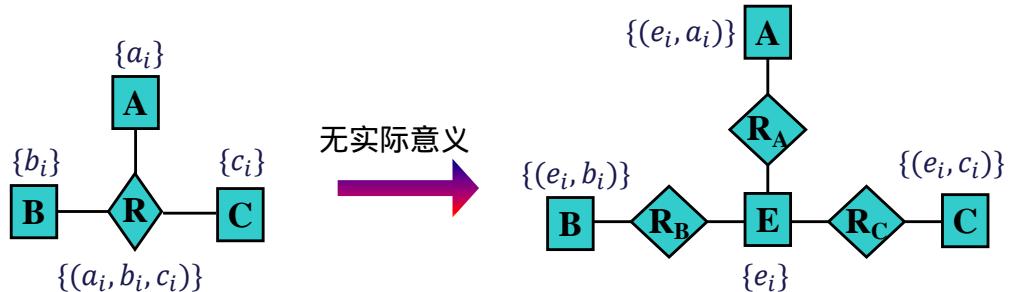


图 2.21: 无意义的转换

2.8 概念数据库设计过程

- 局部 ER 模式设计.
- 全局 ER 模式设计.
- 全局 ER 模式优化.

2.9 ER 模型向关系模式的转换

1. 实体 \rightarrow 关系.
2. 属性 \rightarrow 关系的属性.
 - (a). 复合属性定义为视图, 或者由应用定义, 这里摊平即可.
 - (b). 多值属性 \rightarrow 新的关系 + (取值 + 所在实体的码)
3. 一对多联系 \rightarrow 将单方参与实体的码作为多方参与实体的属性.
4. 多对多联系 \rightarrow 将联系定义为新的关系, 属性为参与双方的码.
5. 一对一联系 \rightarrow 若联系双方均部分参与, 则将联系定义为一个新的关系, 属性为参与双方的码. 全部参与同一对多.
6. 弱实体集 \rightarrow 弱实体集所对应的关系的码由弱实体集本身的分辨符再加上所依赖的强实体集的码.
7. 概化 \rightarrow 高层实体集和低层实体集分别转为表, 低层实体集所对应的关系包括高层实体集的码.
8. 聚集 \rightarrow 实体集 A 与 B 及其联系 R 被抽象成实体集 C, C 与另一实体集 D 构成联系 S, 则 S 的码由 C 和 D 的码构成.

2.10 关系模式向 ER 的转换

! 识别关系间的重合属性.

第三章 关系模型

期末考试提纲

- | | |
|--------------------------------------|--|
| <input type="checkbox"/> 笛卡尔积 | <input type="checkbox"/> 关系运算 |
| <input type="checkbox"/> 关系模型三要素 | <input type="checkbox"/> 元组演算表达式 |
| <input type="checkbox"/> 关系模型的完整性 | <input type="checkbox"/> 域关系演算表达式 |
| <input type="checkbox"/> 全关系系统的十二条准则 | <input type="checkbox"/> 用关系代数表达关系数据操作 |

3.1 关系基本概念

定义 3.1 (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 3.2 (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 3.3 (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.

3.2 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.

3.2.1 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 3.4 (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 3.5 (主属性)**

任何一个候选码中的属性被称为主属性.

**定义 3.6 (主码 (Primary 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 3.7 (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 3.8 (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 3.9 (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 3.10 (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).



3.2.2 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.

3.2.3 数据完整性

定义 3.11 (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 3.12 (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 3.13 (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.

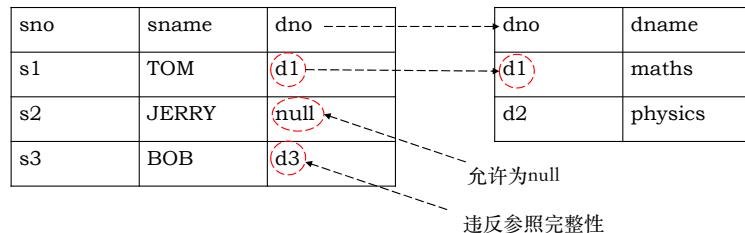


图 3.1: 参照完整性

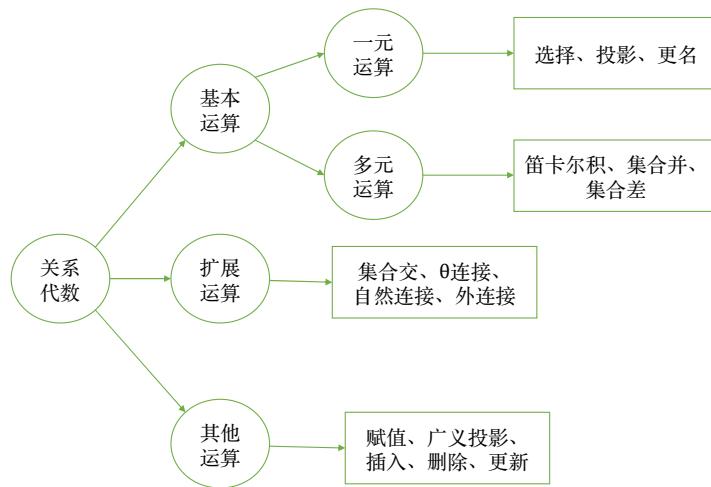
定义 3.14 (用户完整性)

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.

3.3 关系代数运算



3.3.1 基本关系代数运算

3.3.1.1 一元运算

定义 3.15 (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 3.16 (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:

R		
A	B	C
a	b	c
d	e	f
c	b	c

$\Pi_{B,C}(R)$

B	C
b	c
e	f

图 3.2: 投影运算要去掉相同的行

定义 3.17 (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



3.3.1.2 多元运算

定义 3.18 (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 3.1 选修了 001 号或 002 号课程的学生号.

$$\Pi_{sno}(\sigma_{cno=001 \vee cno=002}(SC))$$

$$\Pi_{sno}(\sigma_{cno=001}(SC) \vee \sigma_{cno=002}(SC))$$

定义 3.19 (差运算)

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



例题 3.2 求选修了 001 号但未选修 002 号课程的学生号.

$$\Pi_{sno}(\sigma_{cno=001}(SC)) - \Pi_{sno}(\sigma_{cno=002}(SC))$$

$$\Pi_{sno}(\sigma_{cno=001 \vee cno \neq 002}(SC))$$

定义 3.20 (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:

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



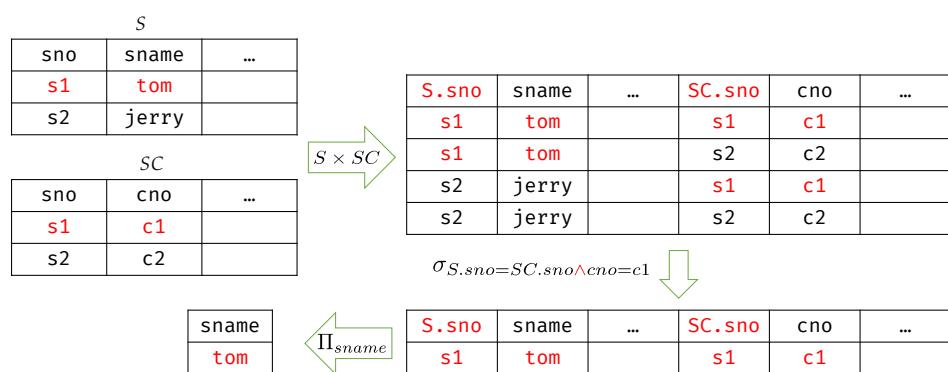
定义 3.21 (笛卡尔积)

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

- $R \times S$ 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 3.3 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 3.3: 笛卡尔积的使用

例题 3.4 求数学成绩比王红同学高的学生姓名.

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

3.3.2 扩展关系代数运算

定义 3.22 (交运算)

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

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



例题 3.5 求同时选修了 001 号和 002 号课程的学生号.

$$\Pi_{\text{sno}} (\sigma_{\text{cno}=001}(\text{SC})) \cap \Pi_{\text{sno}} (\sigma_{\text{cno}=002}(\text{SC}))$$

$$\Pi_{\text{sno}} (\sigma_{\text{cno}=001 \wedge \text{cno}=002}(\text{SC}))$$

定义 3.23 (θ 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, θ 为算术比较符.

例题 3.6 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 3.24 (等值连接)

θ 为等号的时候为等值连接.



定义 3.25 (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:

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

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.



A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

⊗

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ϵ

=

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

图 3.4: 自然连接例子

例题 3.7 求出 001 号学生所在系的名称:

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

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

例题 3.8 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 3.1: 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 3.26 (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_R .
3. 全外连接 = 左外连接 + 右外连接. \bowtie_{LR} .

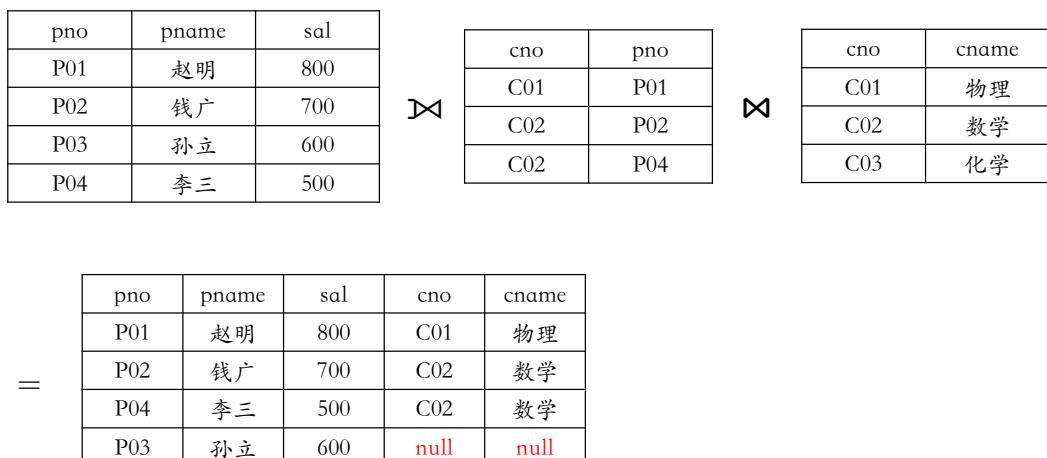


图 3.5: 左外连接示意图

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 3.6: 外连接结合律不成立的反例

定义 3.27 (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 3.28 (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. $\overline{\bowtie}$.

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		
A	B	C
a	b	c
d	b	c
c	a	d

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

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

图 3.7: 半连接示意图

定义 3.29 (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				A	B	C		
a	b		c		b		c		
c	d		a	=	a	b	c		
			d		c	d	null		
					null	a	d		

R			S			=					
A	B	C					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 3.8: 外部并示意图

定义 3.30 (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 3.9 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.

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

定义 3.31 (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:

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

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

除法的计算表达式为:

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



例题 3.10 对于 $SC(sno, cno, grade)$, 求选修了所有课程的学生.

$$\Pi_{sno, cno}(SC) \div \Pi_{cno}(C)$$

现在对于除运算中的被除关系要正确投影:

也就是我们考虑下面的两个式子:

$$\Pi_{sno, cno}(SC) \div \Pi_{cno}(C)$$

$$\Pi_{sno}(SC \div \Pi_{cno}(C))$$

它们对应着下面的图:

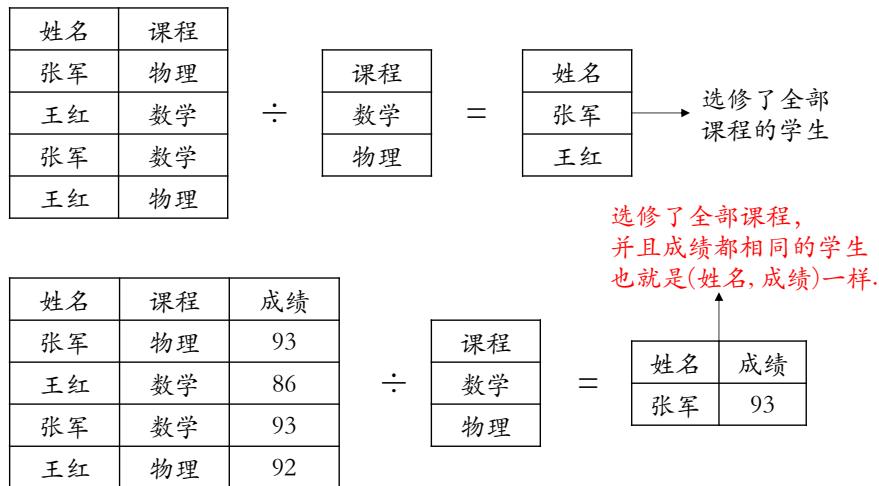


图 3.9: 对于除运算中的被除关系要正确投影

3.3.3 关系代数更新运算

定义 3.32 (赋值运算)

为使查询表达简单、清晰, 可以将一个复杂关系代数表达式分成几个部分, 每一部分都赋予一个临时关系变量, 该变量可被看作关系而在后续表达式中使用.

临时关系变量 \leftarrow 关系代数表达式

现在我们可以把对于除法的计算式子写成:

$$\begin{aligned} temp1 &\leftarrow \Pi_X(R) \\ temp2 &\leftarrow \Pi_X(temp1 \times \Pi_Y(S) - R) \\ result &\leftarrow temp1 - temp2 \end{aligned}$$

定义 3.33 (广义投影)

在投影列表中使用算术表达式来对投影进行扩展

$$\Pi_{F_1, F_2, \dots, F_n}(E)$$

其中, F_1, F_2, \dots, F_n 是算术表达式.

例题 3.11 求员工应该缴纳的所得税:

$$\Pi_{pno, sal \times 0.05}(PROF)$$

数据库修改: **删除**.

- 将满足条件的元组从关系中删除, 就是对永久关系的赋值运算, 也就是下面的式子:

$$R \leftarrow R - E$$

例题 3.12 删除 001 号老师所担任的课程.

$$PC \leftarrow PC - \sigma_{pno=001}(PC)$$

例题 3.13 删除没有选课的学生.

$$S \leftarrow S - (\Pi_{sno}(S) - \Pi_{sno}(SC)) \bowtie S$$

数据库修改: 插入.

- 插入一个指定的元组, 或者插入一个查询结果.

$$R \leftarrow R \cup E$$

例题 3.14 加入计算机系学生选修“数据结构”的信息.

$$SC \leftarrow SC \cup (\Pi_{sno}(S \bowtie \sigma_{dname="计算机系"}(DEPT)) \times \Pi_{cno}(\sigma_{name="数据结构"}(C)) \times \{null\})$$

数据库修改: 更新.

- 利用广义投影改变元组的某些属性上的值.

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

例题 3.15 给每位老师上调 10% 的工资.

$$PROF \leftarrow \Pi_{pno, pname, sal \times 1.1, dno}(PROF)$$

3.4 关系代数查询实例

问题 3.1 求没有选修 c1 号课程的学生.

解答. 这样我们需要的求法是: 所有学生 - 选修了 c1 号课程的学生. 也就是:

$$\Pi_{sno}(SC) - \Pi_{sno}(\sigma_{cno=c1}(SC)).$$

问题 3.2 求仅选修了 c1 号课程的学生号.

解答. 选修 c1 号课程的学生 - 仅选 c1 号课程之外的学生.

$$\Pi_{sno}(\sigma_{cno=c1}(SC)) - \Pi_{sno}(\sigma_{cno \neq c1}(SC)).$$

问题 3.3 求选修 c1 同时又选修其他课程的学生.

解答. 在 SC 这个表里排除掉所有的 c1 记录, 投影一下就得到了除了 c1 还选了别的课的学生. 那么在选了 c1 的学生中除去这部分即可.

$$\Pi_{sno}(\sigma_{cno=c1}(SC)) - \Pi_{sno}(SC - \sigma_{cno=c1}(SC)).$$

问题 3.4 求选修 c1 课程比 s1 学生的该门课程成绩高的学生.

解答. 我们使用两个关系的笛卡尔积, 然后选出 $R.grade < S.grade$ 的元组即可.

$$\Pi_{S.sno}(\sigma_{R.sno=s1 \wedge R.cno=c1 \wedge R.grade < S.grade}(\rho_R(SC) \times \rho_S(SC))).$$

问题 3.5 求每门课程的先修课的先修课.

解答. 我们依然考虑使用笛卡尔积:

cno	pcno	 计算 $C \times C$, 保留这样的元组: (c3, c2, c1)
c1	null	
c2	c1	
c3	c2	

图 3.10: 先修课的先修课

那么最后就是:

$$\Pi_{C.cno, R.pcno}(\sigma_{C.pcno=R.cno}(C \times \rho_R(C))).$$

问题 3.6 求选修了至少两门课的学生.

解答. 依然使用笛卡尔积进行一些杂糅, 然后找到里面的 ($c_1 \neq c_2$) 的元组:

$$\Pi_{R.sno}(\sigma_{R.sno=S.sno \wedge R.cno \neq S.cno}(\rho_R(SC) \times \rho_S(SC))).$$

注 如果是“求选修了至少 N 门课的学生”, 应该需要进行 N 次的笛卡尔积.

如果是“求只选修了 1 门课的学生”, 只需要选课的学生 - 选修了至少两门课的学生.

问题 3.7 求最低的成绩.

解答. 只需要: 使用笛卡尔积, 找到所有的有元组的成绩比它更小的元组, 全体元组减去即可. 剩下的是没有人会比它更小的元组, 也就是最小的元组.

$$\Pi_{grade}(SC) - \Pi_{S.grade}(\sigma_{R.grade < S.grade}(\rho_R(SC) \times \rho_S(SC))).$$

问题 3.8 求选修课程中包含了所有 01 号学生所选修课程的学生号.

解答. 这种“……包含了……所有……”的形式的题目, 使用除法比较好.

$$\Pi_{sno,cno}(SC) \div \Pi_{cno}(\sigma_{sno=s01}(SC)).$$

问题 3.9 求其选修课程被 s01 号学生所修课程包含的学生号.

解答. 第一步先利用笛卡尔积找出和 s01 号学生有选修课程相同的学生, 第二步使用全体选课记录减去这些和 s01 号选课重叠的部分, 剩下的是不重叠的部分(当然这里包括了 s01 本身的选课记录). 最后除去不重叠的, 剩下就是完全被 s01 的选课包含的学生了.

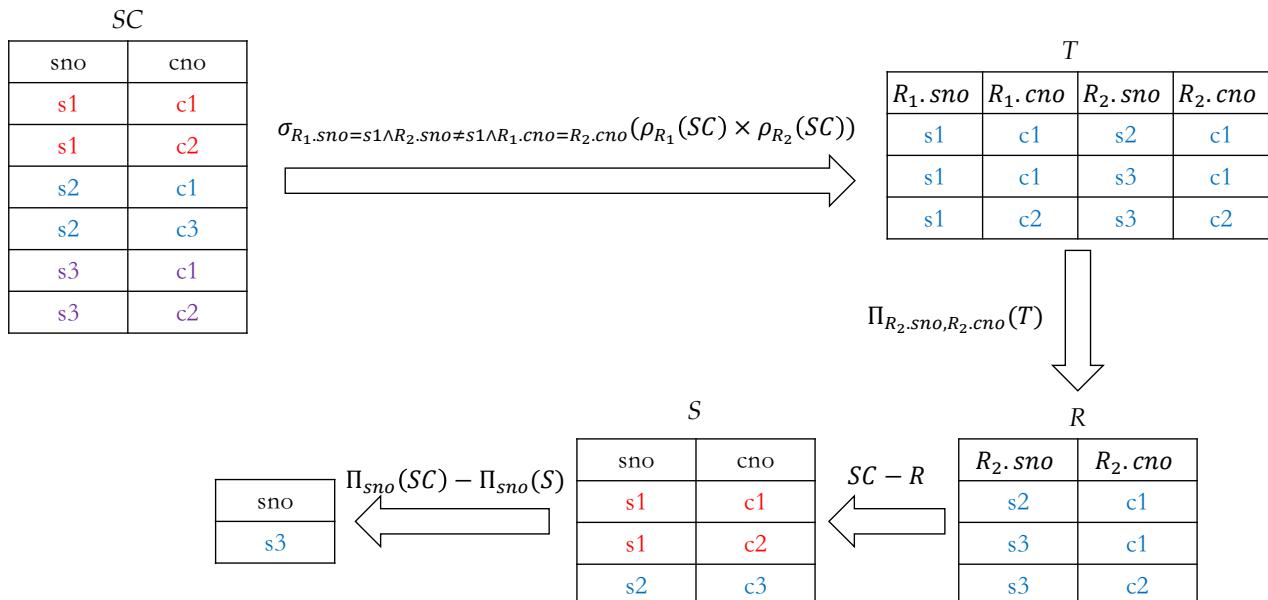


图 3.11: 问题解答

问题 3.10 对于关系 $stock(sno, date, price)$, 找出一直上涨的股票和上涨过的股票.

解答. 上涨过的股票: if $d1 > d2$ then $p1 > p2$. 从而有:

$$\Pi_{R.sno}(\sigma_{R.sno=S.sno \wedge R.date > S.date \wedge R.price > S.price}(\rho_R(stock) \times \rho_S(stock))).$$

现在一直上涨的股票 = 所有的股票 - 下跌的股票. 而下跌过的股票也可以使用上面的方法计算得到.

问题 3.11 根据 $guanxi(source, destination)$, 找出互相认识的人.

解答. 使用笛卡尔积, 找到 $R.destination = S.source$, 这样就是互相认识的人.

3.5 关系演算

ps. 鉴于 PPT 里出现了若干次“验算”和若干次“演算”，下面我统一用“演算”这个词（我没去上课，不知道正确的这个词的用法如何。）

3.5.1 元组关系演算

定义 3.34 (元组关系演算)

形式化的定义为 $\{t | P(t)\}$, 表示所有使谓词 P 为真的元组集合. 这里的 P 是公式, 由原子公式和运算符组成. 这里的 t 为元组变量, 如果存在全程或者存在量词, 则为约束变量, 否则为自由变量.



定义 3.35 (原子公式)

原子公式包括下面的几个公式:

- $s \in R$: s 是关系 R 的一个元组;
- $s[X] \theta c$: 分量 $s[X]$ 与常量 c 之间满足比较关系 θ ;
- $s[X] \theta u[Y]$: 分量 $s[X]$ 与分量 $u[Y]$ 之间满足比较关系 θ .



定义 3.36 (公式的递归定义)

- 原子公式是公式;
- 若 P 是公式, 则 $\neg P$ 也是公式;
- 若 P_1, P_2 是公式, 则 $P_1 \wedge P_2, P_1 \vee P_2, P_1 \Rightarrow P_2$ 也是公式;
- 若 $P(t)$ 是公式, R 是关系, 则 $\exists t \in R(P(t)), \forall t \in R(P(t))$ 也是公式.



例题 3.16 找出工资在 800 元以上的老师.

$$\{t | t \in PROF \wedge t[SAL] > 800\}.$$

例题 3.17 找出工资在 800 元以上的老师的姓名.

$$\{t | \exists s \in PROF (t[PNAME] = s[PNAME] \wedge s[SAL] > 800)\}.$$

这里的 t 实际上只有一个列 PNAME!!!!!!

例题 3.18 给出计算机系老师的姓名.

$$\{t | \exists u \in DEPT \exists s \in PROF (u[DNOME] = "计算机系" \wedge s[DNO] = u[DNO] \wedge t[PNAME] = s[PNAME])\}.$$

例题 3.19 求选修了全部课程的学生号.

$$\{t | \exists u \in S \forall v \in C (\exists w \in SC (w[cno] = w[cno] \wedge u[sno] = w[sno] \wedge t[sno] = u[sno]))\}.$$

例题 3.20 求选修了 s1 同学所修全部课程的学生号. 这句话的意思就是说: 对任意课程, s1 选了 \Rightarrow s2 也选了.

$$\left\{ t \left| \begin{array}{l} \exists u \in S (t[sno] = u[sno]) \wedge \forall v \in C \\ (\exists w_1 \in SC (w_1[cno] = v[cno] \wedge w_1[sno] = s1)) \\ \Rightarrow \exists w_2 \in SC (w_2[cno] = v[cno] \wedge w_2[sno] = u[sno])) \end{array} \right. \right\}$$

元组关系演算与关系代数的等价性:

- **投影:** $\Pi_A(R) = \{t | \exists s \in R (s[A] = t[A])\}$
- **选择:** $\sigma_{F(A)}(R) = \{t | t \in R \wedge F(t[A])\}$
- **广义笛卡尔积:** $R(A) \times S(B) = \{t | \exists u \in R \exists s \in S (t[A] = u[A] \wedge t[B] = s[B])\}$
- **并:** $R \cup S = \{t | t \in R \vee t \in S\}$
- **交:** $R \cap S = \{t | t \in R \wedge t \in S\}$

- **差:** $R - S = \{t \mid t \in R \wedge \neg(t \in S)\}$

元组关系演算有可能会产生无限关系, 这样的表达式是不安全的. 我们可以人为的定义公式 P 的域的概念, 用 $\text{dom}(P)$ 来表示.

定义 3.37 (安全性)

如果出现在表达式 $\{t|P(t)\}$ 结果中的所有值均来自 $\text{dom}(P)$, 则称 $\{t|P(t)\}$ 是安全的.



3.5.2 域关系演算

定义 3.38 (域关系演算)

形式化定义为 $\{\langle x_1, x_2, \dots, x_n \rangle | P(x_1, x_2, \dots, x_n)\}$. 其中 x_i 代表域变量, P 为由原子组成的公式.

- $\langle x_1, x_2, \dots, x_n \rangle \in R$;
- $x \theta c$;
- $x \theta y$;



例题 3.21 找出工资在 800 元以上的老师.

$$\{\langle a, b, c, d, e \rangle | \langle a, b, c, d, e \rangle \in \text{PROF} \wedge e > 800\}.$$

例题 3.22 找出工资在 800 元以上的老师的姓名.

$$\{\langle b \rangle | \exists a, c, d, e (\langle a, b, c, d, e \rangle \in \text{PROF} \wedge e > 800)\}$$

例题 3.23 给出计算机系老师的姓名.

$$\{\langle b \rangle | \exists l, m, n, s, a, c, d, e (\langle l, m, n, s \rangle \in \text{DEPT} \wedge \langle a, b, c, d, e \rangle \in \text{PROF} \wedge m = \text{“计算机系”} \wedge l = d)\}$$

采用域关系验算的实际查询系统: QBE.

定义 3.39 (包 (Bag))

允许重复的集合, 或者多集 (multi-set).



3.6 关系系统

1. 表式系统: 仅支持关系 (即表) 数据结构, 不支持集合级操作. 表式系统不能算关系系统.
2. 最小关系系统: 仅支持关系数据结构和三种关系操作. 许多微机关系数据库系统如 FoxPro 属于此类.
3. 完备关系系统: 支持关系数据结构和所有的关系代数操作. 90 年代初的许多关系数据库管理系统属于这一类.
4. 全关系系统: 这类系统支持关系模型的所有特征.

3.6.1 全关系系统的十二条准则

- 准则 0: 一个关系型的 DBMS 必须能完全通过它的关系能力来管理数据库.
- 准则 1: 信息准则. 关系型 DBMS 的所有信息都应在逻辑一级上用一种方法即表中的值显式地表示.
- 准则 2: 保证访问准则. 依靠表名、主码和列名的组合, 保证能以逻辑方式访问关系数据库中的每个数据项 (分量值).
- 准则 3: 空值的系统化处理. 全关系型 DBMS 应支持空值概念, 并用系统化的方式处理空值.
- 准则 4: 基于关系模型的动态的联机数据字典. 数据库的描述在逻辑级上应该和普通数据采用同样的表示方式, 使得授权用户可以使用查询一般数据所用的关系语言来查询数据库的描述信息.

- 准则 5: 统一的数据子语言. 一个关系系统可以有几种语言和多种终端使用方式 (如 QBE、嵌入式 SQL), 但必须有一种语言, 它的语句可以表示为具有严格语法规规定的字符串.
- 准则 6: 视图更新准则. 所有理论上可更新的视图也应该由系统更新, 即对视图的更新要求, 存在一个算法可以无二义地把更新要求转换为对基本表的更新序列.
- 准则 7: 高级的插入、删除和修改操作. 关系系统的操作对象是单一的关系.
- 准则 8: 数据物理独立性. 无论数据库的数据在存储表示和存取方法上作任何变化, 应用程序和终端活动都保持逻辑上的不变性.
- 准则 9: 数据逻辑独立性. 当对基本关系进行理论上信息不受损害的任何改变时, 应用程序和终端活动都保持逻辑上的不变性.
- 准则 10: 数据完整性的独立性. 关系数据库的完整性约束条件必须是用数据库语言定义并存储在数据字典中的, 而不是由应用程序加以定义.
- 准则 11: 分布独立性. 分布独立性是指 DBMS 具有这样的数据库语言, 使得应用程序和终端活动在下列情况下都保持逻辑上的不变性:
 - 在第一次引入分布式数据时, 即如果原来的 DBMS 只管理非分布式的数据, 而现在引入了分布式数据;
 - 当数据重新分布时, 即如果原来 DBMS 能管理分布式数据, 现在要改变原来的数据分布.
- 准则 12: 无破坏准则. 如果一个关系系统具有一个低级 (一次一记录) 语言, 则这个低级语言不能违背或绕过完整性准则.
 - 为获得完整性的独立性, 需要让完整性约束条件和数据的逻辑结构相独立;
 - 不能旁路 (bypass) 或者关闭约束检查子系统.

第四章 SQL

期末考试提纲

- | | |
|-----------------|---|
| □ 了解数据类型、表的定义 | □ 能用 SQL 表达查询操作 |
| □ 掌握各种索引的定义及其作用 | □ 表连接、分组聚集、集合、嵌套子查询、null、游标、with 定义临时视图 |
| □ 掌握各种查询表达操作的含义 | |

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 4.1: SQL 主要操作符

text2SQL: 建立自然语言与结构化数据之间的关系。

4.1 数据定义

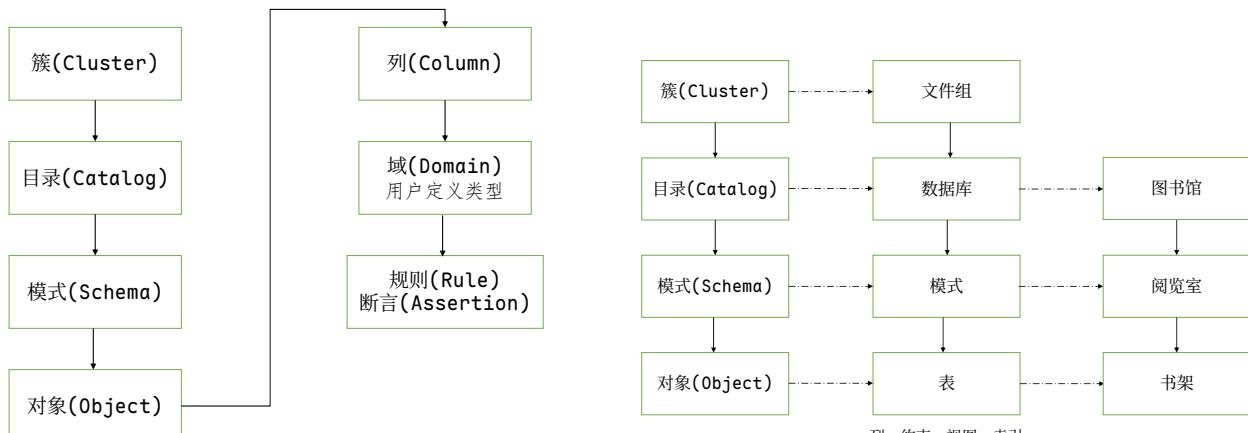


图 4.1: 标准 SQL 中的数据定义对象

图 4.2: 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>.

4.1.1 数据模式定义

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:

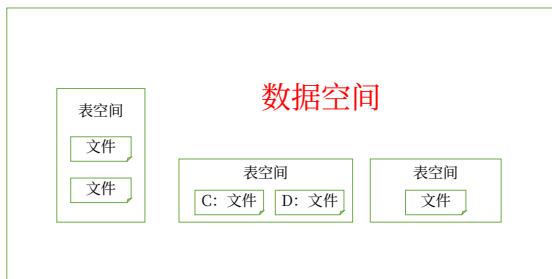


图 4.3: MySQL 表空间

```
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 4.2: 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 4.3: 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.

4.1.2 数据类型

数据类型	范围	unsigned 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 4.4: 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.

数据类型	描述	unsigned 范围	存储字节数
<code>float(m, d)</code>	单精度浮点数 <i>m</i> 是总位数 <i>d</i> 是小数点后位数	...	4 字节
<code>double(m, d)</code>	双精度浮点数 <i>m</i> 是总位数 <i>d</i> 是小数点后位数	...	8 字节
<code>decimal(m, d)</code> <code>numeric</code>	精确小数 <i>m</i> 是总位数 <i>d</i> 是小数点后位数	最大位数 <i>m</i> 为 65 最大支持小数为 <i>d</i> 为 30	
<code>float(n)</code>	$1 \leq n \leq 24$ 是 4 字节 $25 \leq n \leq 53$ 是 8 字节

表 4.5: 定点数与浮点数

`money` 使用 4 位小数存储数据,容易发生小数的舍入错误.

数据类型	范围	描述	存储字节数
char(<i>n</i>)	0 ~ 255	定长字符串	4 字节
varchar(<i>n</i>)	0 ~ 65,535	变长字符串	实际字符串长度
tinytext	0 ~ 2^8		
text	0 ~ 2^{16}		
mediumtext	0 ~ 2^{24}		
longtext	0 ~ 2^{32}		

表 4.6: 字符型

```
-- 创建表时指定字符集和校对规则
CREATE TABLE users (
    name VARCHAR(100)
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
-- ci 表示不区分大小写

-- 查看服务器字符集
SHOW VARIABLES LIKE 'character_set%';
-- 查看数据库字符集
SHOW CREATE DATABASE mydb;
-- 查看表字符集
SHOW CREATE TABLE users;
```

类型名称	日期格式	日期范围	存储需求
year	YYYY	1901 ~ 2155	1 个字节
time	HH:MM:SS	-838:59:59 ~ 838:59:59	3 个字节
date	YYYY-MM-DD	1000-01-01 ~ 9999-12-31	3 个字节
datetime	YYYY-MM-DD HH:MM:SS	1000-01-01 00:00:00 ~ 9999-12-31 23:59:59	8 个字节
timestamp	YYYY-MM-DD HH:MM:SS	1980-01-01 00:00:01 UTC ~ 2040-01-19 03:14:07 UTC	4 个字节

表 4.7: 日期类型及其描述

数据类型	描述
enum('v1', 'v2', ...)	只能取一个值
set('v1', 'v2', ...)	可以取多个值

表 4.8: 枚举型

```
create table test_enum(
    a enum('男', '女'),
    b set('1', '2', '3', '4')
)
insert into test_enum values ('男', '2,4')
```

数据类型	大小
binary(n)	255
varbinary(n)	16384
tinyblob, blob, mediumblob, longblob	256, 16K, 16M, 4G

表 4.9: 二进制类型

下面是几个关于数据类型的问题:

1. char 还是 varchar?

特性	CHAR	VARCHAR
存储方式	固定长度	可变长度
存储效率	可能浪费空间 (固定长度填充空格)	高效 (仅存储实际数据 + 长度信息)
检索速度	更快 (直接定位)	稍慢 (需计算长度)
适用场景	固定长度数据 (如代码、哈希值)	可变长度数据 (如姓名、地址)
空格处理	自动填充空格 (需注意业务逻辑)	不填充空格

2. 显式数据类型转换:

```
cast ( 表达式 as 数据类型 [(数据长度)] )
convert ( 数据类型 [(数据长度)], 表达式 [, 输出样式] )
select cast( 123.45 as decimal(10,4) )      -- 输出结果: 123.4500
select cast( 123.45 as decimal(10,1) )      -- 输出结果: 123.5
select cast( 12.34567 as money )           -- 输出结果: 12.3457
select 'My age is: ' + cast( 28 as char(4) ) -- 输出结果: My age is: 28
select cast( cast( 123.45 as int ) as char(10) ) -- 输出结果: 123
select convert( varchar(30), getdate(), 106 ) -- 输出结果: 17 08 2012
select convert( varchar(30), getdate(), 110 ) -- 输出结果: 08-17-2012
select cast( 'SQL' as binary(3) )           -- 输出结果: 0x53514C
```

3. 隐式数据类型转换: 执行 select 1 + '1', 结果会是 2; 执行 select 1 + 'a', 会显示类型转换错误.

4. 特殊类型: XML, JSON, Array, 空间数据.

5. 用户定义数据类型 UDDT(User-defined Datatype).

- 标准 SQL: create domain 域名 数据类型;
- SQL Server: create type phone_number from varchar(20) not null.
- Oracle:

```
create type animal_ty as OBJECT(
    breed varchar2(25),
    name varchar2(25),
    birthday date
)
```

6. 冗长主码的危害: 占据很大的表空间; 减少索引项, 增加了磁盘 I/O 数, 减慢索引速度.

MySQL 中的自增字段:

```
create table test_incr(
    id bigint auto_increment,
    name char(10)
)
```

SQL Server 中的序列号: identity.

```
create table customer(
    cust_id smallint identity(100, 20) not null,
    -- identity 有一个起始数和增量值
    cust_name varchar(50) not null
)
```

SQL Server 中的序列号: sequence.

```
create sequence mySeq as int start with 1 increment by 1
insert into myTb(id, Name) values (next value for mySeq, 'Tom')
insert into myTb(id, Name) values (next value for mySeq, 'jerry')
```

uniqueidentifier 产生跨数据库和服务器的全局唯一标识符 (GUID), newid() 函数产生 uniqueidentifier 类型的值, newsequentialid() 产生的 GUID 总是大于先前通过该函数生成的 GUID.

如果表中有一列被声明为 rowversion, 只要行被修改, 其 rowversion 列就会发生改变. 它是跨表唯一的, 任何表的修改都会使该值递增.

4.1.3 索引

定义 4.1 (索引)

索引是数据库管理系统中用于加速数据检索的核心工具. 它通过创建特定数据结构 (如 B 树、哈希表等), 将数据表中的某些列的值与对应的物理存储位置关联起来, 从而大幅提升查询效率.



索引的作用:

- 加速查询;
- 支持排序和分组;
- 保证唯一性;
- 优化连接操作.

```
create [unique] index 索引名
[using { btree | hash }]
on 表名 (列名 [asc/desc] [, 列名asc/desc] ...)

drop index 索引名
```

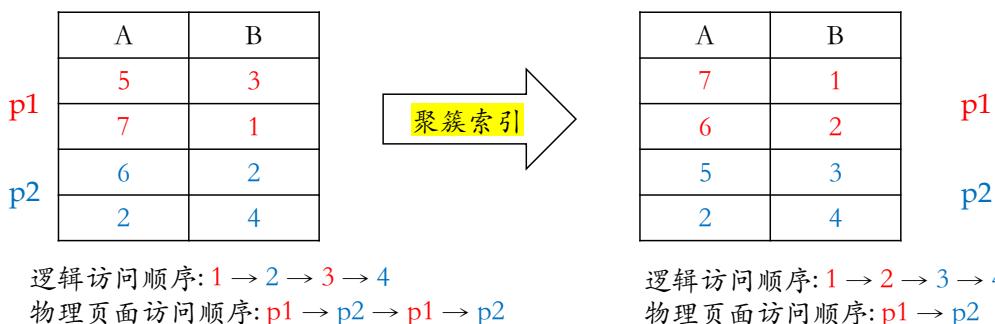


图 4.4: 聚簇索引

定义 4.2 (索引碎片)

索引碎片：页面逻辑顺序与物理顺序不一致.



下面是若干实现索引的方法:

- 聚簇索引 (cluster): 表中元组按索引项的值排序并物理地聚簇在一起. 聚簇索引使得逻辑访问顺序和物理存储顺序尽可能一致. 如图4.4.
- 组合索引: 建立在多个属性列上的索引.

索引 (A, B, C) 的 B+ 树会先按 A 排序, 再在 A 相同的情况下按 B 排序, 最后在 B 相同的情况下按 C 排序. 必须满足最左前缀原则: 即查询条件必须从索引的最左列开始连续匹配.

假如我们针对索引 (user_id, name) 的查询:

```
SELECT user_id, name FROM users WHERE user_id = 100;
```

那么可以直接从索引获取数据, 无需访问表数据.

但是查询 `SELECT * FROM users WHERE user_id = 100;` 需回表查询所有列.

下面是描述索引的性质:

- 覆盖索引: 覆盖索引的核心是“索引覆盖查询需求”. 当查询的字段全部被索引包含时, 数据库可以直接从索引中获取结果, 而不需要回表查询数据行.

```
create index my_idx on R(A) include B
```

换言之, 这是在描述一个索引my_idx会覆盖对 B 的查询!

- 过滤索引: 在索引的定义中加入 where 语句, 索引中只包括那些满足过滤条件的列值.

```
create index filter_idx1 on R(A) where A is not null
```

应用: 比如大部分男而少部分女, 可只对女做索引. 这些是低频值, 不用每次都会更新索引结构.

- 函数索引:

```
select * from student where UPPER(name) = 'TOM'
create index idx2 on student(UPPER(name))
-- 由于建立了一个针对 UPPER(name) 的索引, 会就是把 UPPER(name) 来查
```

索引的使用说明:

- 一个表上可建多个索引;
- 可以动态地定义索引;
- 随时建立和删除索引;
- 索引可以提高查询效率;
- 耗费空间;
- 降低插入、删除、更新效率.

定义 4.3 (索引选择度)

索引选择度 = $1 / \text{索引列的唯一值个数}^a$.

在这种情况下, 应该选择索引选择度低的建立索引.

^a这里老师的定义有点怪. 根据14 SQL Server Indexing Questions You Were Too Shy To Ask: The ratio of unique values within a key column is referred to as index selectivity. The more unique the values, the higher the selectivity, which means that a unique index has the highest possible selectivity. The query engine loves highly selective key columns, especially if those columns are referenced in the WHERE clause of your frequently run queries. The higher the selectivity, the faster the query engine can reduce the size of the result set. The flipside, of course, is that a column with relatively few unique values is seldom a good candidate to be indexed.

这里的定义是: $\text{NUM_DISTINCT} / \text{NUM_ROWS}$. 在这种情况下, 选择度高的更好.



定义 4.4 (索引过滤性)

索引过滤性 = 查询结果行数/总行数.

现在我们假设每个都分布均匀, 比如基数为 NUM_DISTINCT, 每个里面的记录 = 个数都一样.

- “=” 的索引过滤性为 $1/\text{NUM_DISTINCT}$.
- “≠” 的索引过滤性为 $(\text{NUM_DISTINCT}-1)/\text{NUM_DISTINCT}$.
- “≥” 的索引过滤性为更大的 $/\text{NUM_DISTINCT}$.



例题 4.1 我们假设 SC 分布在 10 个页中, 平均每个学生选修 3 门课, 每门课程有 3 个学生选.

有下面的三个操作:

- Q1: 查询某个学生所修的课程 (with probability p_1);
- Q2: 查询选修某门课程的学生 (with probability p_2);
- I: 插入选课元组 (with probability $1 - p_1 - p_2$).

求问: 当前最好应该使用怎样的索引?

解答.

操作	无索引	sno 索引	cno 索引	全索引
Q1	10	4	10	4
Q2	10	10	4	4
I	2	4	4	6
代价	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$
代价最小时概率分布	$p_1 = p_2 = 0.1$	$p_1 = 0.5, p_2 = 0.1$	$p_1 = 0.1, p_2 = 0.5$	$p_1 = p_2 = 0.4$

表 4.10: 不同索引下的操作代价与最优概率分布

为什么是 4? 因为访问索引一次 + 返回的三个元组 3 次.

4.1.4 视图定义

定义 4.5 (视图)

视图是命名的、从基本表中导出的虚表, 它在物理上并不存在, 存在的只是其定义, 属于外模式.

视图中的数据是从基本表中导出的, 每次对视图查询都要重新计算.

视图之上可以再定义视图.



```
create view view_name[(列名[, 列名] ...)]
as (查询表达式)
[with check option]

drop view view_name
```

```
create view computer_teacher
as
( select tno, tname, salary
  from department, teacher
  where department.tno = teacher.tno
  and dname = '计算机系')

select tname
from  computer_teacher
where salary > 1000
```

视图的优点:

- 使不同用户可以从不同角度观察同一数据;
- 逻辑独立性: 视图作为基本表与外模式之间的映象;
- 安全性: 限制用户数据的访问范围;

定理 4.1 (不可更新的视图: 不含基表主码)

视图定义中不包括基表主码时, 视图不可更新.



证明 设当前的视图定义为 $V(A_1, A_2, \dots, A_m)$, 这其中不含基表 R 的主键 P . 那么当我们视图对视图更新时, 这个更新操作会变成对基表的更新. 唯一的问题在于: 主键 P 自动的被填充为 `null`, 而这种插入操作显然是不合法的, 因而此时视图不可更新.

定理 4.2 (不可更新的视图: 包含聚集函数)

视图定义中包含聚集函数时, 视图不可更新.



证明 原因在于: 对于聚集值的更新无法回逆到基表上.

定理 4.3 (不可更新的视图: 不含连接属性)

视图定义中没有包括连接属性时, 视图不可更新.



```
create table RT (A int , B int);
create table ST (B int , C int);
insert into RT values (1,2),(2,3);
insert into ST values (1,2),(2,3);

create view joinV as
(select A, C
 from RT, ST
 where RT.B = ST.B)
```

现在试图: `insert into joinV values (3, 3).`

RT		ST		joinV	
A	B	B	C	A	C
1	2	1	2	1	3
2	3	2	3	3	3
3	null	null	3		

这里试图在B之上插入null, 是不合法的.

图 4.5: 不可更新的视图: 不含连接属性

不可更新视图还包括:

- select 子句中的目标列包含了聚集函数;
- select 子句中使用 unique 或 distinct 关键字;
- select 子句中包含经算术表达式计算出来的列;
- from 子句中包含了多个表;
- 包含了 group by 子句.

这些都是因为违反了全关系系统准则 6: 视图更新准则. 要存在一个算法可以无二义地把更新要求转换为对基本表的更新序列.

4.1.5 临时表和内存表

	临时表	内存表
存储	表结构和数据都存储在内存中	表结构存储在磁盘中, 表数据存储在内存中
会话	单个会话独享	多个会话共享
断开连接	表结构和表数据都没了	表结构和表数据都存在
服务重启	表结构和表数据都没了	表结构存在, 表数据不存在

表 4.11: 临时表与内存表的比较

4.1.6 公用表表达式 CTE

```
with S-total (sno, value) as
    select sno, sum (grade)
    from SC
    group by sno
S-total-avg(value) as
    select avg (value)
    from S-total
select sno
from S-total, S-total-avg
where S-total.value >= S-total-avg.value
```

```
values row(1,2,3), row(10,9,8)
union all
values row(-1,-2,0),row(10,29,30),row(100,20,-9)
```

4.1.7 分区表

定义 4.6 (分区表)

把逻辑上统一的数据分割成较小的、可以独立管理的物理单元(分片)进行存储.



分区表的优点:

- 增强可用性;
- 维护方便;
- 均衡 I/O;
- 改善查询性能.

一般的分区方式:

- 范围分区: 根据某个属性值的范围进行分区;
- 散列分区: 通过分区编号将数据均匀散列到 I/O 设备上, 使得这些分区大小一致;
- 复合分区: 先使用范围分区; 再在每个分区使用散列分区.

4.2 数据查询

开门见山: 数据查询只需要会下面的语法即可.

```
Select    <目标列>
From     <数据源表>
Where    <行过滤>
Group by <分组>
Having   <分组过滤>
Union    <合并>
Order by <输出排序>
Limit    <输出行数>
```

数据查询必须要经过: 笛卡尔积 \rightarrow 选择 \rightarrow 投影.

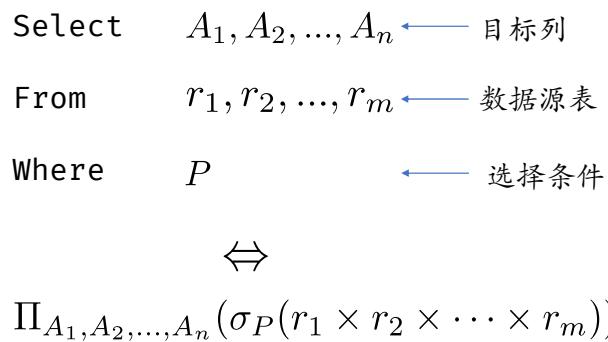


图 4.6: SQL 查询基本结构

例题 4.2 找出选修课程的学生姓名、课程名、成绩.

解答. 最基础的解是:

$$\Pi_{sname, cname, grade}(S \bowtie SC \bowtie C).$$

可以写成上面的三步走的形式, 如下图:

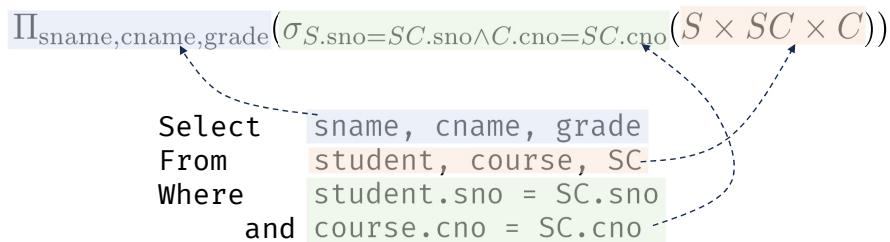


图 4.7: 数据查询三步走

例题 4.3 写出与 $R(A, B) \bowtie S(B, C)$ 等价的 SQL.

解答.

```
Select A, R.B, C
From  R, S
Where R.B = S.B
```

例题 4.4 寻找每个学生在哪个系.

解答.

```
Select    sname, dname
From     student, department
Where    dname = '计算机系'
        and student.dno = department.dno
```

```
-- 给出所有学生的所有信息
select *
from student

-- 给出所有学生的姓名及出生日期
select sname, 2025-age
from student

-- 给出每个老师信息的自然语言描述
select tname + '老师的工资是' + salary
      + ',年龄是' + age
      + ',职称是' + title
from teacher

-- 列出成绩在60~80之间的学生学号
-- 优化小窍门：使用 between 合并两个比较谓词
select sno
from SC
where grade between 60 and 80
```

!!! 注意, SQL 缺省为保留重复值, 也可以用关键字 `all` 显式指明; 若要去掉重复行, 可以用关键字 `distinct` 指明.

$$\Pi_A(R) = \text{select distinct } A \text{ from } R$$

优化小窍门: 只在必要时使用 `distinct`.

输出顺序: `order by` 列名 `[asc | desc]`.

```
select *
from student
order by age asc, sname desc
-- 按照年龄升序输出学生信息, 相同年龄按照姓名降序

select tname, salary*2
from teacher
order by 2
-- 按照 输出列 的编号排序

select sname
from student
order by age
-- 排序列可以不是 输出列
```

更名运算: 可以出现在 `select` 和 `from` 子句中. `old_name (as) new_name`.

```
select sname '姓名',
```

```

sex '性别',
2025 - age '出生日期'
from student
order by 出生日期
-- 或者 order by 3

select S2.sno
from SC as S1, SC as S2
where S1.sno = 's1'
and S1.cno = 'c1'
and S2.cno = 'c1'
and S1.grade < S2.grade
-- 找出比 s1 学生选修 c1 课程成绩高的学生号

```

4.2.1 空值

数据库中允许取 null, 就变成了三值逻辑. true, false, unknown.

- A-Mark null: 表示“未知的”. 值存在, 只是当前没有获得该信息.
- T-Mark null: 表示“不适用”.

全关系系统准则 3: 空值的系统化处理. 全关系型 DBMS 应支持空值概念, 并用系统化的方式处理空值.

注意: 运算中牵扯到了 null, 结果都是 null, 例如: null <> null 结果为 null.

注意: 空值测试. is [not] null.

- 除 is [not] null 之外, 空值不满足任何查找条件.
- 如果 null 参与算术运算, 则该算术表达式的值为 null.
- 如果 null 参与比较运算, 则结果可视为 unknown.

```

select sno
from SC
where grade is null
-- 不要写成 where grade = null

```

MySQL 中的空值处理函数:

- isnull(expr): 如果 expr 值为空, 返回 1, 否则为 0;
- ifnull(check_expr, replace_value): 如果 check_expr 值为空, 返回 replace_value; 否则返回 check_expr.
- nullif(expr1, expr2): 如果两个表达式相等则返回空值, 否则返回第一个表达式.
- coalesce(expr1, expr2, ...): 返回第一个不为 null 的expr.

```

select sno, cno, ifnull (grade, 0)
from SC

select sno, cno, coalesce (grade, 0)
from SC
where coalesce (grade, 0) < 60

```

当指定 order by 的时候, asc 首先输出空值, desc 最后输出空值.

如果要求首先输出空值, 然后由大到小输出非空值怎么办?

```

select id, val,
1 - isnull(val) as is_null

```

```
from order_null
order by is_null, val desc
```

4.2.2 连接运算

连接类型:

- inner join.
- left out join.
- right out join.
- full out join.

Listing 4.1: 不用外连接表达查询的例子

```
-- 列出所有老师的教工号、姓名、工资、所教课程号
select tno, tname, salary, cno
from teacher, TC
where teacher.tno = TC.tno
union select tno, tname, salary, null
from teacher
where tno not in (select tno from TC)
```

Listing 4.2: 用外连接表达查询的例子

```
-- 列出所有老师的教工号、姓名、工资、所教课程号
select tno, tname, salary, cno
from teacher left join TC
on teacher.tno = TC.tno
```

inner join: 只保留符合连接条件的元组.

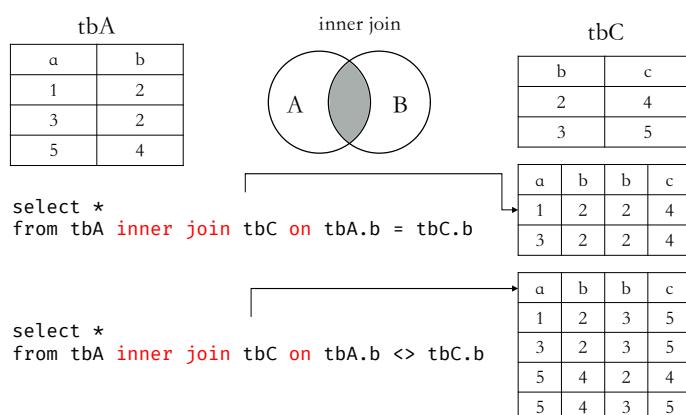


图 4.8: inner join 示例

left join: 对于失配部分, 保留左边信息 +null.

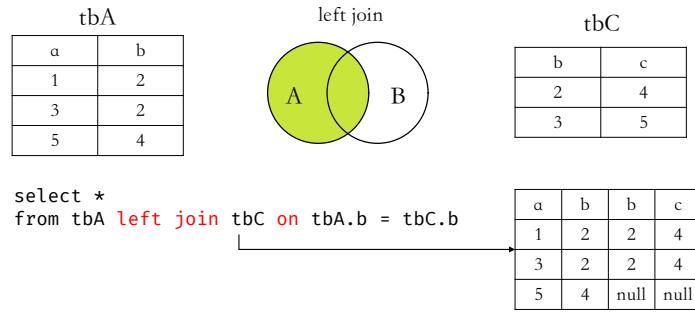


图 4.9: left join 示例

right join: 对于失配部分, 保留右边信息 +(左边)null.

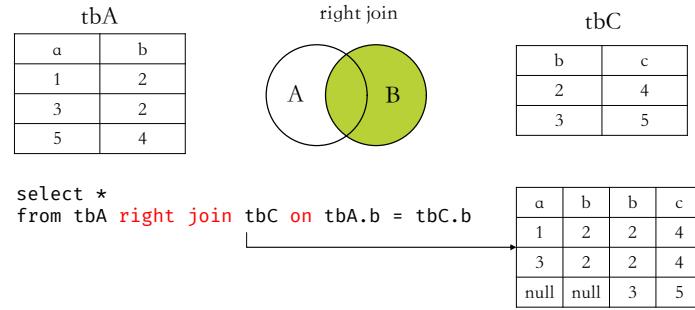


图 4.10: right join 示例

left join excluding inner join: 保留左边表中和右边失配的部分.

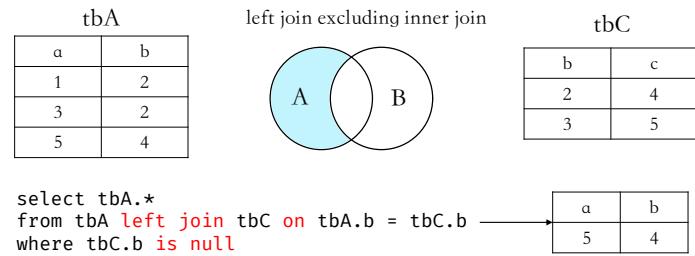


图 4.11: left join excluding inner join

full join excluding inner join: 分别保留两边失配的部分.

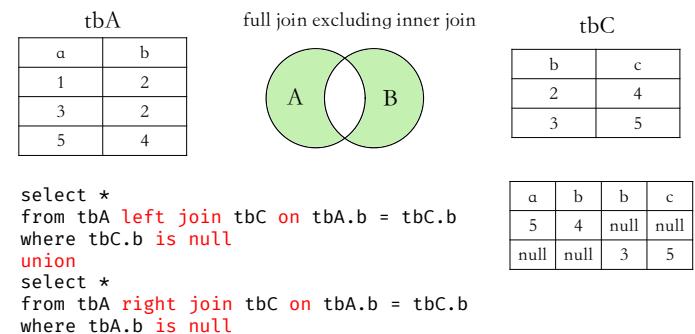


图 4.12: full join excluding inner join

cross join: (R cross join S) as T: 两个关系的笛卡尔积.

tbA

a	b
1	2
3	2
5	4

tbC

b	c
2	4
3	5

select *
from tbA **cross join** tbC

↓

select *
from tbA, tbC

a	b	b	c
1	2	2	4
1	2	3	5
3	2	2	4
3	2	3	5
5	4	2	4
5	4	3	5

图 4.13: cross join

natural join: 匹配相同部分.

tbA

a	b
1	2
3	2
5	4

tbC

b	c
2	4
3	5

select *
from tbA **natural join** tbC

↓

select *
from tbA **join** tbC **using**(b)

b	a	c
2	1	4
2	3	4

图 4.14: natural join

在某些数据库(如 MySQL)中, JOIN 关键字默认等价于 INNER JOIN, 但其他数据库(如 PostgreSQL)可能需要显式写 INNER JOIN.

straight join: 默认情况下, MySQL 优化器会根据统计信息(如表大小、索引使用情况等)决定连接顺序. 但在某些情况下, 优化器的选择可能不是最优的. 此时, STRAIGHT_JOIN 可以手动指定连接顺序, 确保左侧表作为驱动表.

Listing 4.3: 多表连接

```
select *
from tbA A inner join tbA B on A.b=B.b
      inner join tbA C on A.b=C.b

select *
from tbA A join (tbA B, tbA C)
on (A.b=B.b and A.b=C.b)
```

cross apply 与 outer apply: 提取左表中的每一行, 与右表中的所有行进行匹配.

customer(name)	buy(name, product, amt)		
name	name	product	amt
Tom	Tom	toothpaste	5
Jerry	Tom	toothbrush	7
Bob	Tom	soap	3
	Jerry	shampoo	4
	Jerry	soap	6

查询每位顾客购买数量排在前2位的商品
**select C.name, product, amt
from customer C cross apply (**
select top 2 *
from buy B
where C.name = B.name
order by amt desc)

name	product	amt
Tom	toothbrush	7
Tom	toothpaste	5
Jerry	soap	6
Jerry	shampoo	4

图 4.15: cross apply 示例

4.2.3 集合运算

- 集合并: `union (all)`
- 集合交: `intersect (all)`
- 集合差: `except (all)`
- 集合操作缺省去除重复元组
- `intersect` 的优先级高于其他集合操作的优先级
- 加上 `all` 表示当成 multi-set 来操作.

求工资大于 1000 或者年龄大于 60 的教工:

Listing 4.4: 集合操作表达查询的例子

```
-- 查询 1
select tno
from teacher
where salary > 1000
union all
select tno
from teacher
where age > 60

-- 查询 2
select tno
from teacher
where salary > 1000
or age > 60
```

上面的查询 1 和查询 2 等价, 但是查询 2 的效率更高.

Listing 4.5: 求出选修所有课程的同学

```
SELECT sno
FROM student S1
```

```

WHERE
(
    -- 所有课程
    SELECT cno
    FROM C
    EXCEPT
    -- 除去学生选的课程
    SELECT cno
    FROM SC
    WHERE S1.sno = SC.sno
) IS NULL;
-- 剩下的课程为 NULL

```

4.2.4 聚集函数 (Group function)

- 平均值: avg
- 最小值: min
- 最大值: max
- 总和: sum
- 记数: count

Listing 4.6: 聚集函数最容易犯的语法错误

```

select sno
from SC
where grade = max(grade)
-- Error Code: 1111. Invalid use of group function

-- 正确的使用方法
select sno
from SC
where grade = (select max(grade) from SC)

```

聚集函数处理 null: 除了 count 之外忽略 null.

统计型聚集函数: std, stddev, stddev_pop, stddev_samp, variance, var_pop, var_samp.

4.2.5 分组运算

group by 将表中行按指定列上值相等的原则分组, 然后在每一分组上使用聚集函数, 得到单一值.
having 对分组进行选择, 只将聚集函数作用到满足条件的分组上.

```

-- 每门课程 所有同学的平均成绩
select avg(grade)
from SC

-- 特定同学的平均成绩
select avg(grade)
from SC
where sno = 's1'

```

```
-- 列出每个学生的最高、最低、平均成绩
select sno,
       max(grade),
       min(grade),
       avg(grade)
from SC
group by sno
```

分组查询中各子句的顺序: where → group by → having.

-- 所有课程都及格了的同学的平均成绩

```
select sno, avg(grade)
from SC
group by sno
having min(grade) >= 60
```

-- 所有同学的及格了的课程的平均成绩

```
select sno, avg(grade)
from SC
where grade >=60
group by sno
```

-- 列出每一年龄组中男学生（超过50人）的人数

```
select age, count(sno)
from student
where sex = 'M'
group by age
having count(*) > 50
```

Leetcode 1748. 唯一元素的和: 给你一个整数数组 $nums$ 。数组中唯一元素是那些只出现恰好一次的元素。请你返回 $nums$ 中唯一元素的和。

我们把原来的 $nums$ 数组放在表 $numTb$ 的 $nums$ 列之中。

```
select sum(nums)
from
(select nums
 from numTb
group by nums -- 按照 nums 的数值分组
-- 如果组内只有一个，说明是唯一元素
having count(*)=1 ) tmpT
```

group_concat 列名

[order by 排序列]

[separator 分隔符]

```
insert into ds value ('cs', 'bob'),
                   ('cs', 'tom'),
                   ('maths', 'mary'),
                   ('maths', 'lisa'),
                   ('maths', null);
```

```

select dname,
       group_concat(sname)
from ds
group by dname

select dname,
       group_concat(name order by sname)
from ds
group by dname

select dname,
       group_concat(name order by separator ' | ')
from ds
group by dname

```

注意: n 个属性的所有 `group by`, 共有 2^n 个.

Cube: 其实是这样的, 假设我们 `group by` A_1, A_2, \dots, A_m . 同时我们设每个属性上的 `Distinct` 的取值集合为 $D_i (1 \leq i \leq m)$.

那么, `with cube` 会让每个属性在 $\{\text{NULL}\} \cup D_i$ 上变化. 这样总共就是有:

$$\prod_{i=1}^m (|D_i| + 1).$$

sno	cno	grade
小红	C#	90
小红	SQL	85
小李	C#	NULL
小李	SQL	88
小明	C#	82
小明	SQL	93

sno	cno	grade
小红	C#	90
小李	C#	NULL
小明	C#	82
NULL	C#	172
小红	SQL	85
小李	SQL	88
小明	SQL	93
NULL	SQL	266
小红	NULL	175
小李	NULL	88
小明	NULL	175
NULL	NULL	438


```

Select sno, cno, sum(grade)
From SC
Group by sno, cno with cube

```

图 4.16: Cube 示例

```

select Model, Year, Color, sum(Sales)
from car_sales
group by Model, Year, Color with cube

```

Rollup: 按照维度顺序统计. 例如 `group by` A_1, A_2, \dots, A_m , 先不分组统计, 再按照 A_1 分组统计, 再按照 A_1, A_2 分组统计, ..., 最后按照分组 A_1, A_2, \dots, A_m 统计.

同时我们设每个属性上的 `Distinct` 的取值集合为 $D_i (1 \leq i \leq m)$. 那么 Rollup 产生的行数为:

$$|D_1| + |D_1| \times |D_2| + |D_1| \times |D_2| \times |D_3| + \dots + \prod_{i=1}^m |D_i| = \sum_{k=1}^m \prod_{i=1}^k |D_i|.$$

只有 `count(*)` 会把 null 计入, 其他 (包括 `count(B)` 这样的) 都不会计入.

group by 将多个 null 行视为同一个分组.

Grouping: grouping 是一个聚合函数, 它产生一个附加的列, 当用 cube 或 rollup 运算符添加行时, 附加列的输出值为 1, 否则为 0.

```
select if (grouping(A)=1, '汇总', '原始') grpA, A,
       if (grouping(B)=1, '汇总', '原始') grpB, B,
       count(*)
  from group_null
 group by A, B with rollup
```

对于 grouping(A, B, C),

- 分组 (A, B, C) 的标识为 $4*0+2*0+1*0=0$
- 分组 (A, B) 的标识为 $4*0+2*0+1*1=1$
- 分组 (A) 的标识为 $4*0+2*1+1*1=3$
- 分组 () 的标识为 $4*1+2*1+1*1=7$

Grouping sets:

```
group by grouping sets
( (分组属性集1), (分组属性集2), ... (分组属性集n) )

select model, car_year, color, sum(sales)
from car_sales
group by grouping sets (
  (model, theyear),
  (model, color),
  (theyear, color),
  () )
```

$$\begin{aligned} \text{cube}(a,b) &= \text{grouping sets}((a,b),(a),(b),()) \\ \text{rollup}(x,y,z) &= \text{grouping sets}((x,y,z),(x,y),(x),()) \\ \text{group by cube}(a,b), \text{rollup}(x,y,z) &= \text{group by grouping sets}((a,b),(a),(b),()), \text{grouping sets}((x,y,z),(x,y),(x),()) \\ &= \text{group by grouping sets}((a,b,x,y,z),(a,b,x,y),(a,b,x),(a,b), \\ &\quad (a,x,y,z),(a,x,y),(a,x),(a), \\ &\quad (b,x,y,z),(b,x,y),(b,x),(b), \\ &\quad (x,y,z),(x,y),(x),()) \end{aligned}$$

4.2.6 嵌套子查询

in 子查询. [not] in (子查询).

```
SELECT sname
FROM student S, SC
WHERE S.sno = SC.sno
  AND cno = 'c1';

SELECT sname
```

```

FROM student S
WHERE sno IN (
    SELECT sno
    FROM SC
    WHERE cno = 'c1'
);
-- 一般来说，第 1 个查询更有效

```

some/all 子查询:

- 表达式比较运算符 θ some (子查询): 表达式的值至少与子查询结果中的一个值相比满足比较运算符 θ ;
- 表达式比较运算符 θ all (子查询): 表达式的值与子查询结果中的所有的值相比都满足比较运算符 θ ;
- $=some$ 就等价于 in ; $\neq all$ 就等价于 $not in$.

Listing 4.7: 找出平均成绩最高的学生号

```

select sno
from SC
group by sno
having avg(grade) >= all
    (select avg(grade)
     from SC
     group by sno)

```

Listing 4.8: 找出每个系平均成绩最高的学生

```

select dno, S.sno
from student S, SC
where S.sno = SC.sno
group by dno, S.sno
having avg(grade) >= all
    (select avg(grade)
     from student X, SC
     where X.sno = SC.sno
     and X.dno = S.dno
     group by sno)

```

exists 子查询: [not] exists (子查询). 判断子查询的结果集合中是否有任何元组存在.

- in 后的子查询与外层查询无关, 每个子查询执行一次, 称其为无关子查询.
- exists 后的子查询与外层查询有关, 需要执行多次, 称之为相关子查询.

Listing 4.9: 列出选修了 c1 号课程的学生姓名

```

select sname
from student S
where exists
    (select *
     from SC
     where cno = c1
     and sno = S.sno)

```

Listing 4.10: 列出选修了 c1 号和 c2 号课程的学生的学号

```
select sno
from SC SC1
where SC1.cno = c1
and exists
(select sno
from SC
where cno = c2
and sno = SC1.sno)
```

反半连接: not in, not exists.

列出没有选修课程的学生的姓名:

$$\Pi_{sno}(S) - \Pi_{sno}(SC).$$

```
select sname
from student
where sno not in
(select sno
from SC)

select sname
from student S
where not exists
(select sno
from SC
where sno=S.sno)
```

ps. 这里后面介绍了除法和 not exists ... not exists 的关系.

4.2.7 字符串与文本操作

LIKE 操作符:

```
SELECT * FROM table_name
WHERE column_name LIKE pattern;
-- 查找以 "A" 开头的所有名字
SELECT * FROM users WHERE name LIKE 'A%';

-- 查找第二个字符是 "o" 的三字名字
SELECT * FROM users WHERE name LIKE '_o_';

-- 查找包含 "cat" 的名字
SELECT * FROM users WHERE name LIKE '%cat%';
```

REGEXP:

```
SELECT * FROM table_name
WHERE column_name REGEXP pattern;

-- 查找以 "A" 开头且以 "e" 结尾的名字
SELECT * FROM users WHERE name REGEXP '^A.*e$';
```

```
-- 查找包含连续三个数字的名字
SELECT * FROM users WHERE name REGEXP '[0-9]{3}';

-- 查找格式为 "XXX-XXX-XXXX" 的电话号码
SELECT * FROM contacts WHERE phone REGEXP '^[0-9]{3}-[0-9]{3}-[0-9]{4}$';
```

4.3 数据更新

注 期末考试不考!!!

INSERT、DELETE、TRUNCATE、UPDATE、OUTPUT、MERGE.

```
-- 插入单条记录
INSERT INTO employees (name, age, department)
VALUES ('John Doe', 28, 'Engineering');

-- 批量插入
INSERT INTO students (class_id, name, gender, score)
VALUES
(1, '大宝', 'M', 87),
(2, '二宝', 'M', 81);

-- 删除特定记录
DELETE FROM employees WHERE employee_id = 100;

-- 删除所有记录（危险操作！）
DELETE FROM employees;

TRUNCATE TABLE logs; -- 清空 logs 表的所有数据

-- 更新单列
UPDATE employees
SET department = 'Senior Engineering'
WHERE age > 30;

-- 更新多列
UPDATE customers
SET cust_contact = 'Sam Roberts', cust_email = 'sam@toyland.com'
WHERE cust_id = '1000000006';

-- 删除记录并返回被删除的数据
DELETE FROM orders
OUTPUT DELETED.order_id, DELETED.customer_id
WHERE order_date < '2023-01-01';

-- 合并订单数据：匹配则更新，否则插入
MERGE INTO orders AS target
USING (SELECT * FROM temp_orders) AS source
```

```

ON target.order_id = source.order_id
WHEN MATCHED THEN
    UPDATE SET target.amount = source.amount
WHEN NOT MATCHED THEN
    INSERT (order_id, amount) VALUES (source.order_id, source.amount);

```

4.4 服务器端脚本语言

主要掌握游标和触发器的定义和使用!!!

4.4.1 SQL 脚本语法成分

1. 局部变量: 使用 DECLARE 声明变量, 以 @ 开头 (如 @var). 赋值可通过 SET 或 SELECT.

```

DECLARE @Name NVARCHAR(50) = 'Alice';
DECLARE @Counter INT = 0;
SET @Counter = 1;
SELECT @Name = Name FROM Users WHERE UserID = 1;

```

2. 控制流:

(a). 条件判断: IF...ELSE... 和 CASE.

```

IF @Age >= 18
    PRINT 'Adult';
ELSE
    PRINT 'Minor';

SELECT
    Name,
    Status = CASE
        WHEN Salary > 10000 THEN 'High'
        WHEN Salary > 5000 THEN 'Medium'
        ELSE 'Low'
    END
FROM Employees;

```

(b). 循环结构: WHILE 和 LOOP ... LEAVE / ITERATE.

```

DECLARE @i INT = 1;
WHILE @i <= 5
BEGIN
    PRINT 'Iteration ' + CAST(@i AS NVARCHAR);
    SET @i += 1;
END

```

3. 异常处理, 使用 TRY...CATCH.

```

BEGIN TRY
    UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;
    UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;
END TRY

```

```
BEGIN CATCH
    ROLLBACK;
    PRINT 'Error: ' + ERROR_MESSAGE();
END CATCH
```

4. 游标 (Cursor).

```
-- 1. 声明游标
DECLARE employee_cursor CURSOR FOR
SELECT EmployeeID, Name FROM Employees WHERE Department = 'Sales';

-- 2. 打开游标
OPEN employee_cursor;

-- 3. 声明变量存储提取的数据
DECLARE @EmpID INT, @Name NVARCHAR(50);

-- 4. 提取数据并处理
FETCH NEXT FROM employee_cursor INTO @EmpID, @Name;
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Employee ID: ' + CAST(@EmpID AS NVARCHAR) + ', Name: ' + @Name;
    FETCH NEXT FROM employee_cursor INTO @EmpID, @Name;
END

-- 5. 关闭游标
CLOSE employee_cursor;

-- 6. 释放游标
FREE employee_cursor;
```

5. 动态 SQL.

```
DECLARE @SQL NVARCHAR(MAX);
SET @SQL = 'SELECT * FROM Users WHERE Name = ''Alice''';
EXEC(@SQL);

DECLARE @SQL NVARCHAR(MAX), @Name NVARCHAR(50) = 'Alice';
SET @SQL = N'SELECT * FROM Users WHERE Name = @Name';
EXEC sp_executesql @SQL, N'@Name NVARCHAR(50)', @Name;
```

4.4.2 服务器端 SQL 脚本形式

1. 批处理 (Batch). 一组 SQL 语句的集合, 通过 GO 分隔.

```
-- GO 是 SQL Server 的批处理分隔符 (非 SQL 标准)
DECLARE @Msg NVARCHAR(50) = 'Hello';
PRINT @Msg;
GO
PRINT @Msg; -- 错误: @Msg 作用域仅在第一个批处理中
```

2. 存储过程 (Stored Procedure): 预编译的 SQL 代码块, 可接受输入/输出参数.

```

CREATE PROCEDURE usp_GetEmployee
    @ID INT
AS
BEGIN
    SELECT * FROM Employees WHERE EmployeeID = @ID;
END

EXEC usp_GetEmployee @ID = 1;

```

3. 函数 (Function): 标量函数 (返回单个值), 表值函数 (返回表结果集).

```

CREATE FUNCTION dbo.GetAge(@DOB DATE)
RETURNS INT
AS
BEGIN
    RETURN DATEDIFF(YEAR, @DOB, GETDATE());
END

SELECT dbo.GetAge('2000-01-01');

CREATE FUNCTION dbo.GetEmployeesByDept(@Dept NVARCHAR(50))
RETURNS TABLE
AS
RETURN SELECT * FROM Employees WHERE Department = @Dept;

SELECT * FROM dbo.GetEmployeesByDept('Sales');

```

4. 触发器 (Trigger): 在表的 INSERT, UPDATE, DELETE 操作前后自动执行的代码.

```

CREATE TRIGGER trg_AuditLog
ON Employees
AFTER UPDATE
AS
BEGIN
    INSERT INTO AuditLog (Action, Timestamp)
    VALUES ('Update', GETDATE());
END

```

第五章 完整性与安全性

期末考试提纲

完整性定义

约束级别、约束类型、约束检查、延迟约束

安全性定义

角色、权限、统计数据库安全

5.1 数据完整性

约束的对象级别:

- 列级约束, 列值范围. e.g., sno 要求是 8 位整数, 首位是 0 或 1; 选课人数不能少于 10 人, 多于 100 人.
- 行级约束, 同一行各列之间. e.g., 飞行员的星级评定取决于其飞行里程.
- 表级约束, 行间、表上、表间. e.g., 在本地纳税记录超过 5 年才有购房资格.

约束类型:

- primary key;
- foreign key;
- unique;
- default;
- not null;
- check.

存储约束的系统表: sysconstraints.

列名	数据类型	描述
constid	int	约束号
id	int	拥有该约束的表 ID
colid	smallint	在其上定义约束的列 ID, 如果是表约束则为 0
status	int	位图指示状态. 可能的值包括: 1 = PRIMARY KEY 约束 2 = UNIQUE KEY 约束 3 = FOREIGN KEY 约束 4 = CHECK 约束 5 = DEFAULT 约束 16 = 列级约束 32 = 表级约束

表 5.1: 存储约束的系统表: sysconstraints

主键约束 (Primary key) 是列级或表级的.

```
-- 列级约束
CREATE TABLE Students (
    Sno CHAR(8) PRIMARY KEY,
    Sname CHAR(10)
);
```

```
-- 表级约束
```

```
CREATE TABLE Orders (
    OrderID INT,
    ProductID INT,
    Quantity INT,
    PRIMARY KEY (OrderID, ProductID) -- 多列主键需表级声明
);
```

外键约束 (Foreign key) 既可用于列约束, 也可用于表约束.

```
-- 列约束
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT FOREIGN KEY REFERENCES Customers(CustomerID)
);

-- 表约束
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

唯一性约束 (Unique) 列级和表级.

```
CREATE TABLE Users (
    Email VARCHAR(50) UNIQUE
);

CREATE TABLE Logins (
    Username VARCHAR(20),
    Platform VARCHAR(10),
    UNIQUE (Username, Platform) -- 多列组合唯一性需表级声明
);
```

默认值约束 (Default): 列级.

```
CREATE TABLE Employees (
    Name VARCHAR(50),
    Status VARCHAR(10) DEFAULT 'Active'
);
```

非空约束 (Not Null): 列级.

```
CREATE TABLE Students (
    Name VARCHAR(50) NOT NULL
);
```

检查约束 (Check): 列级或者表级.

```
CREATE TABLE Products (
    Price DECIMAL CHECK (Price > 0)
);
```

```
CREATE TABLE Orders (
    Quantity INT,
    Discount DECIMAL,
    CHECK (Quantity * Discount <= 100) -- 跨列检查需表级声明
);
```

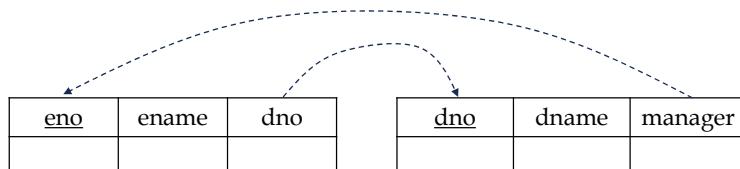
总结:

- 列级约束有六种: 主键 Primary key、外键 foreign key、唯一 unique、检查 check、默认 default、非空/空值 not null/ null
- 表级约束有四种: 主键、外键、唯一、检查

Listing 5.1: 约束命名及其定义

```
constraint 约束名 <约束条件>
sno char(8) constraint Sno_PK primary key

alter table student drop constraint Sno_PK
alter table student add constraint Sex_CHECK check(sex in ('M', 'F'))
```



- ① create table emp(eno primary key ...)
- ② create table dept(dno primary key, manager foreign key ...)
- ③ alter table emp add constraint dno foreign key ...

图 5.1: 定义两个相互参照的表

外键有三种定义方式,他们的删除时的行为也不一样:

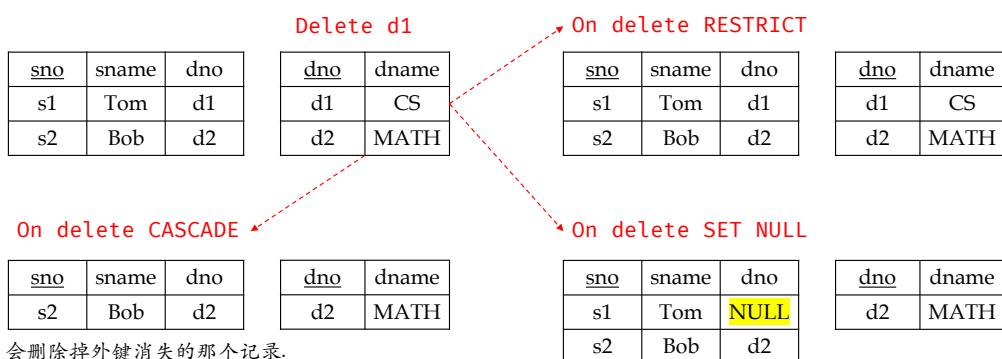


图 5.2: 外键的三种定义

```
create table SC (
    sno char(8),
    cno char(10),
    grade smallint,
    primary key (sno, cno),
    check(sno in (select sno from student)),
```

```

check(cno in (select cno from course)))
-- 这里的 check 定义丝毫不等于外码约束
-- 因为 check 定义只有在 SC 表更新才会触发
-- student 中的删除不会触发 check

```

相互参照的表和自参照的表插入数据:

- 将多个更新操作语句放入一个事务, 在提交时才检查约束.
- 使用延迟约束.

使用延迟约束:

```

CREATE TABLE Orders (
    OrderID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    CustomerID NUMBER,
    OrderDate DATE,
    CONSTRAINT fk_customer
        FOREIGN KEY (CustomerID)
        REFERENCES Customers(CustomerID)
        DEFERRABLE INITIALLY DEFERRED
);

BEGIN
    INSERT INTO Orders (CustomerID, OrderDate) VALUES (999, TO_DATE('2023-01-01', 'YYYY-MM-DD'));
    -- 其他操作...
    COMMIT; -- 提交时检查约束
END;

SET CONSTRAINT fk_customer DEFERRED;

```

SQL Server 可以添加一个未验证的约束:

```

alter table t1 with nocheck add constraint skip_check check (col_a > 1)

alter table t1 nocheck constraint skip_check
insert into t1 values (-2)
alter table t1 check constraint skip_check

```

5.2 数据库安全性

定义 5.1 (主体 (Principal))

主体 (principal) 是可以授予权限以访问特定数据库对象的对象, 包括登录用户、角色、应用程序.



MySQL 中创建一个主体:

```

CREATE USER 'username'@'host' IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON database_name.* TO 'username'@'host';

```

定义 5.2 (客体 (Securable))

客体 (Securable) 是指数据库系统中任何可以设置权限的对象。这些对象包括但不限于数据库本身、表、视图、存储过程、函数、列、序列、模式等。

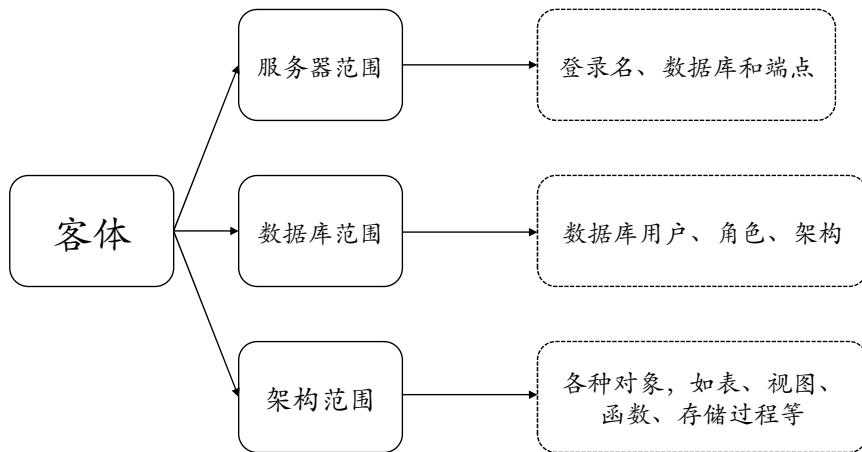


图 5.3: 客体分类

定义 5.3 (权限 (Permission))

权限 (Permission) 允许主体在安全对象上执行操作。

**定义 5.4 (权限的转授和回收)**

权限的转授和回收: 允许用户把已获得的权限转授给其他用户, 或者把已授给其他用户的权限再收回上来。

**定义 5.5 (权限图)**

结点是用户, 根结点是 DBA.

- 有向边 $U_i \rightarrow U_j$, 表示用户 U_i 把某权限授给用户 U_j ;
- 一个用户拥有权限的充分必要条件是在权限图中有一条从根结点到该用户结点的路径。



在权限图上收回权限时, 要注意始终保证授权路径起点是 DBA。

授权命令:

```
grant 权限
on 对象名
to {用户 ... | public}
[with grant option]
```

表级权限: select, update, insert, delete, index, alter, drop, resource 等以及它们的总和 all

收回权限:

```
revoke 权限
on 对象
from {用户 [, 用户] ... | public}
```

收回权限时, 若该用户已将权限转授给其它用户, 则也一并收回。

```
grant select, insert on S to Liming with grant option
revoke insert on S from Liming
```

定义 5.6 (角色)

角色是一组相关权限的结合, 即将多个不同的权限集合在一起就形成了角色.



创建数据库角色: `create role role_name`.

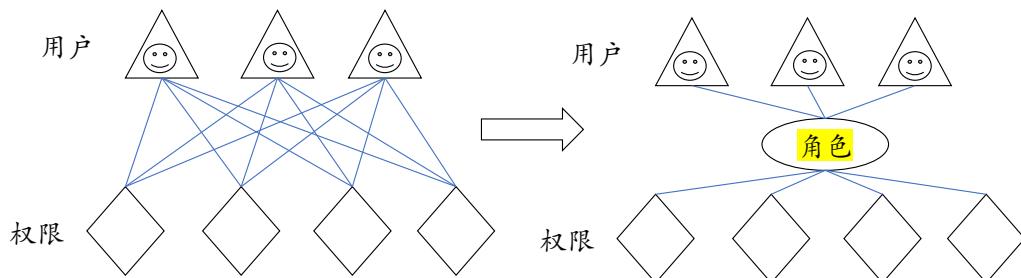


图 5.4: 数据库角色

授权 Tom 只有察看职工平均工资的权限:

```
create view avg_sal
as
(select avg(sal)
from teacher)

grant SELECT on avg_sal to 'Tom'
```

Listing 5.2: 普通员工只能查看自己的记录

```
declare @usr char(30)
set @usr = user
select 'The current user is: '+@usr

select *
from student
where sname = user
```

访问控制类型:

- 自主访问控制 (DAC): 对客体拥有控制权的主体能够将对该客体的访问权自主地授予其它主体, 并在随后任何时刻将这些权限收回;
- 强制访问控制 (MAC):
 - 敏感度标记: 绝密、机密、可信、公开.
 - 主体: 许可证级别; 客体: 密级.
 - 保密性规则:
 - 下读: 仅当主体许可证级别高于或等于客体密级时才能读取相应客体
 - 上写: 仅当主体许可证级别低于或等于客体密级时才能写相应客体

定义 5.7 (审计)

审计就是对指定用户在数据库中的操作情况进行监控和记录, 用以审查用户的相关活动.

审计就是监视和收集指定数据库的活动数据.



```
create server audit MyServerAudit to file ...
```

```

create server audit specification MyServerAuditSpe
    for server audit MyServerAudit
alter server audit specification MyServerAuditSpe
    add (SERVER_PRINCIPAL_CHANGE_GROUP)
create database audit specification MyDBAudit
    for server audit MyServerAudit
alter database audit specification MyDBAudit
    add ( SELECT ON student )

select event_time, succeeded, statement
from sys.fn_get_audit_file(...)

```

加密:

- 短语加密: encryptByPassPhrase、decryptByPassPhrase.
- 非对称密钥加密:

```

create asymmetric key myAsym_key
insert into emp(ename, salary) values ('tom',
    EncryptByAsymkey(Asymkey_ID('myAsym_key'), 100000000))
select DecryptByAsymkey( Asymkey_ID('myAsym_key'), salary)
from emp
where name = 'tom'

```

- 对称密钥加密.

SQL 注入:

- 认证过程发出的查询语句:

```

select * from users
where username = 'jake'
and PASSWORD = 'jakespasswd'

```

- 攻击者篡改这个语句:

```

select * from users
where username = 'jake'
and PASSWORD = 'jakespasswd' or 'x' = 'x'

```

统计数据库安全性:

在某些高敏感性场景下, 为了保护隐私, 系统会限制用户只能进行聚集查询 (Aggregate Queries). 攻击者仍可能通过多次合法的聚集查询推理出某个个体的具体值, 这就叫作“统计推断攻击 (Statistical Inference Attack)”

- 漏洞一: 个体太少. 如果只有一两个学生选了这门课, 那平均值就几乎等于个体值, 容易被推测出来.
- 漏洞二: 多次查询, 太多交叠. 通过多个查询结果相减来推出个体值.

我们加入防范措施:

- 查询引用的数据不能少于 n 条元组;
 - 任意两个查询的交集不能超过 m 条元组;
 - 那么在上面两条的保证下, 推出个体信息至少需要 $1 + (n - 2)/m$ 次查询.
- 连接推理: 可以通过两张表里相同列的连接, 推理出个人信息.

k -anonymity:

定义 5.8 (准标识符 (Quasi-Identifier, QI))

准标识符: 指一组属性 (如年龄、性别、邮编), 单独使用时无法直接标识个体, 但结合外部数据 (如选民名单) 可能定位到具体个人.

**定义 5.9 (泛化)**

将具体值替换为更宽泛的范畴 (如年龄 “25 岁” ⊙ “20-30 岁”).

**定义 5.10 (抑制)**

直接删除高风险数据 (如罕见疾病的患者).
使所有记录的准标识符组合形成至少 k 个相同分组.

**定义 5.11 (k -anonymity)**

若数据满足 k -anonymity, 则对任意记录 r , 存在至少 $k - 1$ 条记录满足:

$$\forall r \in D, |\{r' \in D | \text{QI}(r') = \text{QI}(r)\}| \geq k.$$



第六章 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.

6.1 递归查询

6.1.1 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。

6.1.2 递归查询

层次结构的传递闭包: 使用递归查询.

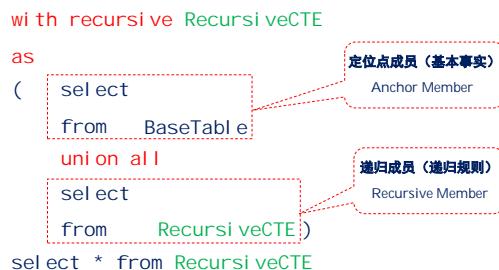


图 6.1: MySQL 中的递归查询

```
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, Components C
  where A.subpart = C.part )
```

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

例题 6.1 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
```

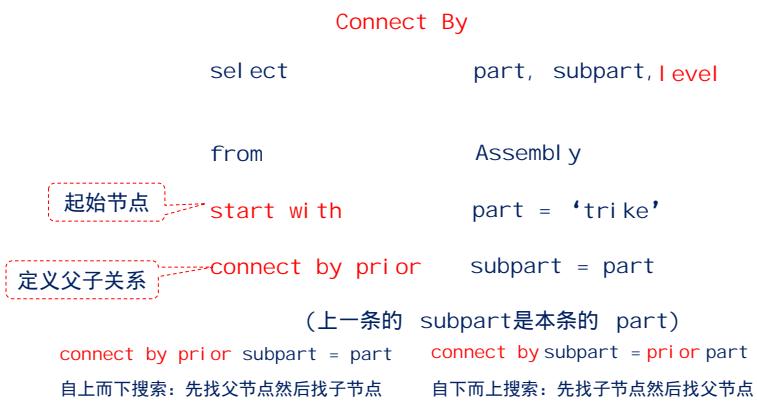


图 6.2: Oracle 中的递归查询

6.1.3 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

6.2 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 6.1: 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 6.2: 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 6.3: 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
```

6.3 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;
```

6.4 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: [\[3\]](#).

高维数据的搜索: HNSW: Hierarchical Navigable Small World [\[4\]](#).

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;
```

第七章 关系规范化

期末考试提纲

- | | |
|--|---|
| <ul style="list-style-type: none"><input type="checkbox"/> 数据异常包括哪些?<input type="checkbox"/> 函数依赖、部分函数依赖、完全函数依赖、传递函数依赖、多值依赖<input type="checkbox"/> 理解 1NF、2NF、3NF、BCNF、4NF 的概念<input type="checkbox"/> 逻辑蕴含、Armstrong 公理<input type="checkbox"/> 属性集闭包及其计算, 候选码计算, 范式判定<input type="checkbox"/> 函数依赖集等价性判定算法 | <ul style="list-style-type: none"><input type="checkbox"/> 最小覆盖计算过程<input type="checkbox"/> 关系模式分解的定义和目标, 函数依赖集向属性集的投影<input type="checkbox"/> 什么是无损连接分解<input type="checkbox"/> 保持函数依赖分解以及判定方法<input type="checkbox"/> 保持函数依赖的 3NF 分解<input type="checkbox"/> 保持无损连接的 BCNF 分解 |
|--|---|

7.1 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 7.1: 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: $\boxed{\text{职工}} \rightarrow \boxed{\text{级别}} + \boxed{\text{级别}} \rightarrow \boxed{\text{工资}}$ 。

7.2 函数依赖

定义 7.1 (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 7.2 (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 7.3 (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 7.4 (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 7.5 (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 7.5 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.

7.3 码的定义（使用函数依赖）

定义 7.6 (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 7.1 $R(ABC; \{A \rightarrow B, B \rightarrow C\})$ 有多少超码?

超码: $\{A, AB, AC, ABC\}$.

定义 7.7 (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 7.2 $R(ABC; \{A \rightarrow B, B \rightarrow C\})$ 有多少候选码? $\{A\}$.

定义 7.8 (主属性)

包含在任意候选码中的属性, 称作主属性.



例题 7.3 $R(ABC; \{AB \rightarrow C, C \rightarrow AB\})$, 计算 R 的主属性. 候选码: $\{AB, C\}$, 主属性: $\{A, B, C\}$.

定义 7.9 (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.

7.4 范式

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

函数依赖为:

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

$$sno \rightarrow sname$$

$$sno \rightarrow dno$$

$$dno \rightarrow dean$$

数据在表7.2中.

<u>sno</u>	sname	dno	dean	<u>cno</u>	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 7.2: 学生数据表

定义 7.10 (范式)

范式是对关系的不同数据依赖程度的要求.



定义 7.11 (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.

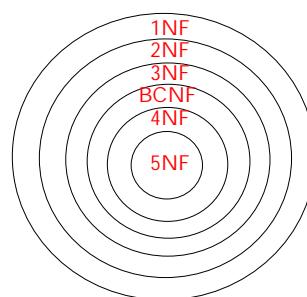


图 7.1: 范式

7.4.1 1NF

定义 7.12 (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.



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7.2.

- 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。

7.4.2 2NF

定义 7.13 (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 7.2 存在非主属性对码的部分依赖, 不是 2NF.

例题 7.4 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.

7.4.3 3NF

定义 7.14 (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 7.5 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? 破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 7.6 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.

7.4.4 BCNF

定义 7.15 (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 7.2: 把关系模式改造为 BCNF

例题 7.7 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 7.16 (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.

7.4.5 多值依赖

定义 7.17 (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 7.3: 多值依赖的例子

定义 7.18 (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	c1	t1	b1		t_3	c1	t1	b2
t_2	c1	t2	b2	⇒	t_4	c1	t2	b1

图 7.4: 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 7.19 (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 7.1 (多值依赖成立)

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



7.4.6 4NF

定义 7.20 (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.

7.4.7 PJNF

定义 7.21 (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 7.22 (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



7.5 Armstrong 公理系统

定义 7.23 (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 7.24 (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 7.2 (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 7.3 (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 $X \rightarrow Y, X \rightarrow Z$.¹
3. 伪传递律 (pseudotransitivity rule): 若 $X \rightarrow Y, WY \rightarrow Z$, 则 $WX \rightarrow Z$.

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

7.6 闭包计算

定义 7.25 (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 7.1: 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7.1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 7.4 (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 7.26 (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 7.5

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



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 7.6 (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.

7.7 候选码计算

定义 7.27 (左部属性)

左部属性, 只出现在 F 左边的属性.



定义 7.28 (右部属性)

右部属性, 只出现在 F 右边的属性.



定义 7.29 (双部属性)

双部属性, 出现在 F 两边的属性.



定义 7.30 (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 7.8 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 7.2: 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 7.3: 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( $W$ );

```

7.8 函数依赖的等价和覆盖

定义 7.31 (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 7.31 实际上要求我们检验: $F \subseteq G^+ \wedge G \subseteq F^+$.

定义 7.32 (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 7.9 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 $F_{min} = \{A \rightarrow B, B \rightarrow A, A \rightarrow C\}$.

7.9 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 $X \rightarrow Z$.

7.10 模式分解

定义 7.33 (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 7.10 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 7.11 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 7.34 (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.



7.10.1 保持函数依赖分解

定义 7.35 (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 ρ 是保持函数依赖的分解.



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

7.10.2 保持无损连接分解

定义 7.36 (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 ρ 是 R 的一个无损分解.

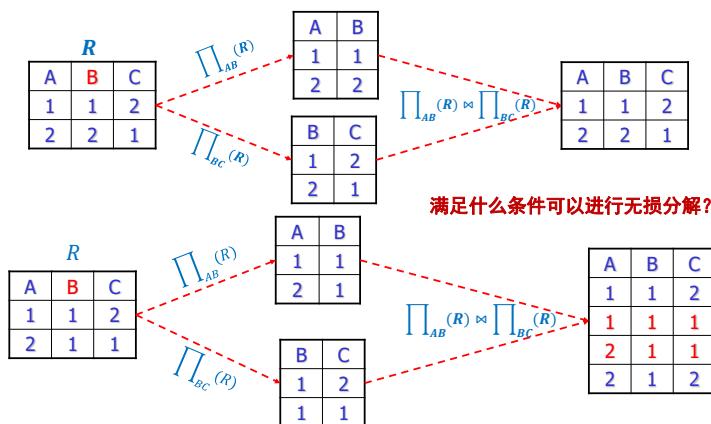


图 7.5: 无损分解的例子

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 ρ 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 ρ 为有损分解.

例题 7.12 判断下面的分解是否是无损分解.

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

	A	B	C	D	E
ABC	a ₁	a ₂	a ₃	b ₁₄	b ₁₅
CD	b ₂₁	b ₂₂	a ₃	a ₄	b ₂₅
DE	b ₃₁	b ₃₂	b ₃₃	a ₄	a ₅

	A	B	C	D	E
ABC	a ₁	a ₂	a ₃	a ₃	b ₁₅
CD	b ₂₁	b ₂₂	a ₃	a ₄	b ₂₅
DE	b ₃₁	b ₃₂	b ₃₃	a ₄	a ₅

	A	B	C	D	E
ABC	a ₁	a ₂	a ₃	a ₄	b ₂₅
CD	b ₂₁	b ₂₂	a ₃	a ₄	b ₂₅
DE	b ₃₁	b ₃₂	b ₃₃	a ₄	a ₅

	A	B	C	D	E
ABC	a ₁	a ₂	a ₃	a ₃	a ₅
CD	b ₂₁	b ₂₂	a ₃	a ₄	a ₅
DE	b ₃₁	b ₃₂	b ₃₃	a ₄	a ₅

图 7.6: 无损连接分解判别算法示例

例题 7.13 判断下面的分解是否是无损分解.

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

	A	B	C	D	E
AD	a ₁	b ₁₂	b ₁₃	a ₄	b ₁₅
AB	a ₁	a ₂	b ₂₃	b ₂₄	b ₂₅
BE	b ₃₁	a ₂	b ₃₃	b ₃₄	a ₅
CDE	b ₄₁	b ₄₂	a ₃	a ₄	a ₅
AE	a ₁	b ₃₂	b ₅₃	b ₅₄	a ₅

	A	B	C	D	E
AD	a ₁	b ₁₂	b ₁₃	a ₄	b ₁₅
AB	a ₁	a ₂	b ₁₃	b ₂₄	b ₂₅
BE	b ₃₁	a ₂	b ₃₃	b ₃₄	a ₅
CDE	b ₄₁	b ₄₂	a ₃	a ₄	a ₅
AE	a ₁	b ₃₂	b ₁₃	b ₅₄	a ₅

	A	B	C	D	E
AD	a ₁	b ₁₂	b ₁₃	a ₄	b ₁₅
AB	a ₁	a ₂	b ₁₃	b ₂₄	b ₂₅
BE	b ₃₁	a ₂	b ₁₃	b ₃₄	a ₅
CDE	b ₄₁	b ₄₂	a ₃	a ₄	a ₅
AE	a ₁	b ₃₂	b ₁₃	b ₅₄	a ₅

	A	B	C	D	E
AD	a ₁	b ₁₂	b ₁₃	a ₄	b ₁₅
AB	a ₁	a ₂	b ₁₃	a ₄	b ₂₅
BE	b ₃₁	a ₂	b ₁₃	a ₄	a ₅
CDE	b ₄₁	b ₄₂	a ₃	a ₄	a ₅
AE	a ₁	b ₃₂	a ₃	a ₄	a ₅

	A	B	C	D	E
AD	a ₁	b ₁₂	b ₁₃	a ₄	b ₁₅
AB	a ₁	a ₂	a ₃	a ₄	b ₂₅
BE	b ₃₁	a ₂	a ₃	a ₄	a ₅
CDE	b ₄₁	b ₄₂	a ₃	a ₄	a ₅
AE	a ₁	b ₃₂	a ₃	a ₄	a ₅

图 7.7: 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:

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

7.10.3 关系模式分解算法

7.10.3.1 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 ρ 中各关系模式是否属于 BCNF, 若是, 则算法终止;

3. 设 ρ 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 σ 代替 R_i , 返回到 2.

定理 7.7

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 7.8

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 7.14 $U = \{A, B, C, D, E\}, F = \{A \rightarrow B, B \rightarrow C, AD \rightarrow E\}$.

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

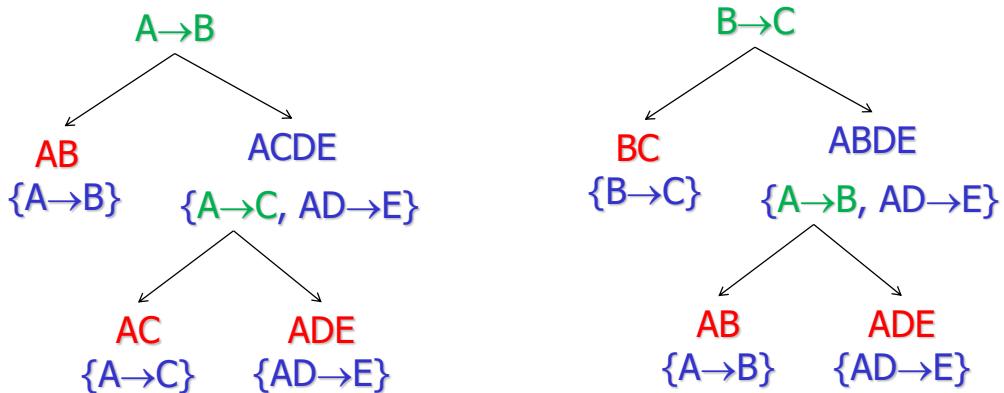


图 7.8: 达到 BCNF 无损连接分解算法

例题 7.15 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 7.16 $R(A_1 A_2 \dots A_n; \{A_1 \rightarrow A_2, A_2 \rightarrow A_3, \dots, A_{n-1} \rightarrow A_n\})$ 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.

7.10.3.2 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 ρ 中各关系模式是否属于 4NF, 若是, 则算法终止;
3. 设 ρ 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 σ 代替 R_i , 返回到 2.

例题 7.17 $U = \{A, B, C, D, E, G\}$, $F = \{A \rightarrow BCG, B \rightarrow AC, C \rightarrow G\}$, 码为 BDE .

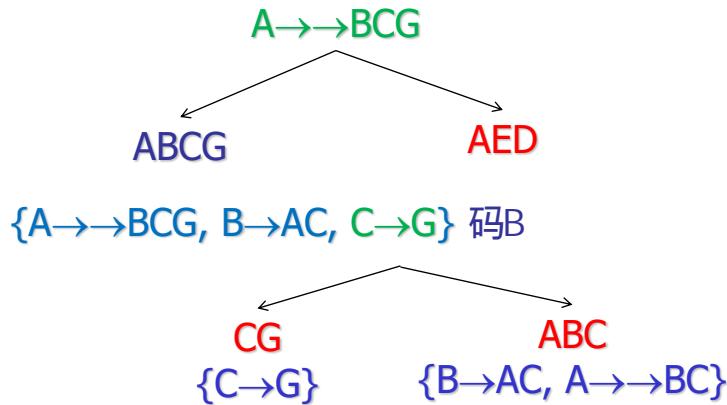


图 7.9: 达到 4NF 无损连接分解算法

7.10.3.3 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.

7.10.3.4 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 X 是 $R(U, F)$ 的码.

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

定义 7.37 (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.

A	B		B	C		A	B	C
a1	b1		b1	c1		a1	b1	c1
a2	b2		b2	c2		a2	b2	c2
a3	b3							

图 7.10: 悬挂元组

7.11 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.

7.11.1 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.

第八章 事务

期末考试提纲

- 事务定义及事务的 ACID 特性
- 可恢复调度、无级联调度
- 调度中的四种数据不一致性
- SQL 中四个事务隔离性级别定义
- 理解快照隔离的概念
- 冲突可串行化及其判定
- 视图可串行化及其判定

定义 8.1 (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.



8.1 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.`

8.1.1 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.

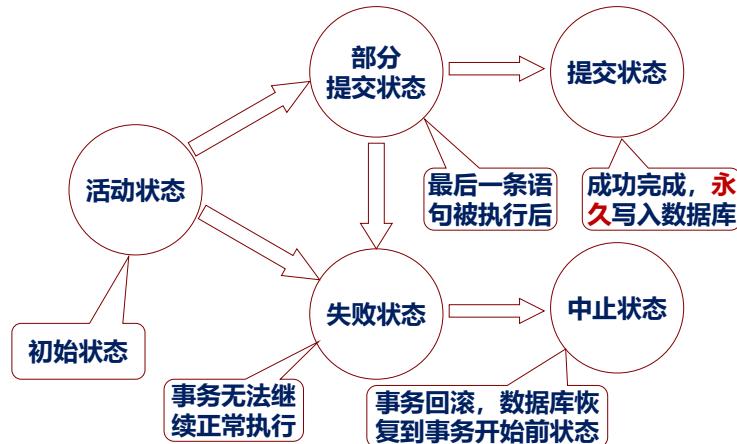


图 8.1: 事务生命周期图

8.2 事务调度

定义 8.2 (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 8.3 (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 8.4 (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 8.1 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 8.5 (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 8.6 (级联调度)**

由于一个事务故障而导致一系列事务回滚.

**定义 8.7 (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.

无级连调度必是可恢复调度.



8.2.1 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.

8.3 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 8.1: 事务隔离性级别

8.4 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.

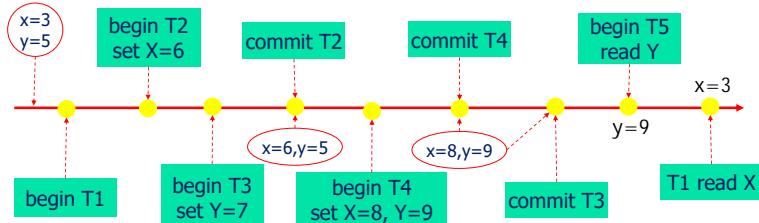


图 8.2: 快照隔离

快照隔离中有不一致的现象.

例题 8.2 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.

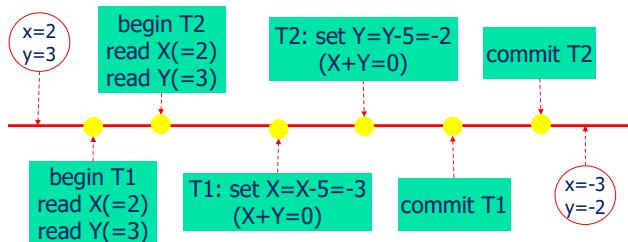


图 8.3: 快照隔离中的不一致现象

SQL Server 中的快照隔离:

- 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.
- 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.

8.5 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?

8.5.1 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 .

- $I_i = \text{read}(Q), I_j = \text{read}(Q)$.
- $I_i = \text{read}(Q), I_j = \text{write}(Q)$.
- $I_i = \text{write}(Q), I_j = \text{read}(Q)$.
- $I_i = \text{write}(Q), I_j = \text{write}(Q)$.

只有上面的情况 1 可交换.

定义 8.8 (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 8.9 (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 8.10 (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 8.11 (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 8.1**

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 8.1 (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 8.2

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 8.1, 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.

8.5.2 视图可串行化

定义 8.12 (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.

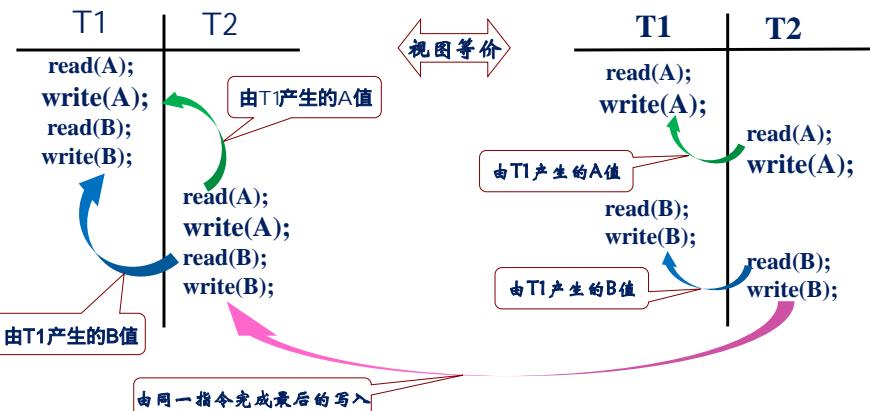


图 8.4: 视图等价

定义 8.13 (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 8.1

冲突可串行化调度一定是视图可串行化的.



推论 8.2

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 8.3 (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); A := A - 50 write(A); read(B); B := B - 10 write(B); read(B); B := B + 50 write(B);	read(B); B := B + 50 write(B); read(A); A := A + 10 write(A);	read(A); A := A - 50 write(A); read(B); B := B + 50 write(B);	read(B); B := B - 10 write(B); read(A); A := A + 10 write(A);

图 8.5: 可串行化但非视图可串行化的调度

8.6 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;

```

第八章 练习

1. 考虑关系 Employee(ID, salary), 有两个元组 (A, 20) 和 (B, 30). 下面有两个事务 T_1 和 T_2 :

```
-- 事务 T1
begin transaction;
insert into Employee values ('C',30);
set salary = salary+10 where ID='A';
commit;

-- 事务 T2
begin transaction;
select sum(salary) as sal1 from Employee;
select sum(salary) as sal2 from Employee;
commit;
```

给出 T_1, T_2 在不同隔离性级别下所返回的 sal1 与 sal2 所有可能的值的情况, 填入表格中 (只需考虑 T_1 是 Serializable 和 Repeatable read, 另外注意 MySQL 有当前读和快照读).

	T_1 : Repeatable read	T_1 : Serializable
T_2	(sal1, sal2)	(sal1, sal2)
Read uncommitted		
Read committed		
Repeatable read		
Serializable		

2. 调度类型判定.

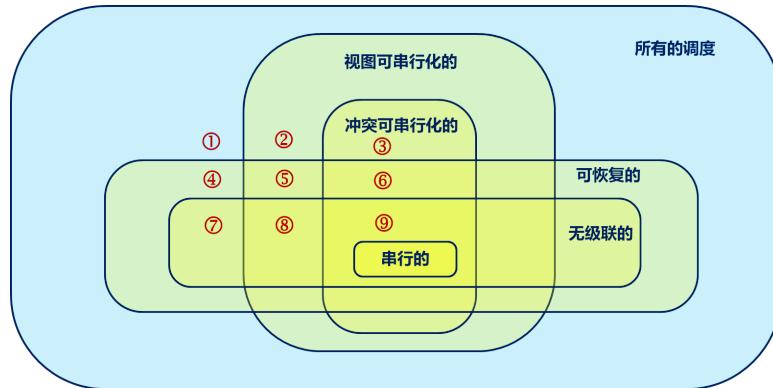


图 8.6: 调度类型

请在 $S_1 \sim S_3, S_4 \sim S_6, S_7 \sim S_9$ 和 $S_{10} \sim S_{12}$ 之间各选择 1 个, 为其标注上页图中的标号, 并说明理由. 比如选择标号 7, 要说明这个调度为什么是无级联的并且不是视图可串行化的.

$$S_1 : r_1(X), r_2(X), w_1(X), w_2(X)$$

$$S_2 : w_1(X), r_2(X), r_1(Y), r_2(X)$$

$$S_3 : r_1(X), r_2(Y), w_3(X), r_2(X), r_1(X)$$

-
- $S_4 : r_1(X), r_1(Y), w_1(X), r_2(Y), w_3(Y), w_1(X), r_2(Y)$
- $S_5 : r_1(X), w_2(X), w_1(X), \text{abort}(T_2), \text{commit}(T_1)$
- $S_6 : r_1(X), w_2(X), w_1(X), \text{commit}(T_2), \text{commit}(T_1)$
- $S_7 : w_1(X), r_2(X), w_1(X), \text{abort}(T_2), \text{commit}(T_1)$
- $S_8 : w_1(X), r_2(X), w_1(X), \text{commit}(T_2), \text{commit}(T_1)$
- $S_9 : w_1(X), r_2(X), w_1(X), \text{commit}(T_2), \text{abort}(T_1)$
- $S_{10} : r_2(X), w_3(X), \text{commit}(T_3), w_1(Y), \text{commit}(T_1), r_2(Y), w_2(Z), \text{commit}(T_2)$
- $S_{11} : r_1(X), w_2(X), \text{commit}(T_2), w_1(X), \text{commit}(T_1), r_3(X), \text{commit}(T_3)$
- $S_{12} : r_1(X), w_2(X), w_1(X), r_3(Y), \text{commit}(T_1), \text{commit}(T_2), \text{commit}(T_3)$

第九章 并发控制

期末考试提纲

- X 锁、S 锁、U 锁、IS 锁、IX 锁、SIX 锁
- 两段锁协议内容及其作用
- 死锁及其解决措施
- 基于时间戳的并发控制协议
- 基于有效性检查的并发控制协议
- MySQL MVCC 中的读视图和可见性算法

9.1 基于锁的协议

定义 9.1 (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.

9.1.1 封锁类型

- 基本锁类型: X 锁、S 锁、U 锁
- 意向锁: IS、IX、IU、SIX
- 码范围锁: RangeS_S、RangeI_N
- 其他锁: 模式锁、闩锁、BU 锁

定义 9.2 (排它锁 (X 锁, eXclusive lock))

lock-X(R): 又称写锁，持有 X 锁可以读写数据项。

事务 T 对数据对象 R 加上 X 锁，则其它事务对 R 的任何封锁请求都不能成功，直至 T 释放 R 上的 X 锁。



定义 9.3 (共享锁 (S 锁, Share lock))

lock-S(R): 又称读锁，持有 S 锁只能读取数据项。

事务 T 对数据对象 R 加上 X 锁，则其它事务对 R 的任何 X 锁请求都不能成功，而对 R 的 S 锁请求可以成功。



封锁的相容矩阵 comp(A,B):

请求锁模式 A	现有锁模式 B	
	S	X
S	是	否
X	否	否

表 9.1: 封锁的相容矩阵

长锁: 保持到事务结束时才释放的锁

短锁: 在事务中途就可以释放的锁

- read uncommitted: 不申请锁。

- read committed: 短 S 锁.
- repeatable read: 长 S 锁.

9.1.2 两阶段封锁协议

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.

定义 9.4 (封锁点)

封锁点: 事务获得其最后封锁的时间.



定理 9.1

若一组事务均服从两阶段封锁协议, 则它们的调度一定是可串行化的.

事务调度等价于和其封锁点顺序一致的串行调度.



证明 令 $\{T_0, T_1, \dots, T_n\}$ 是参与调度 S 的事务集. 如果 T_i 对数据项 R 加 A 型锁, T_j 对数据项 R 加 B 型锁, 且 $comp(A, B) = false$. 若 T_i 先于 T_j , 记作 $T_i \rightarrow T_j$, 得到一个优先图.

设 t_i 是 T_i 的封锁点, 若 $T_i \rightarrow T_j$, 则 $t_i \rightarrow t_j$.

若 $\{T_0, T_1, \dots, T_n\}$ 不可串行化, 则在优先图中存在环, 不妨设为 $T_0 \rightarrow T_1 \rightarrow \dots \rightarrow T_n \rightarrow t_0$, 则 $t_0 < t_1 < \dots < t_n < t_0$, 矛盾!

2PL 可以保证调度是可串行化的吗? 可以. 如上.

2PL 可以保证调度是无级联的吗? 不能保证. 在标准的基本 2PL 中, 一个事务可能读取到另一个尚未提交事务写入的数据 (即“脏读”).

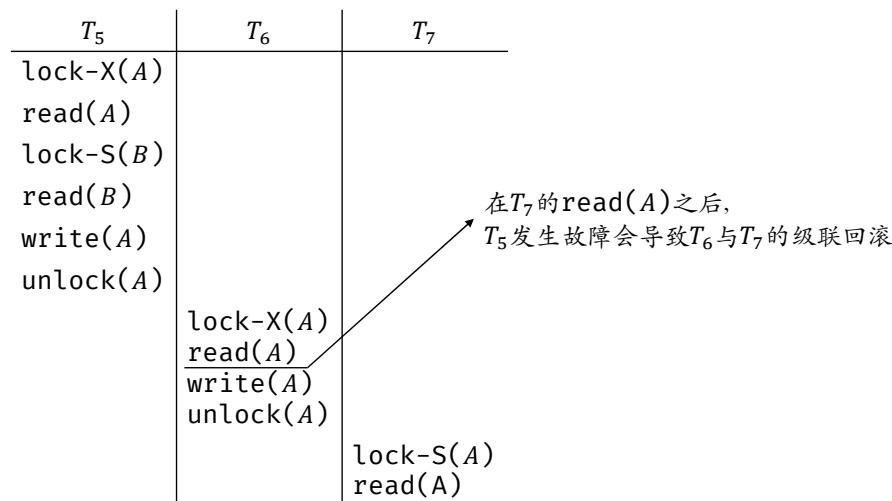


图 9.1: 2PL 的级联回滚

2PL 可以保证调度避免不可重复读吗? 不一定.

严格 2PL(strict two-phase locking protocol): 要求 X 锁是长锁.

严厉 2PL(rigorous two-phase locking protocol): 要求 S 锁和 X 锁均是长锁.

9.1.3 先读后写: 锁转换

请考虑下面的两个事务:

$$T_1 : \text{read}(a_1); \text{read}(a_2); \dots; \text{read}(a_n); \text{write}(a_1);$$

$$T_2 : \text{read}(a_1); \text{read}(a_2); \text{display}(a_1 + a_2);$$

如果我们采用两阶段封锁协议, 则 T_1 必须以排他模式封锁 a_1 . 此时, 两个事务的任何并发执行都相当于一个串行执行. 所以, 我们可以在开始时以共享模式封锁 a_1 , 随后把这个锁变更为排他锁.

上面介绍的就是锁转换 (**lock conversion**):

1. 升级 (upgrade): 从共享到排他的转换;
2. 降级 (downgrade): 从排他到共享的转换.
3. 升级只能发生在增长阶段, 而降级只能发生在缩减阶段。

问题 9.1 升级锁和重新申请锁有区别吗?

- 排队顺序不同
- 升级锁已经在队列中, 可以更快获得批准; 重新申请锁需要从头排队, 获取锁更慢.

问题 9.2 在哪个隔离性级别下会出现锁转换?

- repeated committed 以及更高的隔离级别
- Read committed 和 read uncommitted 不出现锁转换, 因为锁转换要求一直持有读锁, 没有释放.

但是锁转换会带来死锁的问题:

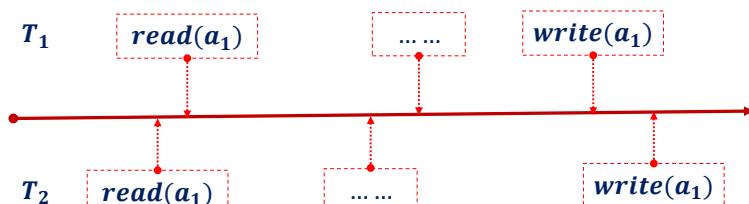


图 9.2: 转换死锁

如果都使用升级锁: T_1 要升级锁的时候, 会被 T_2 的读锁阻止; T_2 要升级锁的时候, 会被 T_1 的读锁阻止

9.1.4 先读后写: 更新锁

定义 9.5 (更新锁 (U 锁, Update lock))

当一个事务查询数据以便将来要进行修改时, 可以对数据项施加更新锁. (这是一种相容性介于 S 锁和 X 锁之间的锁.)

如果事务修改资源, 需将更新锁转换为排它锁.

一次只有一个事务可以获得资源上的更新锁.



此时的相容矩阵 comp(A,B):

请求锁模式 A	现有锁模式 B		
	S	X	U
S	是	否	是
X	否	否	否
U	是	否	否

表 9.2: 封锁的相容矩阵

9.1.5 封锁粒度

封锁对象：属性值、元组、关系、数据库、索引项、索引、物理页、块

- 封锁粒度大：锁的数目少，开销小；冲突几率大，并发度低
- 封锁粒度小：锁的数目多，开销大；冲突几率小，并发度高

定义 9.6 (事务的完整性相关域)

事务的完整性相关域：只封锁与操作有关的数据对象.



但是问题出现了：比如对整个表的封锁会和对行的封锁之间产生冲突！

如何检测到不同粒度之间的锁的隐含冲突？

如果有事务 T1 对某元组加了 S 锁，而事务 T2 对该元组所在的关系加了 X 锁，实则隐含地 X 封锁了该元组，从而造成矛盾。

定义 9.7 (意向锁 (I 锁, Intend lock))

当为某节点加上 I 锁，表明其某些内层节点已发生事实上的封锁，防止其它事务再去显式封锁该节点。表示它的后裔节点拟进行封锁。

从封锁层次的根开始实施 I 锁，依次占据路径上的所有节点，直至要真正进行显式封锁的节点的父节点为止。



此时的相容矩阵 comp(A,B)：

请求锁模式 A	现有锁模式 B		
	I	X	S
I	是	否	否
X	否	否	否
S	否	否	是

表 9.3：封锁的相容矩阵

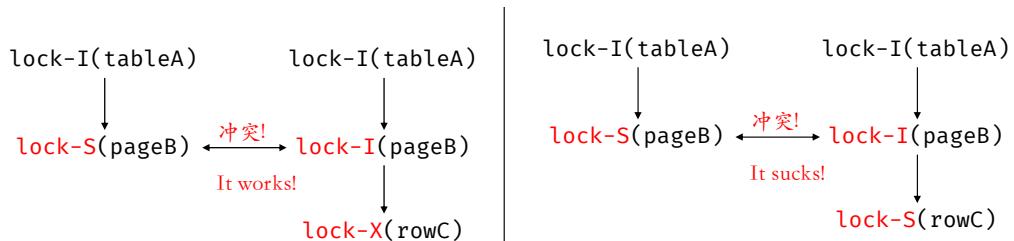


图 9.3：意向锁 I 的不足之处

I 锁没有揭示内层锁的性质。

定义 9.8 (IS 锁)

对一个数据对象加 IS 锁，表示它的后裔节点拟（意向）加 S 锁。



定义 9.9 (IX 锁)

对一个数据对象加 IX 锁，表示它的后裔节点拟（意向）加 X 锁。



此时的相容矩阵 comp(A,B)：

请求锁模式 A	现有锁模式 B			
	IS	S	IX	X
IS	是	是	是	否
S	是	是	否	否
IX	是	否	是	否
X	否	否	否	否

表 9.4: 封锁的相容矩阵

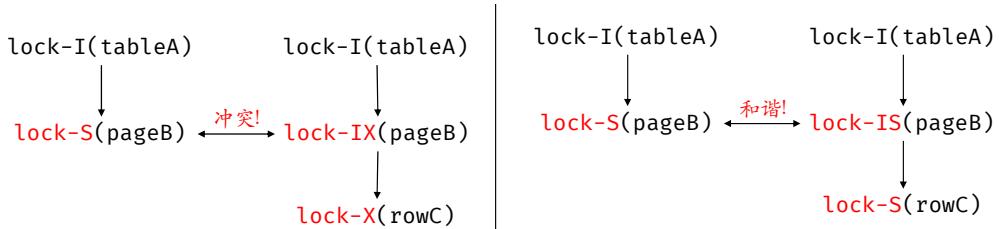


图 9.4: 意向锁 I 的细化

$SIX = S + IX$: 对表加 SIX 锁, 表示该事务要读整个表 (S 锁), 同时会更新个别元组 (IX 锁).

此时的相容矩阵 $comp(A,B)$:

请求锁模式 A	现有锁模式 B					
	IS	S	U	IX	SIX	X
IS	是	是	是	是	是	否
S	是	是	是	否	否	否
U	是	是	否	否	否	否
IX	是	否	否	是	否	否
SIX	是	否	否	否	否	否
X	否	否	否	否	否	否

表 9.5: 封锁的相容矩阵

9.1.6 码范围锁

Listing 9.1: 幻象例子

```
-- 事务 T1:
SELECT * FROM student WHERE score > 90; -- 返回张三
-- 事务 T2:
INSERT INTO student VALUES ('李四', 95); -- 成功
-- T1 再次执行:
SELECT * FROM student WHERE score > 90; -- 返回张三和李四, 幻读发生
```

对于 `select * from R where A >=10 and A <=20`, 在两次读取之间可能出现幻象 (也就是两次读取之间出现了插入, 导致满足条件的元组发生了改变).

- 此时单纯封锁现有数据是无效的. 如何防止其他事务往区间 [10, 20] 插入数据?
- 一种解决思路: 封锁整个表. 并发度极低!
- 第二种解决思路: 码范围锁. 条件: 查询/操作字段上必须有合适的索引; 当前事务隔离级别为: REPEATABLE READ / SERIALIZABLE.

码范围锁通过覆盖索引行和索引行之间的范围来工作, 因为第二个事务在该范围内进行任何行插入、更新或删除操作时均需要修改索引, 而码范围锁覆盖了索引项, 所以在第一个事务完成之前会阻塞第二个事务的进行.

码范围锁包括:

- **范围组件**. 表示保护两个连续索引项之间范围的锁模式 (RangeT).
- **行组件**. 表示保护索引项的锁模式 (K).
- 把这两部分用下划线连接, 如 RangeT_K.

Listing 9.2: 码范围锁典型触发表例

```
-- 满足所有触发条件的例子:
```

```
BEGIN;
```

```
SELECT * FROM student
WHERE score > 90
FOR UPDATE; -- 或 LOCK IN SHARE MODE
```

```
-- 使用的是InnoDB引擎、字段有索引、使用范围条件、隔离级别为RR或更高
```

此时会触发 Next-Key Lock (即 Key-Range Lock), 锁定:

- 现有符合条件的记录
- 它们前后的间隙
- 插入位置 (防止幻读)

范围	行	模式	描述
RangeS	S	RangeS_S	共享范围, 共享资源锁; 可串行范围扫描
RangeS	U	RangeS_U	共享范围, 更新资源锁; 可串行更新扫描
RangeI	NULL	RangeI_N	插入范围, 空资源锁; 用于在索引中插入新码之前测试范围
RangeX	X	RangeX_X	排它范围, 排它资源锁; 用于更新范围中的码

表 9.6: 码范围锁模式 (SQL Server)

请求模式	现有的授权模式						
	S	U	X	RangeS_S	RangeS_U	RangeI_N	RangeX_X
共享 (S)	是	是	否	是	是	是	否
更新 (U)	是	否	否	是	否	是	否
排它 (X)	否	否	否	否	否	是	否
RangeS_S	是	是	否	是	是	否	否
RangeS_U	是	否	否	是	否	否	否
RangeI_N	是	是	是	否	否	是	否
RangeX_X	否	否	否	否	否	否	否

表 9.7: 码范围锁的相容矩阵

码范围锁实际上就是: 1. 对表里这个范围上锁, 2. 对这个表的索引上锁.

现在考虑下面的查询:

```
select *
from employees
where last_name between 'Delaney' and 'DuLaney'
```

根据下面的这个表的索引9.5, 我们知道, 实际上读到的是: Dallas, Donovan, Duluth.

为了防止幻读, 数据库会在范围两侧的区间加码范围锁: 加 RangeS_S 锁在

- 区间 1: (Dallas, Donovan]
- 区间 2: (Donovan, Duluth]

此时其他事务无法插入 Dashagua, 因为 Dashagua 位于 (Dallas, Donovan] 之间.

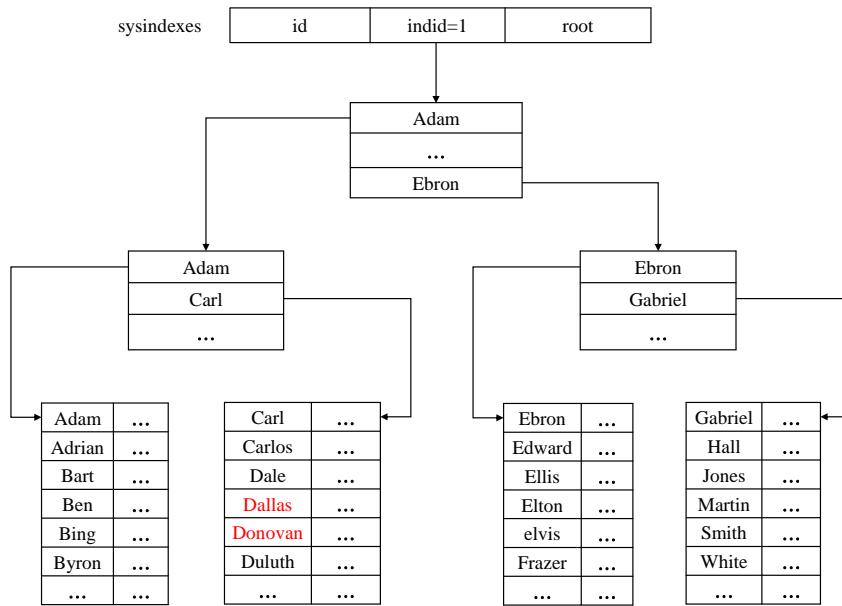


图 9.5: 码范围锁示例

9.1.7 一些具体系统中的特殊锁模式

- 闩锁
- SQL Server 的锁模式
- MySQL 的锁模式

Latch(闩锁) 是数据库系统和操作系统中用于短时间内保护关键资源的一种轻量级同步机制. 用于保护数据库内部结构 (如缓存页、索引节点等) 的并发访问, 通常持续时间非常短.

在 SQL Server 中, 模式锁 (Schema Lock) 是一种特殊的锁类型, 用于保护数据库对象的结构 (schema), 例如表结构、索引、存储过程等, 它防止其他会话对正在变更结构的对象做冲突操作, 确保元数据的一致性.

- 模式修改锁 (Schema Modification, Sch-M): 改结构时获取, 互斥;
- 模式稳定锁 (Schema Stability, Sch-S): 读结构时获取, 允许并发.
- MySQL 中的行锁模式: MySQL 的行锁只出现在基于索引的查找中.
- lock_rec_not_gap(记录锁, 不包含间隙): 只锁定具体的索引记录 (行), 不锁定它前后的间隙. 适用于 READ COMMITTED 隔离级别时的普通行锁.
- lock_gap(间隙锁): 锁定两个索引记录之间的间隙, 不锁定具体的记录. → 避免幻读. 在可重复读 (REPEATABLE READ) 隔离级别中常见.
- lock_ordinary(普通锁, 也叫 Next-Key Lock): 锁定一个具体的记录及其前面的间隙, 是 InnoDB 默认使用的行锁类型.
- lock_insert_intention(插入意向锁): 插入意向 gap 锁, 插入记录时使用, 是 lock_gap 的一种特例.
- 自增锁是为了保护 AUTO_INCREMENT 计数器, 保证多个事务插入数据时, 自增列的值不会重复或出现乱序. 自增锁是一个表级的互斥锁.

9.1.8 封锁带来的问题

9.1.8.1 阻塞

缺少索引会引起阻塞, 考虑下面的查询:

```
create table T1 (id int, col1 char(10))
insert into T1 values ( 101, 'A'),(102, 'B'),(103, 'C')
```

连接1	连接2
<pre>begin tran update T1 set col1 = 'a' where id = 101</pre>	
	<pre>begin tran update T1 set col1 = 'a' where id = 103</pre>

图 9.6: 缺少索引引起的阻塞

1. 连接 1 在表上没有找到索引, 因此进行全表的查询, 全部加 U 锁;
2. 找到 101, 升级为 X 锁, 释放其余的 U 锁;
3. 连接 2 在表上没有找到索引, 因此进行全表的查询, 全部加 U 锁;
4. 但是在试图对 101 加锁的时候和 X 锁冲突, 此时出现了阻塞.

9.1.8.2 死锁 (Deadlock)

两个事务都封锁了一些数据对象, 并相互等待对方释放另一些数据对象以便对其封锁, 结果两个事务都不能结束, 则发生死锁.

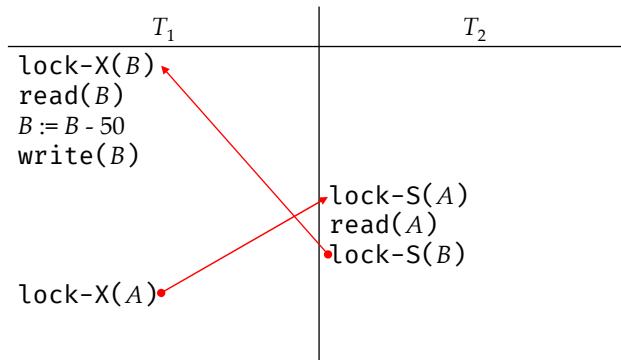


图 9.7: 死锁

死锁发生的条件:

1. **互斥条件**: 事务请求对资源的独占控制
2. **占有等待条件**: 事务已持有一定资源, 又去申请并等待其它资源
3. **非抢占条件**: 直到资源被持有它的事务释放之前, 不能将该资源强制从持有它的事务夺去
4. **循环等待条件**: 存在事务相互等待的等待圈

定理 9.2

在条件 1、2、3 成立的前提下, 条件 4 是死锁存在的充分必要条件.



死锁检测: 构造事务之间的等待图. 事务 T_i 正在等待 T_j 则画一条有向边 $T_i \rightarrow T_j$. 如果有环就说明有死锁.
如何避免死锁:

1. 选择较低的隔离性级别
2. 封锁粒度小一些
3. 不同事物访问一组对象, 按照相同顺序访问

注 通常, 数据库系统会选择回滚较年轻的事务或者根据其他策略(如估计回滚成本)来决定哪个事务被回滚.
也就是9.7中会选择回滚 T_2 .

预防死锁: 破坏占有等待条件

1. 预先占据所需的全部资源: 要么一次全部封锁, 要么全不封锁.
(a). 缺点: 难于预知需要封锁哪些数据并且数据使用率低
2. 所有资源预先排序, 事务按规定顺序封锁数据.

预防死锁: 破坏非抢占条件

1. 使用抢占与事务回滚: 通过规定事务之间的优先级(老事务优先级高于新事务)来破坏死锁的非抢占条件.

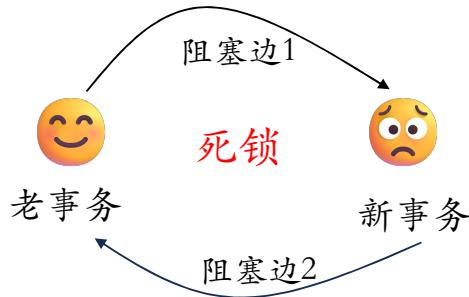


图 9.8: 死锁出现的条件

1. **wait-die**: 只允许老事务等待新事务(阻塞边1), 出现新事务等待老事务时(阻塞边2), 回滚新事务.
2. **wound-wait**: 只允许新事务等待老事务(阻塞边2), 出现老事务等待新事务时(阻塞边1), 回滚**新事务**. 老事务不等待, 直接杀死新事务.

例题 9.1 对于调度 $\text{start}(T_1), \text{start}(T_2), w_1(R_1), w_2(R_2), r_1(R_2), r_2(R_1)$, 分别使用 wait-die 或者 wound-die 来判断 T_2 在哪一步被回滚.

解答. 对于 wait-die, 在 $r_2(R_1)$ 的时候 T_2 开始等待, 被回滚. 对于 wound-die, 在 $r_1(R_2)$ 的时候 T_1 开始等待, 此时杀死 T_2 , T_2 开始回滚.

死锁检测和恢复:

1. 超时法: 如果等待封锁的时间超过限时, 则撤消该事务.
2. 等待图法: 在 LOCK_MONITOR 中实现.

9.1.8.3 活锁 (Live lock)

定义 9.10 (活锁 (Live lock))

可能存在某个事务永远处于等待状态, 得不到执行, 称之为**活锁**(饿死).



活锁的例子: 身处众多读锁中的写锁.

避免活锁的策略是遵从“先来先服务”的原则: 按请求封锁的顺序对各事务排队.

9.1.9 其他锁相关概念

定义 9.11 (锁管理器)

- 事务向锁管理器发送封锁请求和释放请求
- 锁管理器维护一个锁表记录锁的授予情况和处于等待状态的封锁请求



定义 9.12 (锁表)

锁表一般作为内存中的 hash 表, 按被封锁对象的名字建立索引.



...

"Unimportant. There's a million things I haven't done."

9.2 基于时间戳的协议

事务时间戳的分配:

- 每个事务 T_i 进入系统被分配一个时间戳 $TS(T_i)$.
- 如果 T_j 晚于 T_i 进入系统, 则 $TS(T_i) < TS(T_j)$.
- 回滚的事务重新启动, 分配新的时间戳.
- 时间戳顺序决定了串行化顺序. 因此, 若 $TS(T_i) < TS(T_j)$, 则系统必须保证所产生的调度等价于事务 T_i 出现在事务 T_j 之前的一个串行调度.
- 回滚违反发出串行性操作的事务.

要实现这样一种机制, 我们将每个数据项 Q 与两个时间戳值相关联:

1. $WT(Q)$: 所有执行 $\text{write}(Q)$ 的事务中最大的时间戳;
2. $RT(Q)$: 所有执行 $\text{read}(Q)$ 的事务中最大的时间戳.

同时为了避免可能的脏读, 我们为数据项 Q 设置提交位 $C(Q)$: 表示拥有 Q 上写时间戳的事务是否提交.

定义 9.13 (时间戳排序协议 (timestamp-ordering protocol))

保证任何有冲突的 read 和 write 操作按时间戳的次序执行, 该协议运作方式如下:

- 假设事务 T_i 发出 $\text{read}(Q)$.
 - 如果 $TS(T_i) < WT(Q)$: T_i 需读入的值已经被覆盖, $\text{read}(Q)$ 操作被拒绝, 回滚 T_i ;
 - 如果 $TS(T_i) \geq WT(Q)$:
 - 若 $C(Q)$ 为真, 则执行 $\text{read}(Q)$ 操作, $RT(Q) = \max(RT(Q), TS(T_i))$;
 - 若 $C(Q)$ 为假, 则推迟到 $C(Q)$ 为真或者写 Q 的事务中止.
- 假设事务 T_i 发出 $\text{write}(Q)$.
 - 如果 $TS(T_i) < RT(Q)$: T_i 产生的值是先前所需要的值, $\text{write}(Q)$ 操作被拒绝, 回滚 T_i ;
 - 如果 $TS(T_i) < WT(Q)$: T_i 产生的值已经被其后的写事务覆盖, 跳过 T_i 的 $\text{write}(Q)$; (**Thomas 写规则: 写操作在更晚的写操作已经发生时可以跳过.**)
 - 如果 $TS(T_i) > RT(Q) \wedge TS(T_i) > WT(Q)$, 则执行 T_i 的 $\text{write}(Q)$, $WT(Q) = TS(T_i)$.



如何实现时间戳?

1. 计数器
2. 向量时钟
3. 物理时钟
4. 逻辑时钟

$TS(T_i)$	T_1	T_2	T_3	A	B	C
	200	150	175	RT=0 WT=0	RT=0 WT=0	RT=0 WT=0
$r_1(B)$					RT=200	
		$r_2(A)$		RT=150		
			$r_3(C)$			RT=175
$w_1(B)$					WT=200	
$w_1(A)$				WT=200		
		$w_2(C)$				
		中止				
			$w_3(A)$			

TS(T_2) < RT(Q) TS(T_3) < WT(Q)
 write操作被拒绝, 回滚 T_2 write操作被跳过, 但是不回滚 T_3

图 9.9: 基于时间戳协议的练习

9.3 基于有效性检查的协议

在大部分事务是只读事务的情况下, 事务之间发生冲突的概率较低. 所以即使没有并发控制机制, 也可以达到一致性. → 采用减小并发开销的可替代方案可能是更好的.

减小开销所面临的困难是我们事先并不知道哪些事务将陷入冲突中, 为获得这种知识, 我们需要一种监控 (monitoring) 系统的机制.

定义 9.14 (有效性检查协议 (validation protocol))

有效性检查协议 (validation protocol) 要求每个事务 T_i 在其生命周期中按两个或三个不同的阶段执行, 这取决于该事务是一个只读事务还是一个更新事务.

- 读阶段. 在这一阶段中, 系统执行 T_i . 它读取各数据项的值并将它们保存在 T_i 的局部变量中, T_i 的所有 write 操作都是对 **局部的临时变量** 进行的, 并不对数据库进行真正的更新.
- 有效性检查阶段. 对事务 T_i 进行有效性检查的测试 (见下文). 这将决定是否允许 T_i 继续执行到写阶段而不违反可串行性. 如果事务的有效性检查的测试失败, 则系统终止这个事务.
- 写阶段. 若事务 T_i 通过了有效性检查的测试, 则保存 T_i 所执行的任何 write 操作结果的临时局部变量被拷入数据库. 对于只读事务忽略这个阶段.

并发执行事务的三个阶段是可以交叉执行的!

有效性检查的测试:

- 我们将三个不同的时间戳与每个事务 T_i 相关联:
 1. $StartTS(T_i)$: 事务 T_i 开始其执行的时间;
 2. $ValidationTS(T_i)$: 事务 T_i 完成其读阶段并开始其有效性检查阶段的时间;
 3. $FinishTS(T_i)$: 事务 T_i 完成其写阶段的时间.
- 我们令 $TS(T_i) = ValidationTS(T_i)$, 并且若 $TS(T_j) < TS(T_k)$, 则产生的任何调度必须等价于其中事务 T_j 出现在事务 T_k 之前的一个串行调度.
- 有效性协议的检查条件, 我们假设 T_1 已确认, T_2 待确认.
 1. $StartTS(T_2) < FinishTS(T_1) < ValidationTS(T_2)$: 检查 $RS(T_2) \cap WS(T_2) = \emptyset$.
 2. $ValidationTS(T_2) < FinishTS(T_1)$: 检查 $WS(T_2) \cap WS(T_1) = \emptyset \wedge RS(T_2) \cap WS(T_1) = \emptyset$.

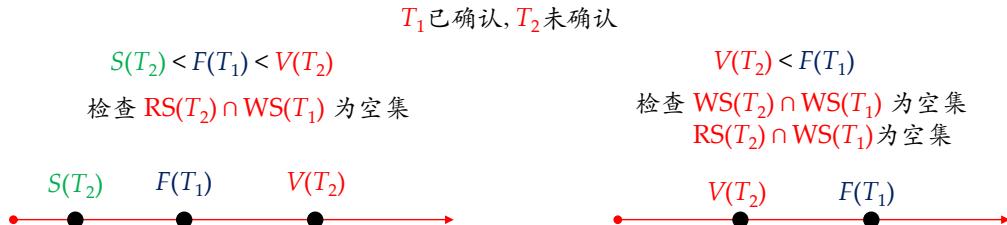


图 9.10: 有效性协议的检查条件

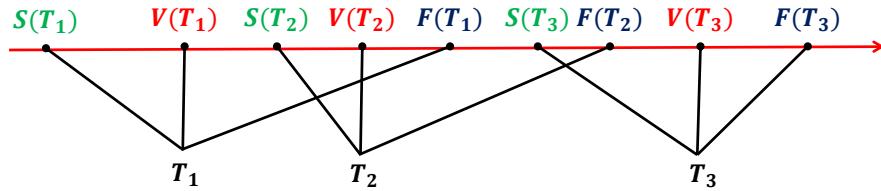


图 9.11: 有效性检查协议的例子

确认 T_2 :

- $S(T_2) < F(T_1)$: 检查 $RS(T_2) \cap WS(T_1) = \emptyset$ 是否成立;
- $V(T_2) < F(T_1)$: 检查 $WS(T_2) \cap WS(T_1) = \emptyset$ 是否成立.

确认 T_3 :

- $S(T_3) > F(T_1)$: 不需要判定 T_3 和 T_1 的相交性;
- $S(T_3) < F(T_2)$: 检查 $RS(T_3) \cap WS(T_2) = \emptyset$ 是否成立.

9.4 MVCC(Multi-Version Concurrency Control)

MVCC: Multi-Version Concurrency Control, 多版本并发控制.

数据行结构: InnoDB 每行数据增加三个隐藏列用于实现 MVCC:

- db trx_id: 插入或更新行的最后一个事务的全局标识符 (每个事务创建都会分配 id, 全局递增)
- db roll_ptr: 指向当前记录的前一个 undo log 版本
- db row_id: 行标识 (隐藏单调自增 id)

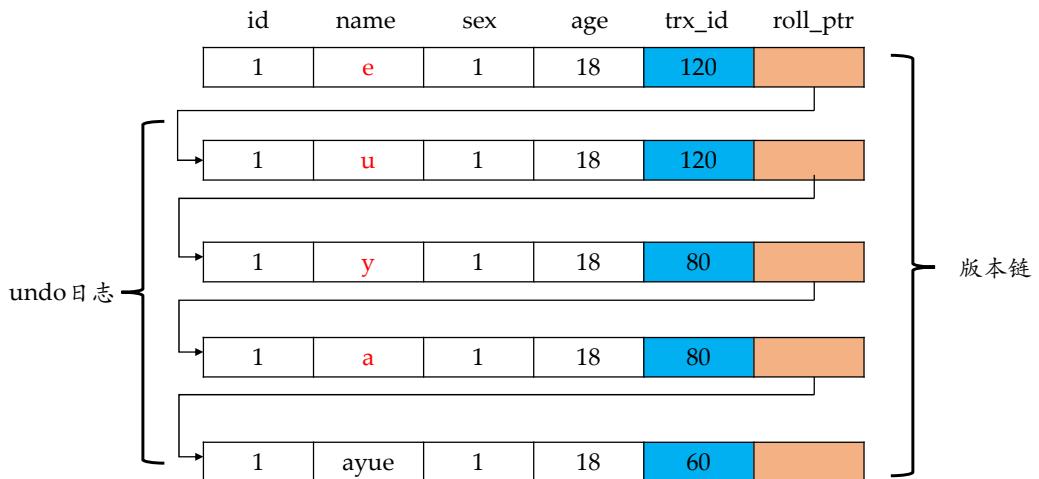


图 9.12: MySQL MVCC 实现: 数据行

读视图: 事务在进行快照读的时候会创建一个读视图 `read_view`.

- `current_trx_id`: 当前事务的 id;
- `alive_trx_list`: 读视图生成时刻系统中正在活跃的事务 id;
- `min_trx_id`: 上面的 `alive_trx_list` 中的最小事务 id;
- `max_trx_id`: 读视图生成时刻目前已创建过的事务 id 最大值 + 1.

可见性算法:

当一个事务读取某条记录 $record_i$ 时会追溯其 undo log 版本链, 找到第一个可以访问的版本, 而该记录的某一个版本 $db_trx_id(record_i)$ 是否能被这个事务读取到遵循如下可见性算法规则:

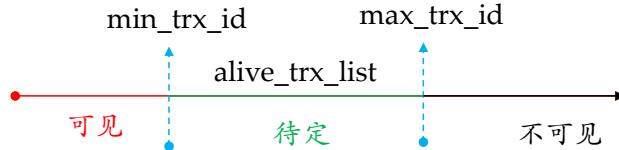


图 9.13: 可见性算法规则

- 当 $db_trx_id(record_i) < min_trx_id$ 时, 那么 $record_i$ 对于当前事务可见 (db_trx_id 在当前事务之前提交);
- 当 $db_trx_id(record_i) = current_trx_id$, 那么 $record_i$ 对于当前事务可见 (db_trx_id 是被当前事务自身生成或修改的);
- 当 $db_trx_id(record_i) > max_trx_id$ 时, 那么 $record_i$ 对于当前事务不可见 (db_trx_id 在当前事务之后提交);
- 当 $min_trx_id \leq db_trx_id(record_i) < max_trx_id$ 时, 而且 $db_trx_id(record_i) \in alive_trx_list$, 那么 $record_i$ 对于当前事务不可见 (db_trx_id 属于则说明这条记录还未提交, 对于当前操作的事务是不可见的);
- 当 $min_trx_id \leq db_trx_id(record_i) < max_trx_id$ 时, 而且 $db_trx_id(record_i) \notin alive_trx_list$, 那么 $record_i$ 对于当前事务可见 (db_trx_id 不属于则说明这条记录已经提交, 对于当前操作的事务是可见的).

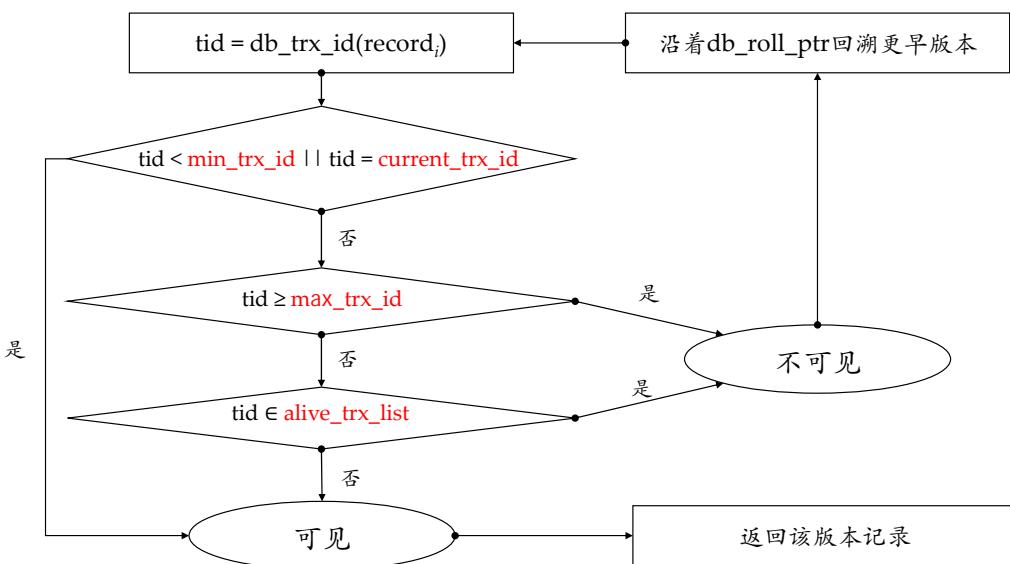


图 9.14: 可见性算法规则流程图

9.4.1 MySQL MVCC 在不同隔离性级别下的读视图

假设对于下面的两个事务 T_A 和 T_B , 有下面的调度:

T_A	T_B
begin tran(trx_id=101)	
	begin tran(trx_id=102)
select a ... where id = 1	
	update ... set a = 1 where id = 1
	commit
select a ... where id = 1	
commit	

图 9.15: MySQL MVCC 的一个例子

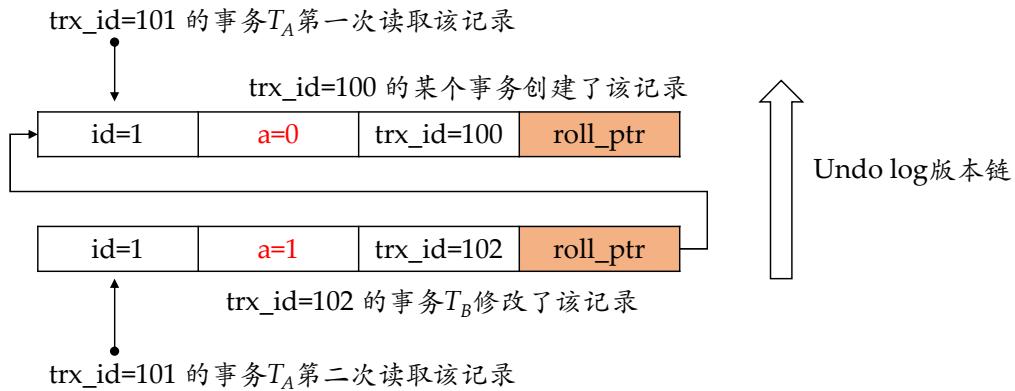


图 9.16: MVCC 读取

现在我们首先假定: T_A 处于 read committed 级别.

1. T_A 第一次读取时的 read_view:

- current_trx_id: 101
- alive_trx_list: [101, 102]
- min_trx_id: 101
- max_trx_id: 103

T_B 此时尚未提交, 只有前一个记录: $db_trx_id(record_{id=1}) < min_trx_id$, 从而它对 T_A 可见, 返回 $a = 0$.

2. T_A 第二次读取时的 read_view:

- current_trx_id: 101
- alive_trx_list: [101]
- min_trx_id: 101
- max_trx_id: 103

T_B 此时已经提交, 此时有: $min_trx_id \leq db_trx_id(record_{id=1}) < max_trx_id$, 而且很明显 T_B 已经不在活跃队列之中, 所以它对于 T_A 可见, 返回 $a = 1$.

read committed 下事务每次读取都会生成新的 read_view.

现在我们假设 T_A 处于 repeatable read 级别:

1. T_A 第一次读取时的 read_view:

- current_trx_id: 101
- alive_trx_list: [101, 102]
- min_trx_id: 101

- max_trx_id: 103

T_B 此时尚未提交, 只有前一个记录: $\text{db_trx_id}(\text{record}_{\text{id}=1}) < \text{min_trx_id}$, 从而它对 T_A 可见, 返回 $a = 0$.

2. T_A 第二次读取时的 read_view:

- current_trx_id: 101
- alive_trx_list: [101, 102]
- min_trx_id: 101
- max_trx_id: 103

T_B 此时已经提交, 此时有: $\text{min_trx_id} \leq \text{db_trx_id}(\text{record}_{\text{id}=1}) < \text{max_trx_id}$, 但是 T_B 还在活跃队列之中, 所以它对于 T_A 不可见, 继续沿着 db_roll_ptr 回溯其他版本.

repeatable read 下事务的 read_view 保持不变.

- read committed 的非一致性读总是读取被访问行的最新一份快照数据.
- repeatable read 的非一致性读总是读取事务开始时的行数据版本.

MVCC 不能解决幻读!!!

第十章 恢复控制

期末考试复习提纲

- | | |
|---|--|
| <ul style="list-style-type: none"><input type="checkbox"/> 故障类型<input type="checkbox"/> 备份概念及其类型<input type="checkbox"/> 日志定义、事务类型及其恢复操作、先写日志的 WAL 原则 | <ul style="list-style-type: none"><input type="checkbox"/> 检查点概念及其作用<input type="checkbox"/> 基本的故障恢复操作<input type="checkbox"/> ARIES 恢复算法所遵循的原则、所涉及到的重要数据结构、三个恢复阶段 |
|---|--|

10.1 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 10.1 (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 10.2 (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 10.3 (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.



定义 10.4 (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.



10.2 备份

定义 10.5 (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:

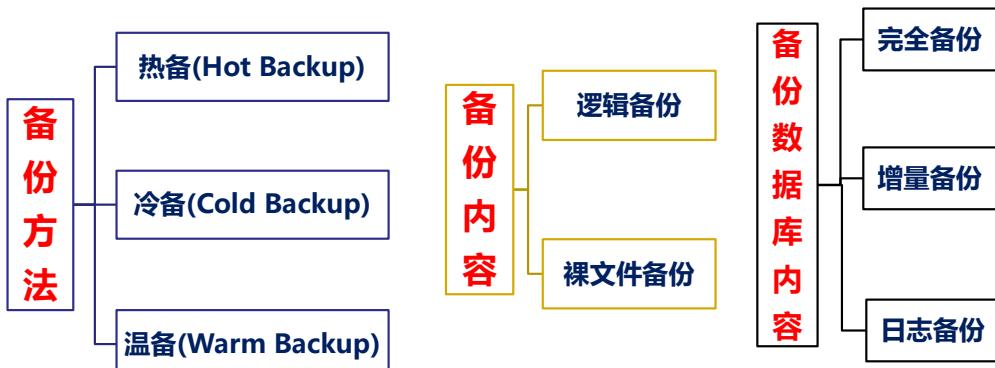


图 10.1: MySQL 备份

```
# 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/
mysql_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
```

10.3 日志

定义 10.6 (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 commit 标识。
2. **夭折事务**：日志文件中只有 start 标识，没有记录事务的 commit 标识。

基本的恢复操作：

1. 对圆满事务的更新日志执行 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: redo = redo².
2. 对夭折事务的更新日志执行 undo 操作, 即撤销该操作, 修改对象被赋予旧记录值. 幂等性: undo = undo².
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.

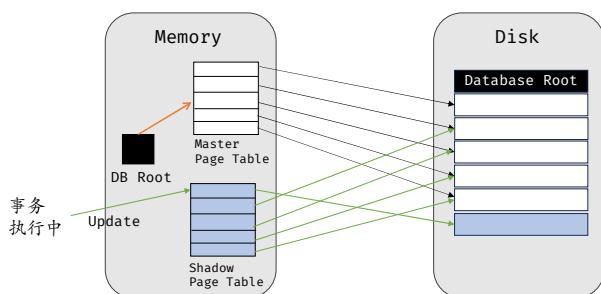


图 10.2: Shadow Paging: 正在修改的事务

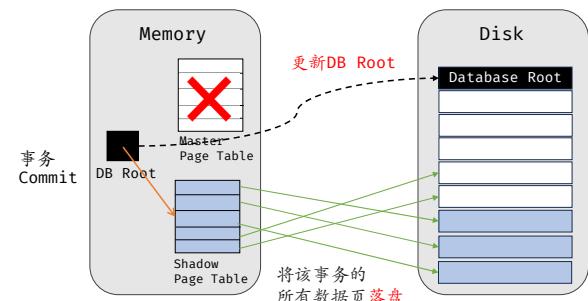


图 10.3: Shadow Paging: 事务 Commit

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 10.1 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.

10.4 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.

为什么不能先写数据库?

如果先写 DB, 则可能在写的中途发生系统崩溃, 导致内存缓冲区内容丢失, 而外存 DB 处于不一致状态, 由于日志缓冲区内容已破坏, 导致无法对 DB 恢复.

WAL 总可以保证恢复的一致性!!!

日志缓冲区和数据库缓冲区的写时机不同:

1. 同步 (synchronous) 写日志: 只有事务的相关日志已经完全在磁盘上了, 才会向进程发送该事务已提交的确认消息.
2. 异步 (asynchronous) 写缓冲区: 只需要将数据页的写入操作投递给操作系统即可, 不需要等待其完成.

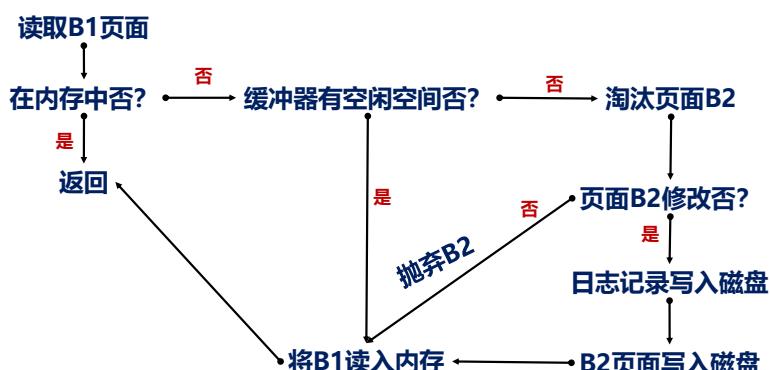


图 10.4: 读取一个页面的过程 (actually from 操作系统)

在主线程每秒一次的循环中, 将 undo log 缓冲器的内容刷新到重做日志文件中, 即便某个事务尚未提交. 由参数`innodb_flush_log_at_trx_commit`控制.

- 0 代表提交事务时, 并不立即刷出日志, 而是等待主线程每秒的刷新;
- 1 代表提交事务时, 将 undo log 同步写磁盘, 也即伴有 fsync() 的调用;

- 2 代表提交事务时, 将 undo log 异步写磁盘, 也即写入文件系统缓存中.
fsync 是昂贵操作, MySQL 一次事务提交最多会导致 3 次 fsync.
将多个并发提交的事务共享一次 fsync 操作:
- binlog_group_commit_sync_delay = N: 在等待 N 微秒后, 进行 binlog 刷盘操作
- binlog_group_commit_sync_no_delay_count = N: 达到最大事务等待数量, 开始 binlog 刷盘

10.5 故障恢复

10.5.1 事务故障恢复

事务故障恢复过程: 撤消事务已对数据库所做的修改

- 反向扫描日志文件, 查找该事务的更新操作
- 对事务更新操作执行 undo 操作, 即将事务更新前的旧值写入数据库
- 继续反向扫描日志文件, 查找该事务的其他更新操作, 并做同样处理
- 直至读到事务的开始标识, 结束事务故障恢复过程

为什么同一事务的日志记录需要反向链接在一起? 加快撤销速度.

反向 undo: 保持一致性.

10.5.2 系统故障恢复

系统故障造成不一致状态的原因

- 未完成事务对数据库的更新已写入数据库
- 已提交事务对数据库的更新未写入数据库

系统故障恢复过程:

- 正向扫描日志文件, 将圆满事务记入重做队列, 将夭折事务记入撤消队列;
- 反向扫描日志, 对撤消队列中事务 T_i 的每条日志记录执行 undo 操作;
- 正向扫描日志文件, 对重做队列中事务 T_i 的每条日志记录执行 redo 操作.

10.5.3 介质故障恢复

介质故障恢复过程:

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

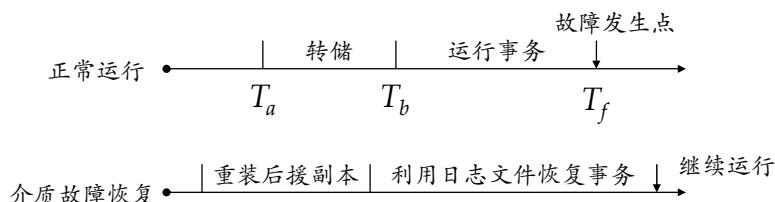


图 10.5: 介质故障恢复过程

10.6 检查点 (Checkpoint)

如何减少系统故障恢复过程中必须重做的事务数量?

- 当发生系统故障时, 我们必须搜索整个日志, 以决定哪些事务需要 redo, 哪些需要 undo
- 大多数需要被重做的事务其更新已经写入了数据库中 (redo²)

- 尽管对它们重做不会造成不良后果, 但会使恢复过程变得更长
如何确保已提交事务的结果已经写入数据库了?
- 保证在检查点时刻磁盘上日志文件与数据库的内容是一致的 (就像 Ctrl+S 一样)

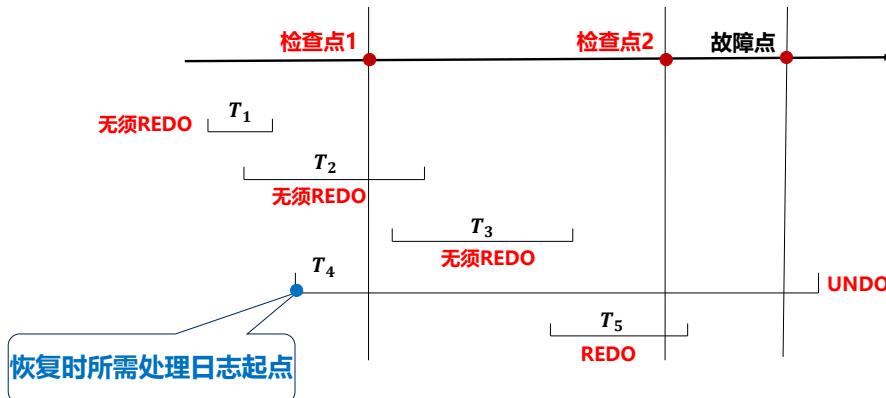


图 10.6: 检查点在系统故障恢复中的作用

10.6.1 一致性检查点

一致性检查点: Consistent Checkpoint

一致性检查点是最朴素的实现, 步骤如下:

1. 停服: 禁止新事务启动, 等待正在执行的事务结束;
2. 遍历 Buffer Pool, 将所有 Dirty Page 刷盘;
3. 记录一条 Checkpoint Log;
4. 恢复服务: 允许所有事务正常执行.

Recovery 直接从 Checkpoint Log 开始, 之前的 Log 一律跳过.

10.6.2 模糊检查点

模糊检查点: Fuzzy Checkpoint

1. 记录一条 Checkpoint Begin Log;
2. 输出活跃事务列表 ATT(Active Transaction Table) 和脏页表 DPT(Dirty Page Table);
3. 记录一条 Checkpoint End Log, 包括 ATT 和 DPT, 然后把 Log 刷盘;
4. 把 Checkpoint Begin Log 的 LSN 信息记录到 Master Record 中.

Master Record: 磁盘上的一个文件, 记录最近一次 Checkpoint Log 的 LSN, 通过它可以快速定位最近一次 Checkpoint 的 Log.

fuzzy checkpoint 操作之所以名为 fuzzy (模糊), 意思就是并不知道 checkpoint 操作何时结束, 只知道何时开始, 只需要在 checkpoint-end 结束的时候, 把 checkpoint-begin 时计算的 ATT, DPT 内容一起写入即可.

可以看到, 这个算法既不要求开始时所有写事务都结束, 也不要求进行时停止所有写事务, 极大提升了并发性能.

下面来看一个例子来理解 fuzzy-checkpoint. 根据图10.7:

- 事务 T1: 修改了页面 P_1 , 之后提交.
- 事务 T2: 修改了页面 P_2 , 还未提交之前就开始了 checkpoint 操作.
- 第一次 checkpoint 操作: 在开始 checkpoint 时, 系统中事务 T_2 还在进行, P_2 为脏页面, 于是 $ATT = \{T_2\}$, 以及 $DPT = \{P_2\}$, 于是这一次 checkpoint-end 日志, 仅把已提交的事务 T_1 的修改同步到了数据库文件上, 即 $A = 120$. 注意, 事务 T_3 在 checkpoint-begin 操作之后, 所以并不会加入到这两个表中.
- 结束了第一次 checkpoint 操作之后, MasterRecord 修改为 checkpoint-begin 操作的 LSN.

- 继续到第二次 checkpoint 时, 这时候事务 T_2 已经完成, 而事务 T_3 还在进行, 于是这一次 checkpoint 时, $ATT = \{T_3\}$, 以及 $DPT = \{P_3\}$, 这样会把第一次 checkpoint 到第二次 checkpoint 之间的操作落盘, 如何落盘呢? 这时候会遍历这中间的 WAL 日志, 来修改 ATT 和 DPT 这两个表, 一旦一个事务变得不活跃, 就落盘.

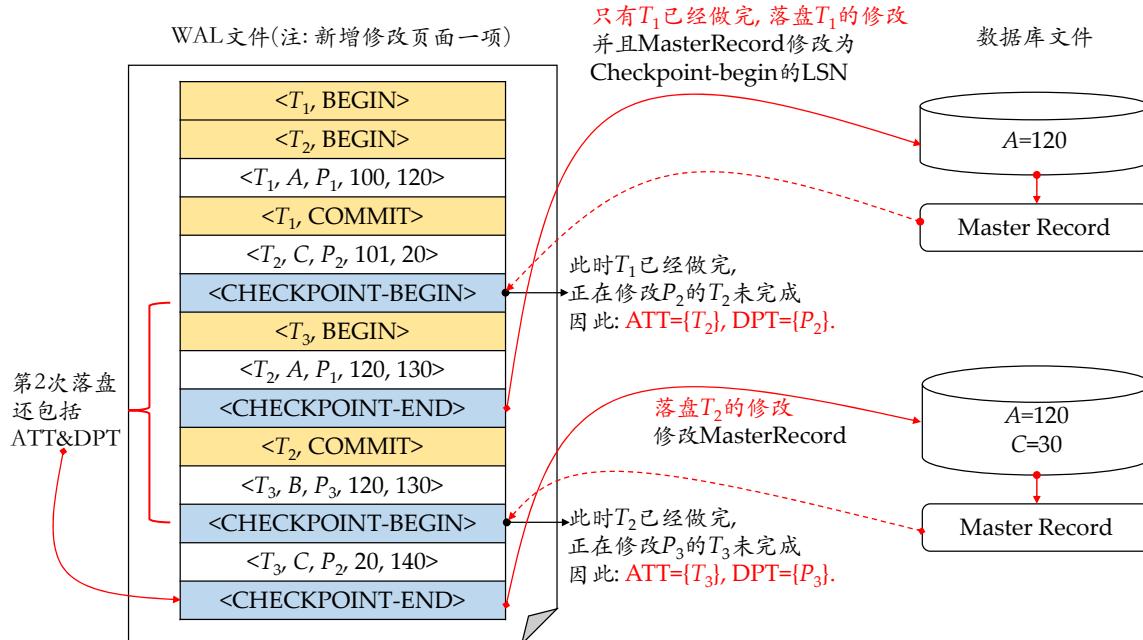


图 10.7: 模糊检查点的一个例子

如何处理检查点生成过程中重复变脏的页面?

- 为避免在一次检查点生成过程中重复将页面写入磁盘, 为每个页面设置标志位, 开始时所有的位都相同 (都为 0 或 1);
- 当检查点检查到某个页面时, 它将其标志位翻转, 直接跳过标志位相反的页面.

如何快速完成检查点生成过程中的写盘动作?

- 尽量保证磁盘页的写入是连续的
 - 缓冲区中非连续页面可以被一次聚集写入 (gather-write) 磁盘
 - 检查点线程按照缓冲区编号顺序扫描页面, 当发现脏页时, 检查与该页在磁盘上连续的其他页面是否也是脏的
 - 假如它看到页面 5 是脏的, 它可能会写入页面 10、25、38、500 等, 这些页面在磁盘上是连续的
- MySQL 检查点执行时机:
- Master Thread Checkpoint: 每秒或每 10 秒刷出一定比例的脏页
 - FLUSH_LRU_LIST Checkpoint: LRU 列表中空闲页不够时淘汰的页面中有脏页
 - Dirty Page too much Checkpoint: innodb_max_dirty_pages_pct
 - Async/Sync Flush Checkpoint: redo_lsn - checkpoing_lsn 超过日志文件大小 75%

定义 10.7 (最小恢复 LSN)

最小恢复 LSN(MinLSN) 是面这些 LSN 中的最小 LSN:

- 检查点起点的 LSN
- 最老的活动事务起点的 LSN



LSN11	LSN12	LSN13	LSN14	LSN15	LSN16	LSN17	LSN18	LSN19	LSN20
T1begin	T2begin	T2write	checkpoint	T1write	T1commit	checkpoint	T3begin	T3commit	checkpoint
			LSN11			LSN12			LSN12

图 10.8: MinLSN 计算示例

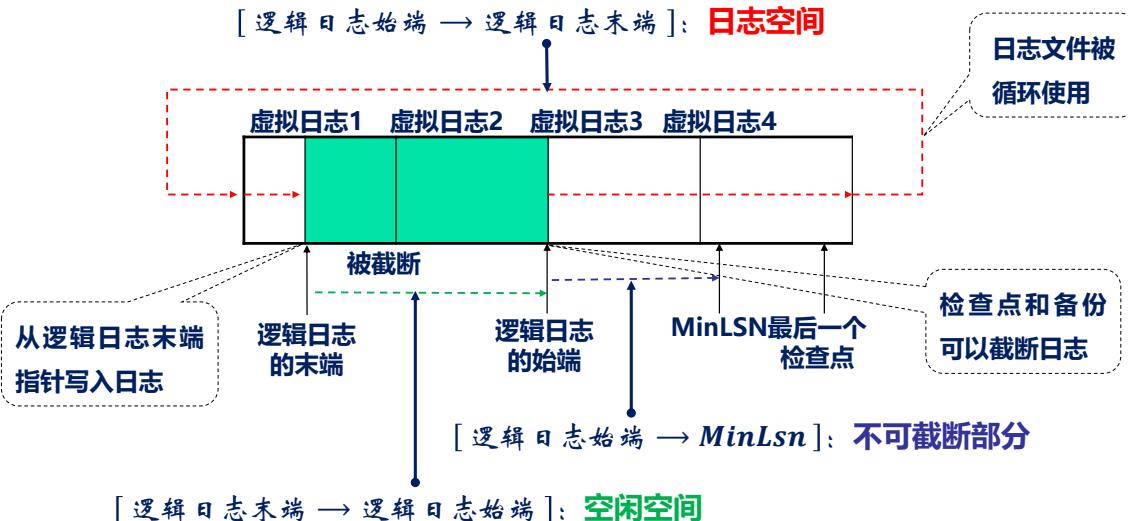


图 10.9: MinLSN 与物理日志

10.7 ARIES 恢复算法

下面介绍 ARIES 恢复算法 [5].

10.7.1 Buffer Manager(BM) 的实现策略

定义 10.8 (Steal Policy)

是否允许未提交事务的修改在持久化存储上生效 (Whether the DBMS allows an uncommitted txn to overwrite the most recent committed value of an object in non-volatile storage), 被称为 Steal policy.

- 允许未提交事务的修改持久化存储上生效 → Steal policy.
- 不允许 → No Steal Policy.



定义 10.9 (Force Policy)

一个事务在提交之前是否需要将所有修改同步到持久化存储上 (Whether the DBMS requires that all updates made by a txn are reflected on non-volatile storage before the txn is allowed to commit.)

- 必须将事务的所有修改都同步到持久化存储上, 事务才被允许提交 → Force policy.
- 不必 → No force policy.



Shadow Paging 是采用 no-steal+force 策略的方案.

WAL 是 steal+no-force 的策略.

checkpoint 的问题: 要求 checkpoint 在进行的时候, 不得有任何的写事务在进行, 换言之 checkpoint 操作会影响写事务的进行. 所以, 如果 checkpoint 的频率太高, 影响写事务; 频率太低, 又会让单次 checkpoint 时间很长.

ARIES 给出了 checkpoint 的优化做法, 通过引入了一种重要的变量 LSN(Log Sequence Number). ARIES 算法也是 no force + steal.

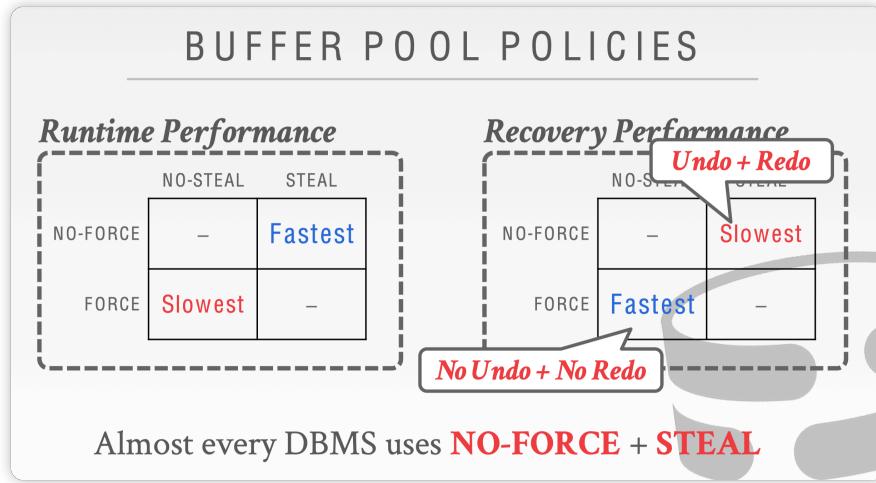


图 10.10: Buffer Pool Policies

10.7.2 日志类型

下面以 `insert into X values(1)` 为例:

- 物理日志 (Physical Log): 描述具体某一个 page 的修改操作.

```
<T1,
Table=X,
page=20,
Offset=50,
Before Image=(nil),
After Image=(1)>
```

- 逻辑日志 (Logical Log): 逻辑日志的本质就是对更新语句 (update query) 本身的落盘.

```
<T1,
insert into X values (1)>
```

- 物理逻辑日志 (Physiological Log): 每一条日志仅仅涉及一个 page 的修改 + 日志内容为更新语句 (update query) 本身.

```
<T1, Table=X,
page=20,
Type=Insert,
Data=(1)>
<T1, Index=X_pkey,
page=40,
Type=Insert,
Data=(1)>
```

InnoDB 的 Redo Log 属于 Physiological Log, Undo Log 属于 Logical Log.

MySQL Binlog 的 Row-Based 和 Statement-Based Log 属于 Logical Log.

	数据量	幂等性	redo 性能	undo 性能
Physical Log	大	成立	快	快
Logical Log	小	不成立	不支持	慢
Physiological Log	中	不成立	快	快

表 10.1: 各种日志类型对比

为什么需要逻辑 Undo 日志?

当修改索引时, 如何保护索引页?

- 传统封锁技术: 事务在更新一个数据项时给它施加 X 锁, 直至事务结束.
- 如果事务 T 向 B+ 树插入一项, 有可能造成页面拆分, 需要从根到叶都加 X 锁, 并且保持到事务结束, 并发性极低;
- 可以采取闩锁, 使锁较早释放;
- 如果在事务提交前释放了闩锁, 则其它事务可执行插入或删除操作, 造成对 B+ 树结点的进一步改变.
- 如果使用物理 undo 执行事务回滚, 也即将 B+ 树内部结点(执行插入操作前)的旧值写回, 那么其它事务在其后执行的插入或删除操作可能会丢失.
- 因此: 插入操作必须通过一个逻辑 undo(删除操作)来完成.

总结: ARIES 算法遵循的一些原则

- ARIES 算法遵循 WAL 原则;
- ARIES 算法使用模糊检查点;
- ARIES 算法的 BM 遵循 Non-Force, Steal;
- ARIES 算法使用 Page-oriented(大约就是物理或者逻辑日志)的 Redo, 使用 Logical 级别的 Undo.

10.7.3 ARIES 恢复算法中的数据结构

日志记录结构:

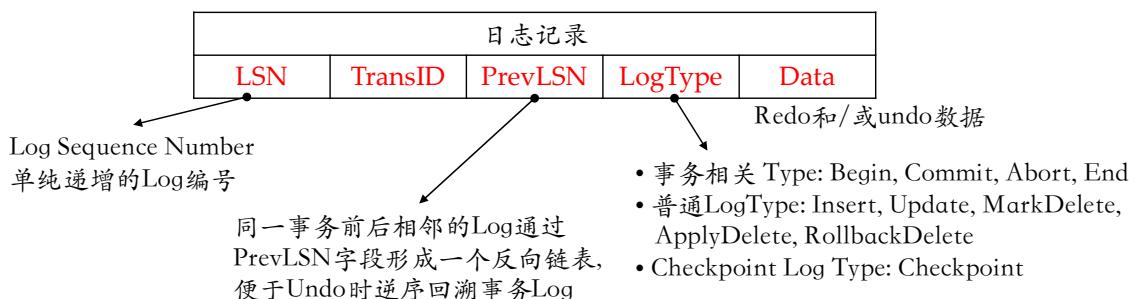


图 10.11: 日志记录结构

补偿日志 CLR(Compensation Log Record)

- ARIES 中的 Logical Undo 操作不具有幂等性, 不可以重复执行;
- CLR 把一条 Logical Undo 日志进行物化, 也即生成一个 Page-oriented 级的 Redo Log, 通过更改页面来达到 Undo 的效果;
- CLR 是 Redo-Only 的, 保证已经 Undo 了的操作不会再被 Undo.
- UndoNext LSN: 只在 CLR 中出现, 用来指示下一条需要 Undo 的 Log 的 LSN, 也即 UndoNxtLSN 是当前日志正在补偿的日志记录的 PrevLSN 值
- 如果 Undo 到一半数据库挂掉后重启, 在重新执行 Undo 时, 只需先取出最后一条 CLR Log 的 UndoNext LSN, 就能继续之前的 Undo 工作.(不会在执行某条 Log 的过程中突然崩溃.)

页面结构:

数据库的每个页都有 PageLSN 域, 该域包含对该页面所做的最近更新日志记录的 LSN.

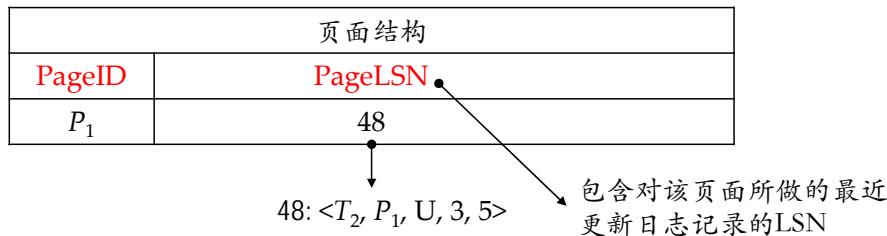


图 10.12: 页面结构

脏页表 DPT:

- Dirty Page Table 记录了所有的 Dirty Page 以及它们的 Rec LSN;
- RecLSN 用于标识引起该页变脏的第一个日志记录的 LSN;
- 当一页被插入到脏页表时, RecLSN 被设置成日志的当前末尾;
- 只要页被写入磁盘, 该页就被从脏页表中移除;
- 从 RecLSN 开始可以保证找到这个页面从硬盘取出后的所有更改 (沿着 Log 链找就可以了).

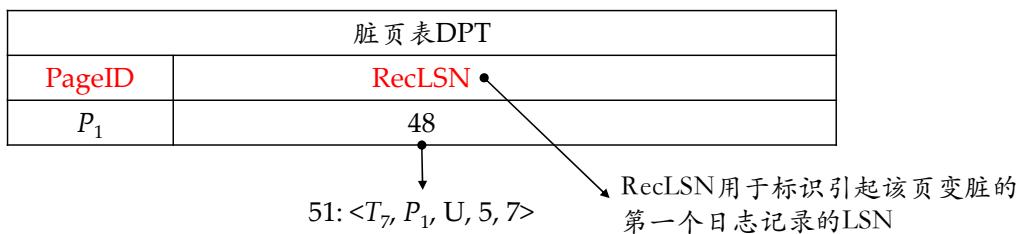


图 10.13: 脏页表结构

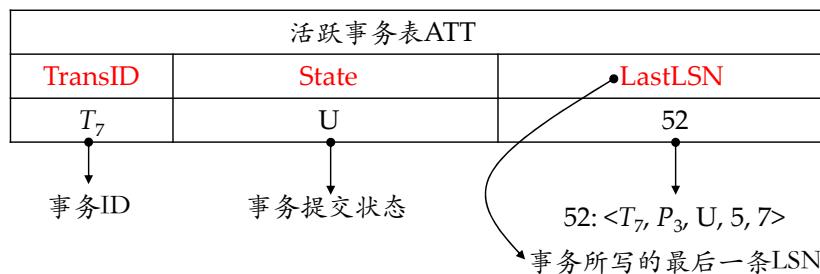
活跃事务列表 (ATT):

图 10.14: 活跃事务列表 (ATT) 结构

10.7.4 ARIES 恢复算法实现过程**ARIES 恢复算法三个原理**

- 先写日志: 在将数据库对象的修改写入磁盘之前, 先将对应的日志记录写入稳存
- 重现历史: 在崩溃后进行重启时, 重做崩溃前的所有操作, 使系统恢复到崩溃时的状态
- 补偿日志: 在回滚事务时, 如果出现对数据库的改变, 为其生成补偿日志, 保证在重复进行重启时不需要重复这些操作

ARIES 算法三个阶段

- 分析阶段: 决定哪些事务要 undo, 哪些页在崩溃时是脏的, 以及 redo 应从哪个 LSN 开始

- Redo 阶段: 从分析过程决定的位置开始, 执行 redo, 重现历史, 将数据库恢复到发生崩溃前的状态
- Undo 阶段: 回滚在发生崩溃时那些不完整的事务

ARIES 算法: 分析过程

- 从 MasterRecord 获得最近一个检查点, 顺序扫描所有 Log, 恢复出 ATT 和 DPT.
- ATT: 对于任一个事务 T_i , 如果:
 - 遇到 T_i 的 Begin Log, 就把 T_i 加入 ATT, 同时把状态设为 Undo Candidate;
 - 遇到 T_i 的 Commit Log, 就把 ATT 中的 T_i 状态设为 Committed;
 - 遇到 T_i 的 End Log, 就把 T_i 从 ATT 中移除;
 - 遇到 T_i 的其他 Log(Redo, Undo, CLR 和 Abort Log), 更新 T_i 的 LastLSN.
- DPT: 遇到任意 Redo Log 或 CLR, 如果对应的 Page
 - 不在 DPT 中, 将它加入 DPT, 同时记录 RecLSN;
 - 已经在 DPT 中, 无需处理.

ARIES 算法: Redo 过程 通过重演所有没有反映在磁盘页上的动作来重复历史

- Redo 过程从 RedoLSN 开始向前扫描日志, 该点之前的所有日志记录已经反映在磁盘数据库页上
- 只要找到一个 update 日志记录, 它就执行如下动作:
 - 如果该页不在脏页表中, 或者该 update 日志记录的 LSN 小于脏页表中该页的 RecLSN, Redo 过程就跳过该日志记录
 - 否则从磁盘调出该页, 如果其 PageLSN 小于该日志记录的 LSN (Log LSN > Page LSN), 重做该日志记录, 修改 PageLSN 为该日志的 LSN.
- redo 阶段为什么要重现历史?

一个事务中的 Delete 操作同时删除了 Table Page 和 Index Page 中的一条数据, 在数据库挂掉前只有 Table Page 被刷盘, 并且这个事务没有提交.

在 Recovery 期间, 如果不 Redo 这个 Delete 操作, 直接做 Logical Undo, 向 Table Page 和 Index Page 各自 Insert 一条数据, 那么 Index Page 中会出现重复的数据

需要通过重现历史来将数据库中的数据恢复到 Operation Consistency 的状态, 保证后续 Logical Undo 的正确执行

ARIES 算法: Undo 过程 Undo 过程反向扫描日志, 取消所有 ATT 中的事务

- 找到 ATT 中最大的 Last LSN, Undo 它对应的事务, Undo 完成后把该事务从 ATT 中移除. 重复上面步骤, 直到 ATT 为空.
- 如果找到一个 CLR, 它用 UndoNextLSN 字段跳过一个已经 Undo 了的事务日志, 否则, 它用事务日志的 PrevLSN 字段查找下一个要被撤消的日志.
- 每当一个 update 日志记录被用于撤消, Undo 过程产生一个包含 undo 执行动作的 CLR, 并将 CLR 的 UndoNextLSN 设置为 update 日志记录的 PreLSN 值.

第十一章 数据库存储

期末考试提纲

- RAID1、RAID5 定义及其特性, 所适用的数据库应用场合
- LSM 树、B+ 树
- 位图索引、按列存储
- 数据库的页结构和行结构

11.1 存储介质

这部分的内容也可以看操作系统的教材《Operating Systems: Three Easy Pieces》[1].

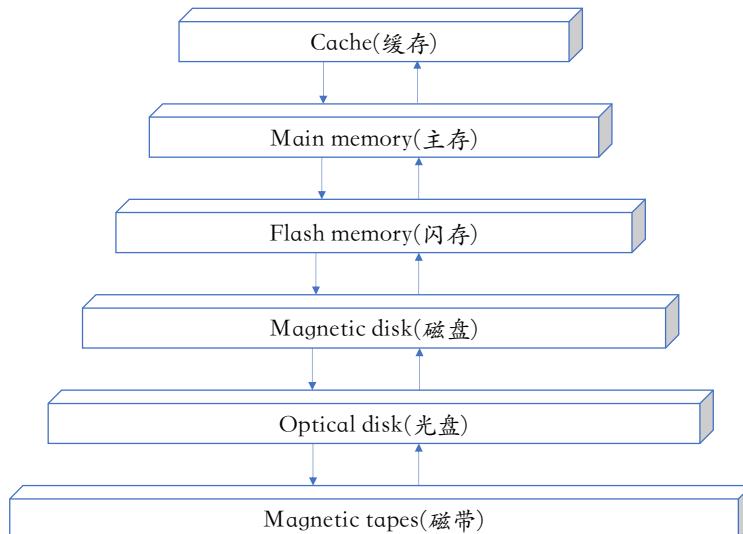


图 11.1: 物理存储介质的层次

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.

- 高速缓冲存储器 (Cache): 最快最昂贵的存储介质; 很小, 由操作系统管理.
- 主存储器 (main memory): 存放可被处理的数据的存储介质; 易失, 相对整个数据库太小.
- 快闪存储器 (flash memory): 读性能类似主存, 写速度非常慢; 电子可擦除可编程只读存储器.
- 光学存储器 (Optical storage): 只读 (CD-ROM)、一次写多次读 (WORM)、多次写 (CD-RW).
- 磁带 (tape): 顺序访问, 归档存储, 容量大, 价格便宜.
- 磁盘存储器 (Magnetic-disk storage):
 - 直接读取设备, 支持随机读取;
 - 非易失联机数据存储设备;
 - 访问数据时, 磁盘 → 内存;
 - 修改后的数据, 内存 → 磁盘.

Hint: 请看一下操作系统里的描述.

磁盘的基本构成:

- 盘片 (platter)、磁道 (track)、扇区 (sector)、柱面 (cylinder)、磁盘臂 (disk arm)
- 读写头 (read-write head): 反转磁性物质磁化方向

- 磁盘控制器 (disk controller):

- 接受读写扇区命令, 定位读写头
- 向扇区写入数据时附加校验和 (checksum), 读取时重新计算校验和

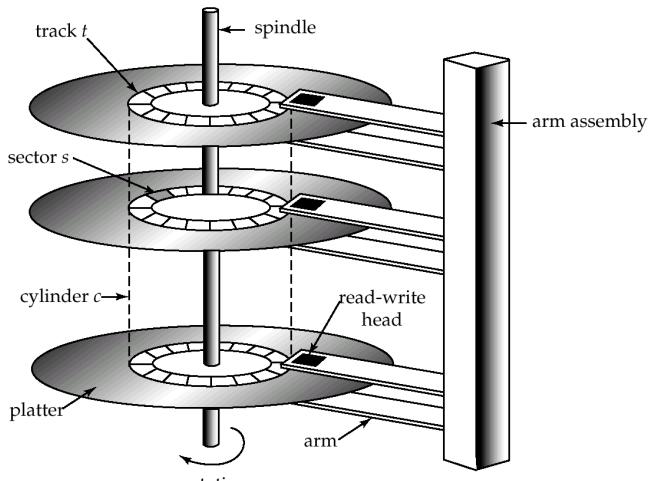


图 11.2: 磁盘的物理结构

磁盘性能度量:

- 访问时间
- 寻道时间 (seek time): 平均寻道时间是最大寻道时间的 $1/3$
- 旋转等待时间 (rotational latency time): 平均旋转时间是旋转一周的 $1/2$
- 数据传输率 (data-transfer rate): 25~100 兆/秒

磁盘访问优化:

- 调度: 电梯算法
- 磁盘块的大小: 小, 更多的磁盘传输次数 vs. 大, 空间浪费
- 文件组织: 按与预期数据访问方式最接近的方式组织磁盘块 + 碎片整理
- 日志磁盘: 顺序写, 消除寻道时间 + 检查点

发掘磁盘顺序读的性能:

- 预读 (prefetch): 利用局部性原理.

11.2 廉价磁盘冗余阵列 (RAID)

廉价磁盘冗余阵列: Redundant Arrays of Inexpensive Disks, RAID. 是一种利用大量廉价磁盘进行磁盘组织的技术.

- 价格: 大量廉价的磁盘比少量昂贵的大磁盘合算得多
- 性能: 大量磁盘可以提高数据的并行存取
- 可靠性: 冗余数据可以存放在多个磁盘上, 单个磁盘的故障不会导致数据丢失

注 过去 RAID 是大而昂贵的磁盘的替代方法, 今天使用 RAID 是因为它的高可靠性和高数据传输率. 因此 “I” 代表 independent, 而非 inexpensive.

定义 11.1 (冗余 (Redundancy))

冗余 (Redundancy): 存储额外信息以便当磁盘故障时能从中重建.



定义 11.2 (镜像冗余)

- 一个逻辑磁盘由两个物理磁盘组成, 写操作在每个磁盘上执行
- 如果其中一个发生故障, 数据可以从另一个磁盘读出
- 只有第一个磁盘故障尚未恢复, 第二个磁盘也发生故障, 这时才会发生数据丢失

**定义 11.3 (校验码冗余)**

纠错码 (Error Correcting Code, ECC): 内存中每个字节都有一个奇偶校验位与之相连, 它记录该字节中为 1 的比特位的总数是偶数 (=0) 还是奇数 (=1), 如果字节中有一位被破坏, 则字节的 ECC 与存储的 ECC 就不会相匹配; 通过 ECC 可以检测到所有的 1 位错误.



通过拆分提高并行:

- 将数据拆分到多个磁盘上以提高传输率
- 通过并行提高性能的两种途径:
 - 负载平衡多个小的存取操作 (即页面存取), 以提高这种存取操作的吞吐量
 - 并行执行大的存取操作, 以减少大的存取操作的响应时间
- 两种不同的拆分方式:
 - 比特级拆分 (Bit-level striping):** 将每个字节按比特分开, 存储到多个磁盘上. 对于由 4 个磁盘组成的阵列, 将每个字节的第 i 个比特位和第 $i + 4$ 个比特位写到第 i 个磁盘上, 其存取速度是单个磁盘的 8 倍.
 - 块级拆分 (Block-level striping):** 对于由 n 个磁盘构成的阵列, 文件的第 i 块存放在第 $(i \bmod n) + 1$ 个磁盘上.

Hint: 下面介绍不同 RAID 级别. 比较重要的是 RAID 1 和 RAID 5.

11.2.1 RAID 0

RAID 0: 块级拆分且没有任何冗余的磁盘阵列, 用于高性能访问且数据丢失不十分重要的应用场合.

11.2.2 RAID 1

带块级拆分的磁盘镜像, 提供最佳写性能, 一般用于类似于数据库系统中日志文件存储的应用场合.



图 11.3: RAID 1: 无冗余拆分

11.2.3 RAID 3: 位交叉奇偶校验

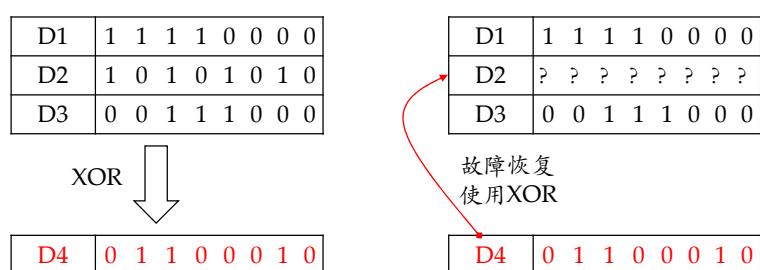


图 11.4: RAID 3: 位交叉奇偶校验

奇偶校验值的计算是以各个硬盘的相对应位进行异或的逻辑运算, 然后将结果写入奇偶校验硬盘.

11.2.4 RAID 4: 块交叉奇偶校验

块级拆分, 在一个独立的磁盘上为其他 N 个磁盘上对应的块保留一个奇偶校验块.

读取一个块只访问一个磁盘, 每个存取操作的传输率低, 但可以并行地执行多个读操作, 从而产生较高的总 I/O 率.

11.2.5 RAID 5: 块交叉的分布奇偶校验

将数据和奇偶校验位分布到所有的 N 个磁盘上.

- 奇偶校验块不能和这个块对应的数据存储在同一个磁盘上 → 一个磁盘坏了就可以立刻重建其数据;
- RAID 5 所有磁盘都参与对读请求的服务, 而 RAID 4 中奇偶校验磁盘不参与读操作;
- RAID 5 包容 RAID 4, 在相同成本下提供更好的读写性能.

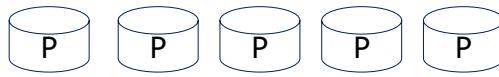


图 11.5: RAID 5: 块交叉的分布奇偶校验

11.2.6 RAID 6

反正很可靠就对了.

11.2.7 RAID 总结

- RAID 0: 高性能, 可靠性差.
- RAID 1: 最佳写性能, 开销大.
- RAID 3: 高数据传输率, 大数据量.
- RAID 5: 高的总 I/O 率, 适合随机读, 大数据量.
- RAID 6: 高可靠性.

11.2.8 选择合适的 RAID 级别

如果应用需要每秒 r 次读, w 次写:

- RAID 1 要求每秒 $r + 2w$ 次 I/O 操作;
- RAID 5 要求每秒 $r + 4w$ 次 I/O 操作;
一次写操作实际上涉及到 4 次 I/O 操作: 读取旧数据、读取旧奇偶校验、写入新数据、写入新奇偶校验.
- RAID 5 比 RAID 1 使用较少的磁盘可能是幻觉. 比如说需要 1TB, 每个磁盘是 1TB. 使用 RAID 1 需要 2 块磁盘, 使用 RAID 5 需要至少 3 块磁盘. (如果只有两块磁盘, 那谁和谁进行异或呢?)
当写操作较少且数据非常大时, RAID 5 较优, 否则 RAID 1 更佳.

RAID 的 Write Back: RAID 控制器先把数据放入自身缓存, 后续再写入.

11.3 缓冲区

为快速找到页面, 内存页面地址被散列 dbid-fileno-pageno 标识 (数据库 ID, 文件号、页面号) 的 hash 地址.
当需要访问一个磁盘块时:

- 如果该块已在缓冲区中, 返回块在内存中的地址;

- 如果块不在缓冲区中,
 - 缓冲区管理器为该块在缓冲区中分配空间, 如果有必要, 替换缓冲区中的其他块
 - 如果被替换的块被修改过, 则将其写回磁盘
 - 将所需块调入缓冲区, 返回其在缓冲区的地址

缓冲区管理中的特殊内存块:

- 被钉住的块 (pinned blocks): 不允许写回磁盘的块.
 - 当一个块上的更新正在进行时, 不允许写回磁盘
 - 可以钉住被频繁访问的小表
- 块的强制刷出 (forced output of blocks):
 - 先写日志原则: 被更新的数据页刷出时, 对应的日志记录被强制刷出
 - 生成检查点时, 日志和数据缓冲区被强制刷出
 - 提交事务时, 其日志记录被强制刷出

替换策略:

- LRU
- MRU(最近最常使用): e.g., 在进行连接 $R \bowtie S$ 时, R 中的一个元组一旦被处理完就不再需要了, 同时 S 中的一个元组要等待其他元组做完, 这个时候就应该使用 MRU.

SQL Server 缓冲区管理: Lazywriter(缓冲池管理器), 使用时钟算法.

...

多大缓冲区是合适的?

- 若一个页面每秒被访问 n 次, 将它驻留在内存节省

$$n \times \frac{\text{price-per-disk-drive}}{\text{accesses-per-second-per-disk}}$$

- 保持一个页面在内存的代价

$$\frac{\text{price-per-MB-of-memory}}{\text{pages-per-MB-of-memory}}$$

- 5-minute rule: 如果一个被随机访问的页面的使用频率超过每 5 分钟一次, 那么它应该被驻留在内存
- 1-minute rule: 如果一个被随机访问的页面的使用频率超过每 1 分钟一次, 那么它应该被驻留在内存

11.4 数据库存储结构

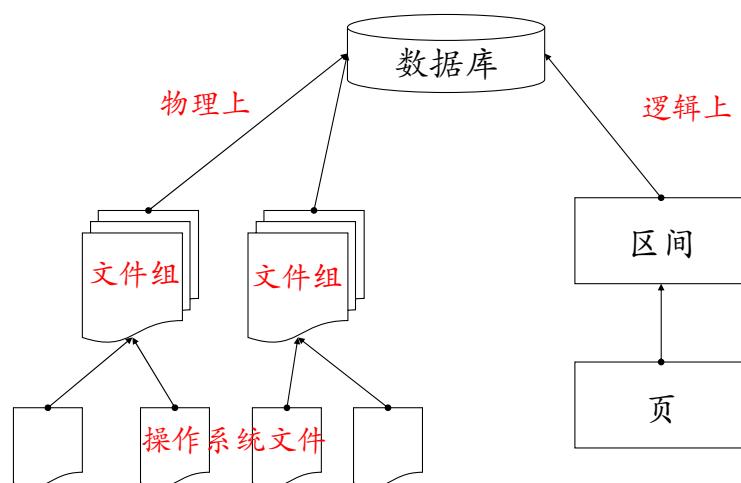


图 11.6: 数据库存储结构

11.4.1 页面与区间

- 数据文件被划分成 8k 的页面;
 - 每个文件中的页面号都以 0 开始;
 - 页面号的形式为 (file#:page#), 如 (3:124);
 - 8 个连续页面构成一个区间——64k, 总是从能被 8 整除的页面开始;
 - 存储分配总是按照区间为单位进行, 对象每次增长 1 个区间;
 - I/O 可以按页面 (8 KB) 或者区间 (64 KB) 来进行.
- 如何为对象分配空间?
- GAM
 - PFS
- 如何找到对象占据的空间?
- IAM
- 行在页内是如何存储的?

11.4.2 GAM - 全局分配位图 (bitmap)

- 记录文件当中哪些区间已经被分配的页面, 可以看成是一个 8000 个字节的位图, 每个位代表一个区间;
- 位 0 代表区间 0, 位 1 代表区间 8, 位 2 代表区间 16
- 0: 被使用; 1: 未使用;
- 差不多 64000 位, 所以可以表示 64000 个区间;
- 表达 4GB 数据空间, 如果文件大于 4GB, 增加新的 GAM 页.

11.4.3 PFS - 空闲页空间

- 记录文件中每个页面是否已经被分配以及有多少空闲空间;
- 每一个页面在 PFS 页中有一个字节对应
- 每个 PFS 覆盖 8088 个连续页面 (64 MB)
- 页面充满度: 0, 1-50%, 51-80%, 81-95%, 96-100%
- 第一个 PFS 位于文件的第二个页面 (page1), 以后每 8088 都是一个 PFS 页

11.4.4 IAM - 索引分配位图

如何发现一个特定对象的区间或页面?

- 每个表/索引都至少有一个 IAM, 记录该对象拥有哪些区间;
- IAM 覆盖的范围与 GAM 相同, 如果位为 1, 说明该区间被分配给该对象, 如果位为 0, 说明该区间未被分配给该对象;
- 如 11000000, 说明第一、二个区间被分配给该对象;
- 一个对象每占据文件的 4G 空间, 就需要一个 IAM, 对象的所有 IAM 构成一个双向链表.

11.4.5 基本页结构

所有页面包括 **页面头**、**页面体**、**页面槽**.

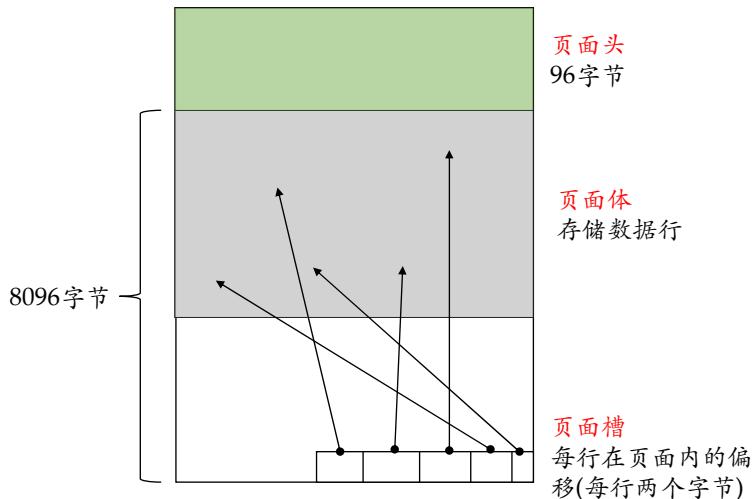


图 11.7: 基本页结构

页头字段含义:

Field	What It Contains
pageID	数据库中该页的文件号和页号
nextPage	在页链中该页的下一页的文件号和页号
prevPage	在页链中该页的上一页的文件号和页号
objID	该页所属对象的 ID
lsn	用于修改该页的日志序列号 (LSN)
slotCnt	该页上槽的总数
level	该页在索引中的级别 (对于叶级别该值为 0)
indexId	该页的索引 ID (对于数据页该值为 0)
freeData	该页第一个空闲空间的字节偏移量
pminlen	行的固定长度部分的字节数
freeCnt	该页上空闲字节的数目
reservedCnt	由所有事务保留的字节数
xactreserved	由最近启动的事务保留的字节数
tornBits	每个扇区 1 位, 用于监测页分裂的写
flagBits	一个两字节的位置, 包含有关该页的额外信息

数据行结构:

行结构	状态位A	状态位B	行定长 部分大小	定长 数据	列数	NULL 位图	变长 列数	可变列偏移 数组	变长 数据
助记符	TagA	TagB	Fsize	Fdata	Ncol	Nullbits	VarCount	VarOffset	VarData
大小	1字节	1字节	2字节	Fsize-4	2字节	[Ncol/8]	2字节	2*VarCount	

$$\text{VarOff}[\text{VarCount}] - (\text{Fsize} + 4 + \text{Ceiling}(\text{Ncol} / 8) + 2 * \text{VarCount})$$

图 11.8: 数据行结构

11.5 索引

11.5.1 B+ 树索引

我们首先来看叶节点:

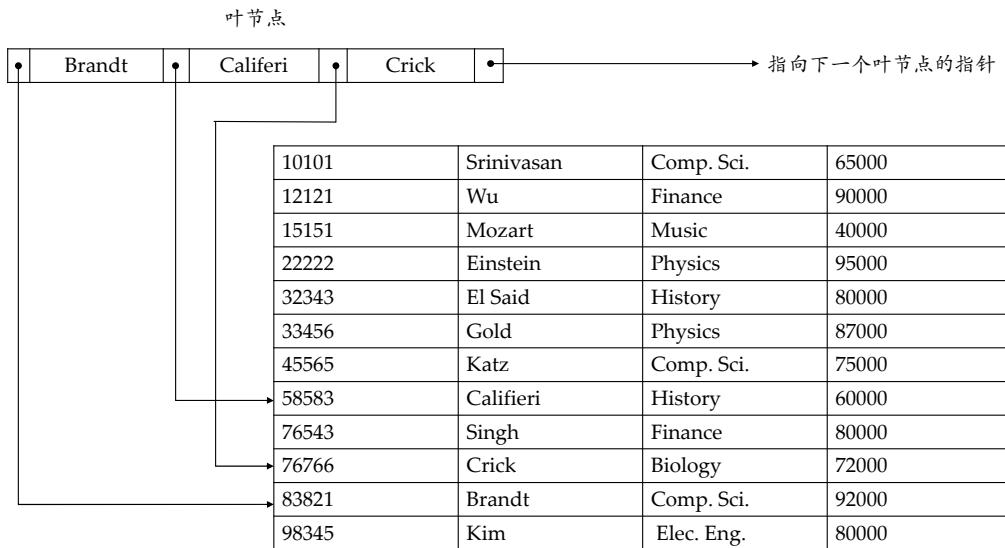


图 11.9: B+ 树的叶节点

如果此 B+ 树的阶数是 m , 则除了根之外的每个节点都包含最少 $\lfloor m/2 \rfloor$ 个元素最多 $m - 1$ 个元素, 对于任意的结点有最多 m 个子指针. 对于所有内部节点, 子指针的数目总是比元素的数目多一个. 所有叶子都在相同的高度上, 叶结点本身按关键字大小从小到大链接.

除了根之外每个非叶子节点有 $\lceil n/2 \rceil \sim n$ 个子女. 有 n 个子女则含有 $n - 1$ 个码.

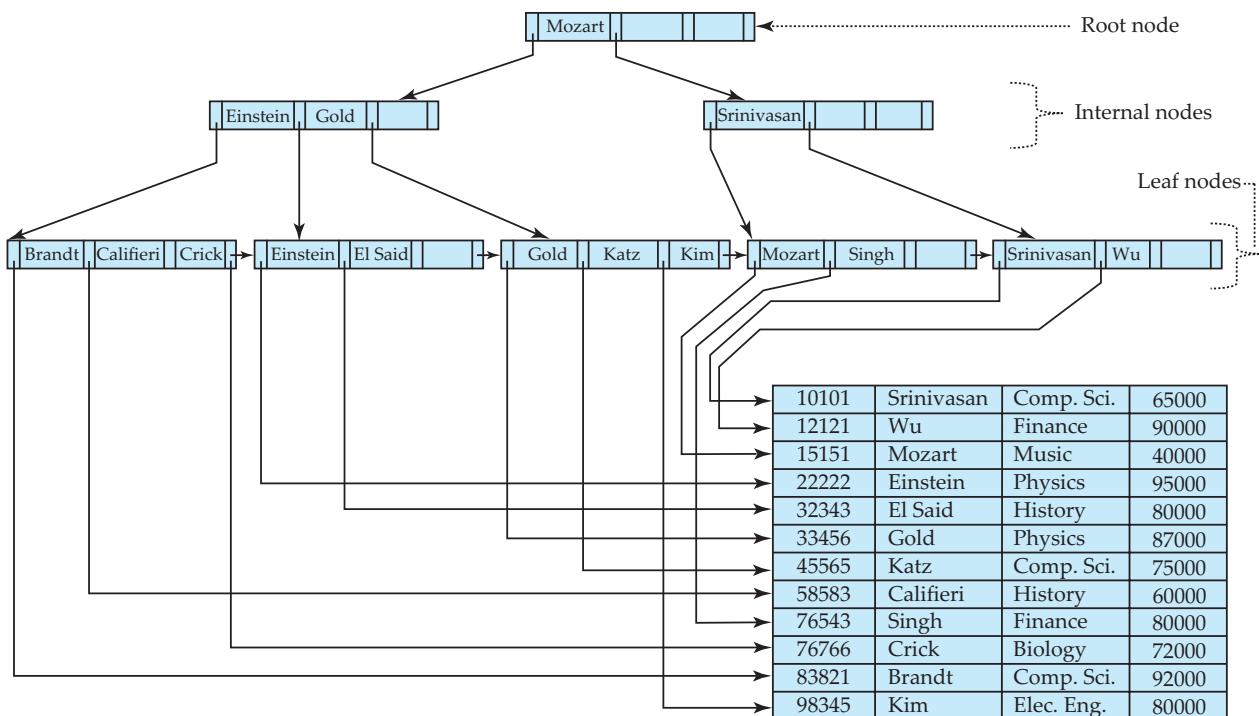


图 11.10: $n = 4$ 的时候的 B+ 树

在查找时, 若非叶子节点上的关键字等于给定值, 并不终止, 而是继续向下直到叶子节点. 因此, 在 B+ 树中, 不管查找成功与否, 每次查找都是走了一条从根到叶子节点的路径.

B+ 树的插入算法:

1. 若为空树, 创建一个叶子节点, 然后将记录插入其中, 此时这个叶子节点也是根节点, 插入操作结束.
2. 根据关键字找到叶子节点, 向这个叶子节点插入记录. 插入后, 若当前节点关键字的个数小于 m , 则插入结束. 否则将这个叶子节点分裂成左右两个叶子节点, 左叶子节点包含前 $m/2$ 个记录, 右节点包含剩下的记录, 将第 $m/2 + 1$ 个记录的关键字进位到父节点中(父节点一定是索引类型节点), 进位到父节点的关键字左孩子指向左节点, 右孩子指向右节点. 将当前节点的指针指向父节点, 然后执行第 3 步.
3. 针对索引类型节点(内部节点): 若当前节点关键字的个数小于等于 $m - 1$, 则插入结束, 否则, 将这个索引类型节点分裂成两个索引节点, 左索引节点包含前 $(m - 1)/2$ 个 key, 右节点包含 $m - (m - 1)/2$ 个 key, 将第 $m/2$ 个关键字进位到父节点中, 进位到父节点的关键字左孩子指向左节点, 进位到父节点的关键字右孩子指向右节点. 将当前节点的指针指向父节点, 然后重复这一步.

B+ 树的删除算法:

1. 首先查询到键值所在的叶子节点, 删除该叶子节点的数据.
2. 如果删除叶子节点之后的数据数量, 满足: 关键字的个数大于等于 $\lceil m/2 \rceil$, 则直接返回.
3. 否则, 就需要做平衡操作: 如果该叶子节点的左右兄弟节点的数据量可以借用(选择多的那个), 就借用过来满足平衡条件. 否则, 就与相邻的兄弟节点合并成一个新的子节点了.

B+ 树矮而胖: 假定树结点大小与磁盘块大小相等, 为 $4k$, $n = 100$, 有 1 百万个搜索码, $\log_{50}(1000000) = 4$, 访问 4 个磁盘结点.

二叉树瘦而高.

InnoDB 一颗 B+ 树可以放多少行?

例题 11.1 假定一个页 16K, 一行 1K, 一页存放 16 行.

假定主码为 bigint, 长度 8 字节, 指针 6 字节, 共 14 字节. 一个页中能存放 $16384/14=1170$.

一棵高度为 2 的 B+ 树, 能存放 $1170*16=18720$ 条这样的记录.

一棵高度为 2 的 B+ 树, 能存放 $1170*1170*16=21902400$ 条这样的记录.

11.5.2 散列索引

- 桶(bucket): 存储一条或多条记录的存储单元
- 散列(hash): 散列函数 $h(K) = B$, K 是搜索码, B 是桶地址
 - 分布是均匀的, 桶包含记录的个数是均匀的
 - 分布是随机的, 散列值不能与搜索码值呈现出相关性
- 桶溢出(bucket overflow): 因桶不足(insufficient bucket) 或偏斜(skew) 所致

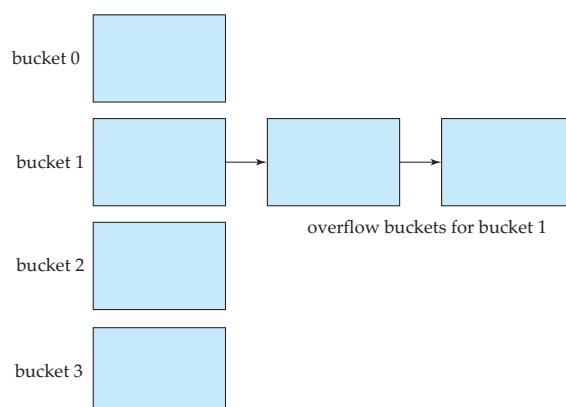


图 11.11: 溢出链

桶溢出的时候，在后面加上一条链。

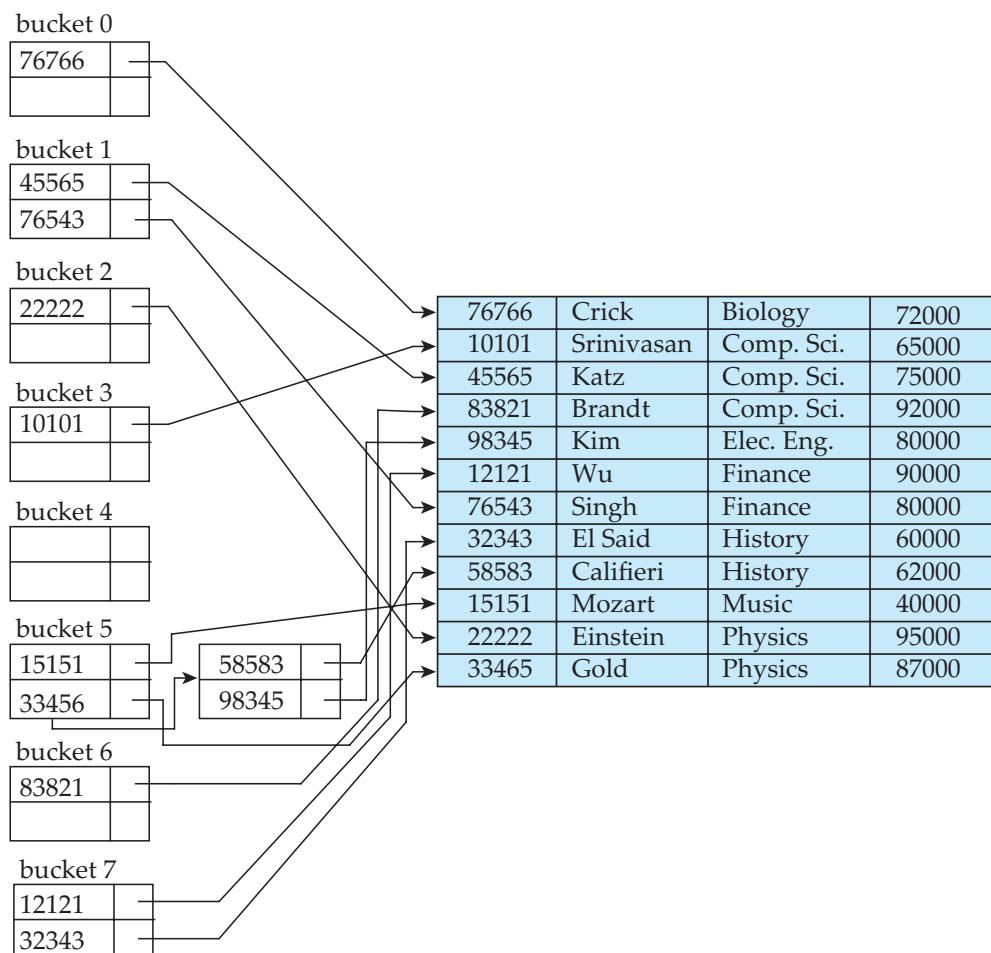


图 11.12: 散列索引

Listing 11.1: MySQL 中的散列索引

```
CREATE TABLE testhash (
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    KEY USING HASH(fname)
) ENGINE=MEMORY;
```

B+ 树索引的索引：当对某个页面访问次数满足一定条件时会将页面地址存于 Hash 表，下次查询时不需要 B+ 树那样从根节点到叶子节点逐级查找，只需一次哈希算法即可立刻定位到相应的位置。

11.5.3 位图索引

- 针对一些特殊的列建立索引；
- 列中的每一个值对应一个向量中的一位；
- 向量的长度对应于记录的条数；
- 不适合列中值的个数太多的情况。

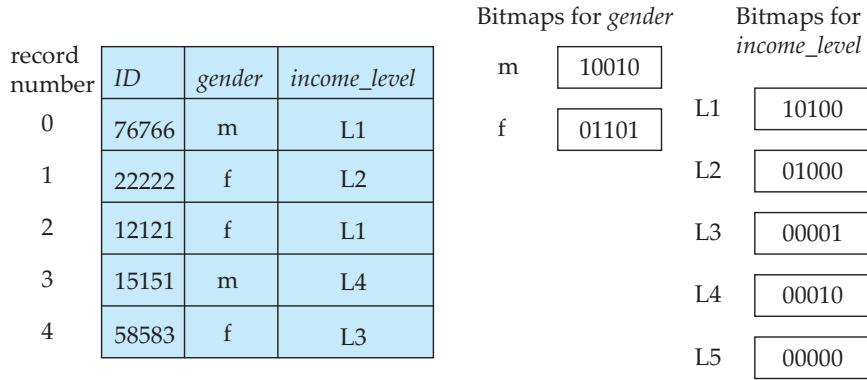


图 11.13: 位图索引

现在我们考虑下面的查询:

```
select *
from instructor_info
where gender = 'f' and income_level = 'L2';
```

那么我们可以计算出 *gender* 值为 'f' 的位图和 *income_level* 值为 L2 的位图的交 (intersection) 就能算出最终需要查询的记录.

11.5.3.1 位片索引 (Bit-sliced Index)

位片索引是将属性列的域值按照某种方式进行垂直分割, 然后以二进制位图的形式存储.

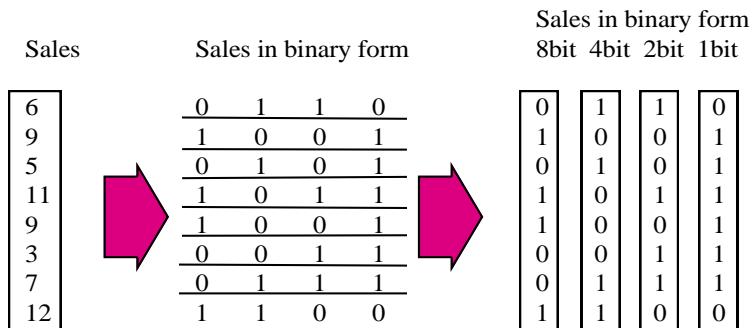


图 11.14: 位片索引

11.5.3.2 关系表的行式存储和列式存储

行式存储的特点及其适合的场合:

- 一条记录的所有字段存储在一起
- 各个字段的异质性导致压缩效果差
- 逻辑上访问单个字段, 物理上会把同一记录的其他字段一并返回
- 适合取回一条完整记录的场合 (OLTP)
- 不适合针对单列的聚合分析 (OLAP)

列式存储的特点:

- 提高带宽利用率
- 提高数据压缩率: 将同一个属性域的数据存储在一起, 提高了局部性以及压缩比率. **宽表 + 稀疏表**.
- 增加插入操作的代价.

- 增加了磁盘寻道时间.

- 增加重构元组的代价.

列式存储的实现: MonetDB.

列式存储的实现: C-Store[8] & Vertica[2].

- 表被拆分成 projections → Sales 表被拆分成两个 projections;

- 第一个 projection 按照 date 排序, 按照 HASH(sale_id) 分段;

- 第二个 projection 按照 cust 排序, 按照 HASH(cust) 分段.

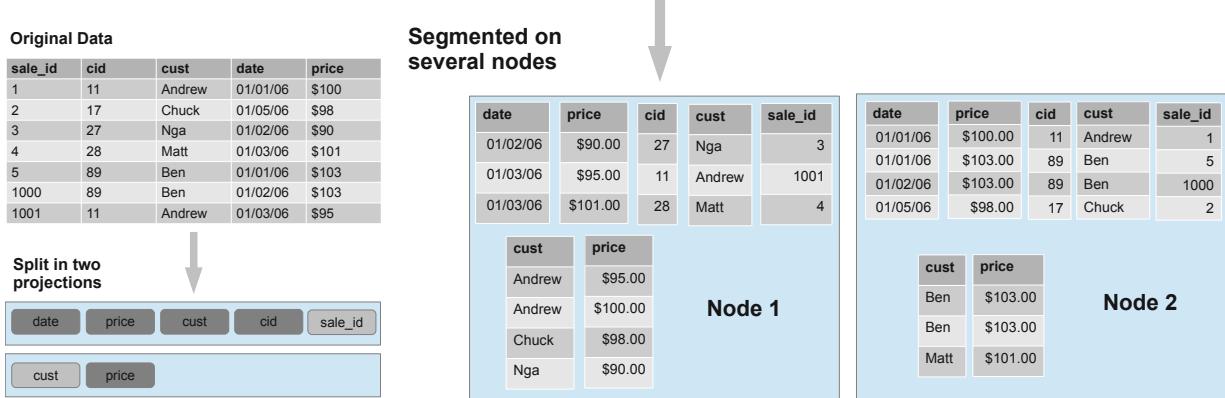


图 11.15: Vertica 的实现

稀疏属性可以压缩存储.

11.5.4 多维索引

多维索引的主要目的是为了加速对多维数据的检索速度, 例如范围查询 (Range Query) 和最近邻查询 (Nearest Neighbor Query). 根据教材里的说法, 这是关于空间数据 (Spatial Data) 的存储的.

- Nearest neighbor queries, given a point or an object, find the nearest object that satisfies given conditions.
- Range queries deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.
- Queries that compute intersections or unions of regions.
- Spatial join of two spatial relations with the location playing the role of join attribute.

11.5.4.1 k-d tree

早期用于多维索引的结构之一: k-d tree.

一棵 k-d 树的每一层都把空间分成两个部分, 在树顶层的节点处分区是沿着一个维度进行的, 在下一层节点中则沿着另一个维度进行. 分区的进行方式是这样的: 在每个节点处, 存储在子树中的大约一半的点落在一边且另一半落在另一边, 当一个节点的点数少于给定的最大点数时, 分区停止.

下面我们把叶节点中的最大点数被设置为 1, 其中每条线对应于 k-d 树的一个节点, 线的编号表示相应节点出现在树中的层级.

比如说, 一个范围查询可能查找所有在 x 维度上是 50 到 80, y 维度是 40 到 70. 可以通过以下递归过程来执行范围搜索:

1. 假设节点是一个内部节点, 令它在点 x_i 处按一个特定维度 (比如说 x) 被拆分. 左子树中的项所具有的 x 值 $< x_i$, 且右子树中的项所具有的 x 值 $\geq x_i$. 如果查询范围包含 x_i 则在两个孩子上递归地执行搜索. 如果查询范围在 x_i 的左侧, 则只对左孩子执行递归搜索, 否则只在右子树上执行搜索.
2. 如果节点是叶节点, 则检索查询范围内包含的所有项.

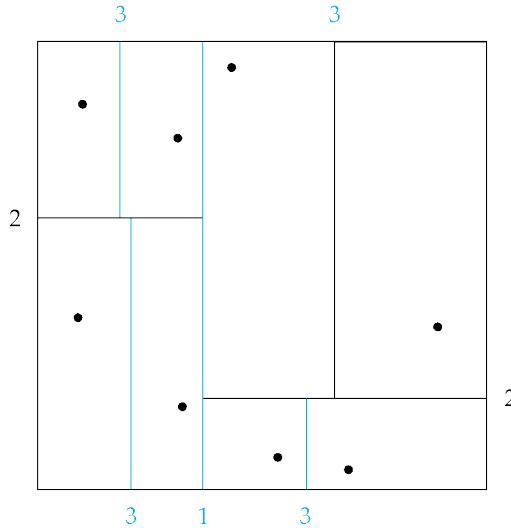


图 11.16: k-d 树的例子

11.5.4.2 四叉树

四叉树 (Quadtrees):

1. 每个非叶子节点将其区域划分为四个大小相等的象限;
2. 每个这样的节点都有四个子节点, 分别对应于这四个象限, 依此类推;
3. 叶子节点的点数介于零到某个固定的最大点数之间 (示例中设置为 1).

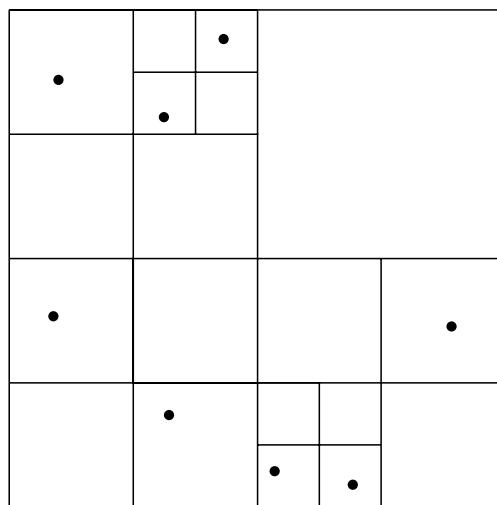


图 11.17: 四叉树的例子

11.5.4.3 R-tree

一种称为 R 树 (R-tree) 的存储结构对于跨越空间区域的对象 (比如线段、矩形和其他多边形) 以及点进行索引是有用的. R 树是一种平衡树结构, 索引对象存储在叶节点中, 这很像 B+ 树. 但是, 与每个树节点关联的不是值的范围, 而是矩形边框 (bounding box). 叶节点的边框是与包含叶节点中存储的所有对象的轴平行的最小矩形. 类似地, 内部节点的边框是与包含其孩子节点边框的轴平行的最小矩形, 一个对象 (比如多边形) 的边框被类似地定义为与包含该对象的轴平行的最小矩形.

每个内部节点存储孩子节点的边框以及指向孩子节点的指针. 每个叶节点存储被索引的对象. 下面的例子中边框 i 的坐标被标记为 BB_i .

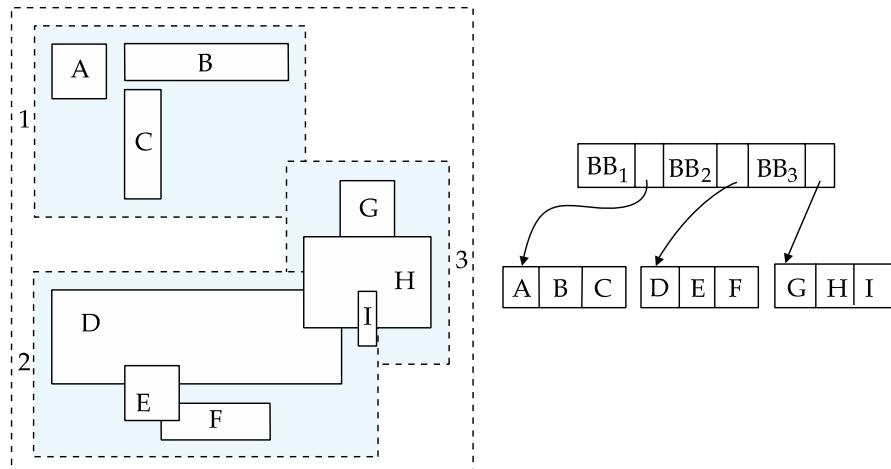


图 11.18: R 树的例子

11.5.5 LSM 树

LSM 树 (Log-structured merge tree): 写优化的索引结构. [6]

- 磁盘的顺序读性能远胜于随机读;
- B+ 树的读性能: 磁盘顺序读;
- B+ 树的写性能: 磁盘随机写.

11.5.5.1 LSM 树的设计思想

将 N 个数据划分成多个小的有序结构, 每 m 个数据在内存里排序一次, 这样就获得 N/m 个有序结构; 查询时从最新的一个有序结构里做二分查找, 如果没找到就继续查找下一个有序结构; 读取的时间复杂度是 $N/m \times \log_2 N$

随着小树越来越大, 内存中的小树会 flush 到磁盘中, 磁盘中的树定期做 merge 操作, 合并成一棵大树, 以优化读性能.

合并时, 不会像 B+ 树一样, 在原数据的位置上修改, 而是直接插入新的数据, 从而避免了随机写.

LSM-Tree 属于传输型, 因为它会使用日志文件和一个内存存储结构把随机写操作转化为顺序写.

11.5.5.2 LSM 树的增删查改

LSM 树结构横跨内存和磁盘, 包括:

- memtable
- immutable memtable
- SSTable

写入流程:

1. 来了一个 $\text{put}(k, v)$ 操作, 首先追加到 WAL 日志中, 接下来加到 C0 层
2. 当 C0 层的数据达到一定大小, 就把 C0 层和 C1 层合并
3. 合并出来的新的 new-C1 会顺序写磁盘, 替换掉原来的 old-C1
4. 合并之后所有旧文件都可以删掉

删除操作:

1. 当有删除操作时, 并不需要像 B 树一样, 在磁盘中找到相应的数据后再删除, 只需要在 memtable 中插入一条数据当作标志, 如 $\text{delKey}:1933$;
2. 当读操作读到 memtable 中的这个标志时, 就会知道这个 key 已被删除;

3. 在随后的**日志合并**中, 这条被删除的数据会在合并过程中一起被删除.

更新操作:

1. 和删除操作类似, 都是只操作 memtable, 写入一个标志, 随后真正的更新操作被延迟在合并时一并完成.

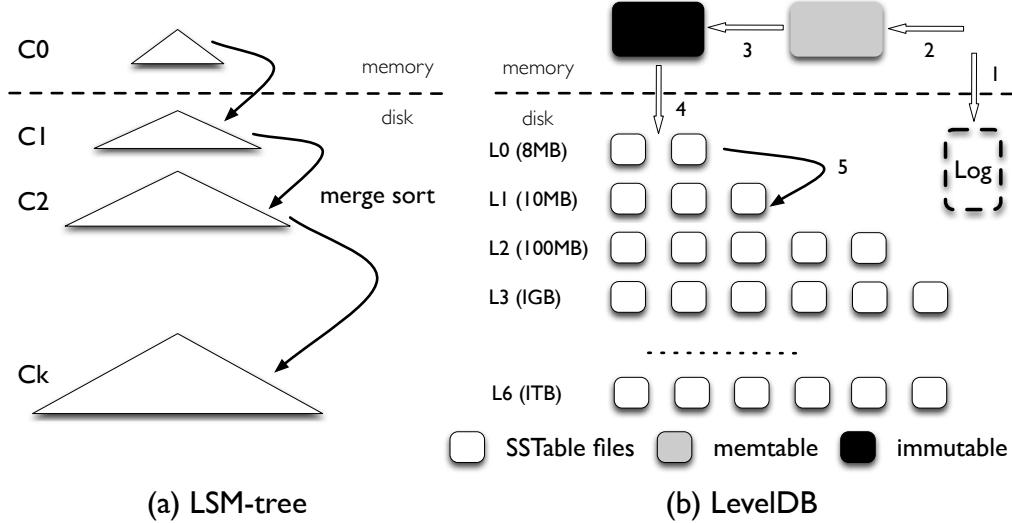


图 11.19: LSM 树的结构

查询流程:

1. 读操作需要依次读取 memtable、immutable memtable、SSTable0、SSTable1...
2. 需要反序遍历所有的集合, 序号小的集合中的数据一定会比序号大的集合中的数据新
3. 在这个反序遍历过程中一旦匹配到要读取的数据, 则一定是最新的数据, 返回该数据
4. 对于不存在的数据, 则会白白遍历所有集合: 引入布隆过滤器 (Bloom Filter)¹来加速读操作, 当它显示相应的 SSTable 中没有要读取的数据时, 就跳过该 SSTable.

Bloom Filter: 其中有 m 个位, 使用 k 个 hash functions h_1, h_2, \dots, h_k , 映射后的值范围为 $\{1, \dots, m\}$.

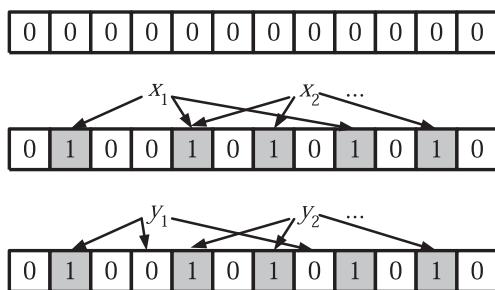


图 11.20: Bloom Filter 的一个例子

如图, 我们先向集合里插入了 x_1 和 x_2 , 分别 hash 了 k 次, 对于得到的每个位置都设置为 1.

接着, 我们试图查询 y_1 是否位于集合之中, 分别 hash k 次, 发现有位置是 0, 这就意味着 y_1 肯定不在集合中(利用反证法, 假设在, 那么存在一个 x_t 使得 $h_i(y_1) = h_i(x_t), i = 1, 2, \dots, k$, 这就意味着 hash 得到的位置应该都是 1, 现在却反而有 0, 矛盾!); 但是现在查询 y_2 , 发现 hash 的位置都是 1 了, 这就说明有可能在集合中, 也有可能不在.

¹操作系统的期末考试论文题也考到了.

11.5.5.3 LSM-tree 读写放大

读写放大 (read and write amplification) 是 LSM-tree 的主要问题:

- 读写放大 = 磁盘实际读写的数据量 / 用户访问的数据量
- 以 Level Style Compaction 机制为例, 它每次拿上一层的所有文件和下一层合并, 下一层大小是上一层的 r 倍, 单次合并的写放大就是 r 倍;
- 假如现在有三层, 文件大小分别是: 9, 90, 900, $r = 10$. 又写了个 1, 这时就会不断合并, $1 + 9 = 10$, $10 + 90 = 100$, $100 + 900 = 1000$, 总共写了 $10 + 100 + 1000$.
- **读写放大 = $10/1 + 100/10 + 1000/100 = 30 = r \times level$.**
key-value 越小读放大越大.
- 为了查询一个 1KB 的数据, 最坏需要读 L0 层的 8 个文件, 再读 L1 到 L6 的每一个文件, 一共 14 个文件;
- 而每个文件内部需要读 16KB 的索引, 4KB 的布隆过滤器, 4KB 的数据块, 一共 $24 * 14 / 1 = 336$ 倍.

第十二章 查询处理

期末考试提纲

- 三种连接算法的实现策略及各自适用场合

12.1 查询处理流程

定义 12.1 (查询处理)

查询处理是指从数据库中提取数据的一系列活动，主要包括：

- 将用高层数据库语言表示的查询语句翻译为能在文件系统这一物理层次上实现的表达式；
- 为优化查询而进行各种转换；
- 查询的实际执行。

实际上就是实现一个

- 输入：SQL 语句
 - 输出：满足查询条件的数据
- 的过程。



12.1.1 查询处理步骤

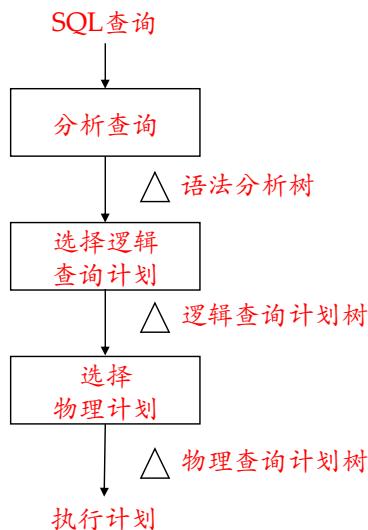


图 12.1: 查询处理步骤

12.1.2 查询优化

定义 12.2 (查询优化)

查询优化：对一个给定的查询，从多种可能执行方式中选择出最有效率的作为查询执行计划。



高效的执行计划有两方面的衡量标准：

- 使中间结果最小化
- 采用尽可能好的操作算法

定义 12.3 (基于代价的方法 (cost-based))

基于代价的方法 (cost-based):

- 生成所有可能的候选计划, 通过统计信息和存取路径确定每一计划中各个操作的代价和综合代价, 最后选择代价最小的一个;
- 优化代价较大.



定义 12.4 (启发式方法 (rule-based))

启发式方法 (rule-based):

- 主要是代数优化, 依据关系代数的等价变换做一些逻辑变换;
- 假定因素太多, 优化的代价虽小, 但优化的准确程度较差.



两种方法结合起来使用: 利用启发式方法尽量减少候选计划, 利用基于代价的方法准确地确定执行计划.

关系代数表达式的优化:

利用关系代数表达式的等价规则: 各种结合律、分配律、交换律

- 将选择运算尽可能移向叶节点;
- 将选择运算尽可能移向叶节点;
- 合并可能的投影操作.

查询代价:

- 查询处理对各种资源的使用情况
 - 磁盘存取 (简化为磁盘块传送数)
 - CPU 时间
 - 通信开销
- 一个重要的影响因素: 主存中缓冲区的大小 M
 - 最好的情形, 所有的数据可以读入到缓冲区中
 - 最坏的情形, 缓冲区只能容纳数目不多的数据块

12.1.3 代价估算

访问方法	估计代价 (逻辑读的数量)
表扫描	表所占用的全部数据页的数量
聚簇索引	索引的高度 + 要扫描的页的数量 (要扫描的页的数量 = 满足条件的行的数量/每个数据页所包含的行的数量)
堆上的非聚簇索引	索引的高度 + 叶结点所指的页面数 + 满足条件的行的数量 (可能所有的行均不在同一页面中, 所以要得到一行就必须进行一次逻辑读)。由于相同的数据页经常要读取多次 (从缓冲区中读取), 所以这样逻辑读的数量将比表的页面数要多
聚簇索引上的非聚簇索引	索引的高度 + 叶结点所指的页面数 + (满足条件的行的数量 × 搜索一个聚簇索引码的代价)
非聚簇覆盖索引	索引的高度 + 索引的叶结点所占用的索引页 (满足条件的行数/每个叶页面包含的行数)

表 12.1: 不同访问方法的估计代价

12.1.4 统计信息

关系 R 的统计信息:

- $T(R)$: R 中的元组数
- $S(R)$: R 中每个元组的字节数
- $B(R)$: R 中所有元组所占据的块数
- $V(R, A)$: R 的属性 A 上的唯一值的个数
- $f(R)$: 每个块上的 R 中元组的最大数目

索引 I 的统计信息:

- $HT(i)$: 索引的层数
- $LB(i)$: 索引叶结点的块数

数据库系统通常会维护一些系统表或数据字典, 用来存储与数据库对象 (如表、索引等) 相关的统计信息, 这就是Statistics 字典表, 使用 ANALYZE TABLE 命令触发.

Listing 12.1: 基于规则的优化示例

```
create table test (col1 unique, col2)
select col1, sum(col2)
group by col1
```

上面的代码中, 由于 $col1$ 是 unique 的, 所以系统并不执行分组操作.

12.2 执行计划

12.2.1 数据查找

堆上的表:

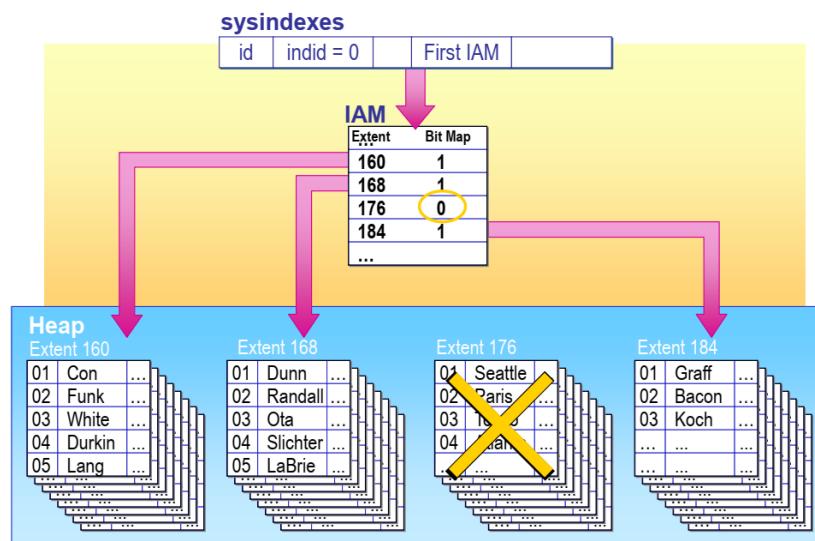
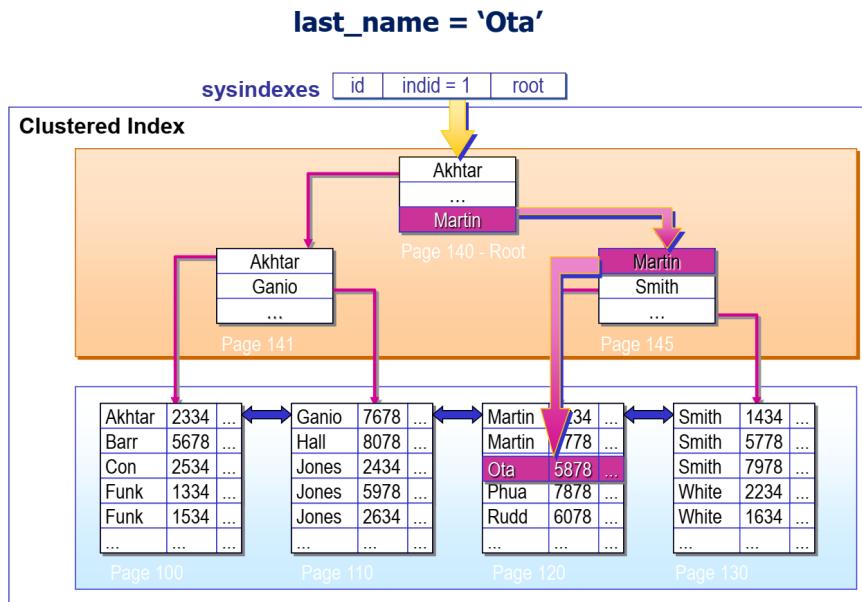


图 12.2: 堆上的表

聚簇索引 (Clustered Index) 是指数据库表中数据行的物理存储顺序与某一索引的键值顺序一致, 也就是说, 聚簇索引决定了表中数据的物理排列方式. 因为数据在磁盘上的存储顺序是按照聚簇索引的键值排序的, 所以一个表只能有一个聚簇索引.



堆上的非聚簇索引表结构

聚簇索引表上的非聚簇索引

"Unimportant. There's a million things that I haven't done."

12.2.2 各种访问方法

对于关系 $R(A, B, C, P)$:

- 表扫描: `select * from R`. 整张表的扫描, 开销是 100%.
- 无序聚簇索引扫描: `select * from R create clustered index cidx1 on R(P)`. 虽然按照一定顺序扫描, 但是依然是整张表的扫描, 开销是 100%.
- 有序聚簇索引扫描
- 无序覆盖非聚簇索引扫描
- 有序覆盖非聚簇索引扫描
- 非聚簇索引查找 + 有序局部扫描 + Lookups
- 索引交集

"Unimportant. There's a million things that I haven't done."

12.3 算子实现

每一个基本的代数运算都有多种不同的实现算法, 适用于不同的情况

- 等值条件, 范围条件
- 数据是聚集的, 数据是非聚集的
- 相关属性上有索引, 相关属性上没有索引
- 执行代价不同

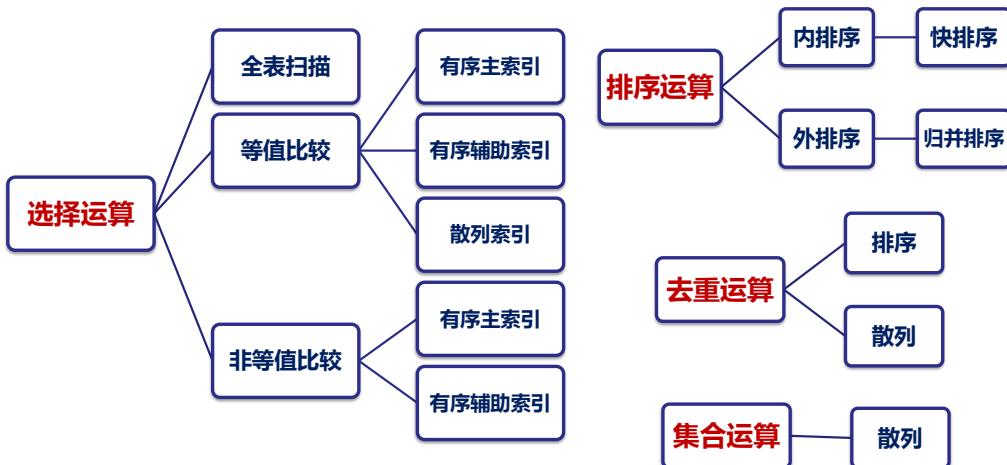


图 12.3: 实现各种逻辑运算的物理算子概览

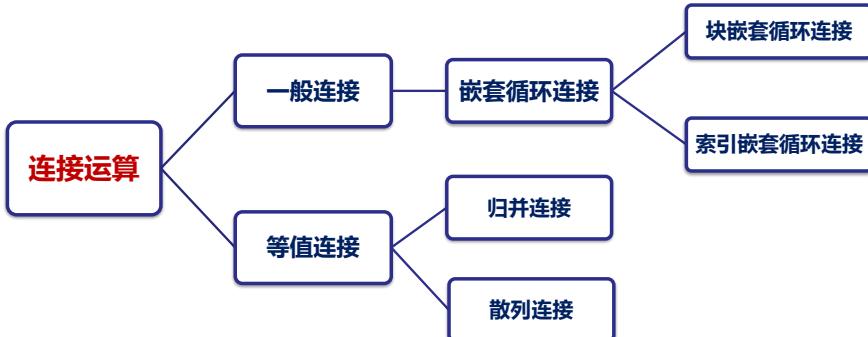


图 12.4: 实现各种逻辑运算的物理算子概览

12.3.1 选择运算

全表扫描:

- 方法: 依次访问表的每一个块, 对于每一个元组, 测试它是否满足选择条件
- 代价: $B(R)$, 如果是码查找, 平均代价为 $B(R)/2$, 因为主码只有一个, 平均下来只要查找一半的长度
- 缺点: 效率低
- 优点: 对关系的存储方式没有要求, 无需索引
- 适用于任何选择条件

等值比较选择 $\sigma_{A=v}(r)$:

- 利用有序主索引.
 - 在 B+ 树索引中找到相应的叶结点, 其中包含指向满足该等值条件的记录的一个或多个指针. (满足条件的记录在一个块或相邻的多个块中)
 - 代价: I/O 操作次数等于 B+ 树的高度, 加上包含具有搜索码值的记录的磁盘块数
- 利用有序辅助索引.
 - 在 B+ 树索引中找到相应的叶结点, 其中包含指向满足该等值条件的记录的一个或多个指针. (满足条件的记录在不相邻的多个块中)
 - 代价: I/O 操作次数等于 B+ 树的高度, 加上检索到的记录数目
- 散列索引.
 - 通过计算散列函数找到相应的桶, 在桶中检索到满足该等值条件的一条或多条记录
 - 代价: I/O 操作次数等于到桶的定位, 加上存储桶所包含的块数

非等值比较选择 $\sigma_{A>v}(r)$:

- 利用有序主索引.
 - 在索引中查找 v 值以检索出满足条件 $A = v$ 的首条记录. 从该元组开始到文件尾进行文件扫描返回所有满足 $A > v$ 条件的元组.
 - 代价: I/O 操作次数等于 B+ 树的高度, 加上包含满足搜索条件的记录的磁盘块数
- 利用有序辅助索引.
 - 在索引中查找 v 值以检索出满足条件 $A = v$ 的首条记录. 从该索引条目直至最大值索引条目提供了指向满足条件的记录的指针, 根据指针逐个取实际记录
 - 代价: I/O 操作次数等于 B+ 树的高度, 加上包含满足搜索条件的记录的磁盘块数

12.3.2 排序

排序场合:

- SQL 查询可以指定对输出进行排序
- 关系运算的某些操作, 如连接运算, 排序后实现高效
对于可放进内存的关系, 使用如快排序之类的技术.
- 对不能放进内存的关系, **使用外排序**.
- 外排序: 创建有序段 + N 路归并
- 所有的输入数据最初分成许多有序的归并段文件, 然后不断归并成许多更大的归并段文件, 直到剩下一个文件为止

12.3.3 集合运算

相同的散列函数对两个关系进行划分, 由此得到各个划分, 对关系 r 的一个划分建立内存中的 hash 索引, 然后针对每一个划分进行以下的步骤:

1. 并: 把 s 中相应划分中的元组加入以上的 hash 索引中, 条件是元组不在 hash 索引中, 把 hash 索引中的元组加入到结果中
2. 交: 对 s 中相应划分中的每一个元组, 检索 hash 索引, 若它出现在其中, 就将该元组写入结果
3. 差: 对关系 s 相应划分中的每一个元组, 检索 hash 索引, 若它出现在其中, 就将它从 hash 索引中删除. 将 hash 索引中剩余的元组加入结果中

12.3.4 去重

1. 用排序的方法.
 - 创建归并段文件时可以发现重复元组, 并在将归并段文件写回磁盘之前去除重复元组, 从而减少块传输次数
 - 归并时再去除剩余的重复元组
2. 用散列的方法
 - 基于整个元组上的一个散列函数对关系进行划分
 - 每个分划被读入内存, 建立内存中的散列索引
 - 建立散列索引时, 只有不在索引中的元组才被插入. 否则, 元组就被抛弃. 分划中的所有元组处理完后, 散列索引中的元组被写到结果中

12.3.5 连接运算

- 嵌套循环连接
- 块嵌套循环连接

- 索引嵌套循环连接
 - 归并连接
 - 散列连接
- r 称为连接的外关系, s 称为内关系

嵌套循环连接: 不需要索引, 可用于任何连接条件.

算法 12.1: 嵌套循环连接

```

Input: 关系  $r$  和  $s$ , 连接条件  $\theta$ 
Output: 连接结果  $r \bowtie_{\theta} s$ 
1 for 每个元组  $t_r$  属于  $r$  do
2   for 每个元组  $t_s$  属于  $s$  do
3     测试元组对  $(t_r, t_s)$  是否满足连接条件  $\theta$ ;
4     if  $(t_r, t_s)$  满足连接条件  $\theta$  then
5       将  $t_r \cdot t_s$  加入到结果中;
6     end
7   end
8 end

```

块嵌套循环连接: 在外层循环中使用较小的关系代价略小.

- 当较小的关系放在外层循环时, 每次外层循环迭代时, 只需要加载较少的数据到内存中进行处理. 这减少了磁盘 I/O 操作的次数, 从而提高了整体性能.
- 相反, 如果较大的关系放在外层循环中, 每次外层循环迭代时都需要加载大量的数据到内存中, 增加了 I/O 开销.

算法 12.2: 块嵌套循环连接

```

Input: 关系  $r$  和  $s$ , 连接条件  $\theta$ 
Output: 连接结果  $r \bowtie_{\theta} s$ 
1 for 每个块  $B_r$  属于  $r$  do
2   for 每个块  $B_s$  属于  $s$  do
3     for 每个元组  $t_r$  在  $B_r$  do
4       for 每个元组  $t_s$  在  $B_s$  do
5         检查  $(t_r, t_s)$  是否满足连接条件;
6         if  $(t_r, t_s)$  满足连接条件 then
7           将  $t_r \cdot t_s$  加入结果;
8         end
9       end
10      end
11    end
12 end

```

索引嵌套循环连接:

- 如果满足下列条件, 索引查找可以代替文件扫描
 - 是等值连接或自然连接
 - 在内关系的连接属性上存在索引
- 对外关系 r 的每个元组 t_r , 利用索引查找内关系 s 的与 t_r 满足连接条件的元组

归并连接: 用于执行两个关系之间等值连接操作的高效算法, 尤其适用于已经排序的数据. 其基本思想是将两个输入关系按照连接属性进行排序, 然后通过同步扫描这两个关系来找到满足连接条件的元组对.

算法 12.3: 归并连接算法

Input: 关系 r 和 s

Output: 连接结果 $r \bowtie_{r.A=s.A} s$

- 1 初始化指针 $i = 1, j = 1;$
- 2 **while** 仍有未处理的块或元组 **do**
- 3 在 r 和 s 的当前块中查找连接属性的最小值 v ;
- 4 **if** v 在另一关系的当前块中不存在 **then**
- 5 删除具有该值的所有本地元组;
- 6 移动指针以跳过这些元组;
- 7 **end**
- 8 **else**
- 9 收集 r 和 s 中所有连接属性等于 v 的元组;
- 10 **if** 某关系当前块不够, 需更多元组 **then**
- 11 从磁盘加载下一个包含该值的块;
- 12 **end**
- 13 执行局部笛卡尔积, 生成连接结果;
- 14 输出所有符合条件的结果元组;
- 15 **if** 某关系当前缓冲区无可用元组 **then**
- 16 重新加载该关系的缓冲区;
- 17 **end**
- 18 **end**
- 19 **end**

散列连接:

- 适用于自然连接和等值连接
- 基本思想
 - 将两个关系按连接属性值划分成有相同散列函数值的元组集合
 - 关系 r 在一个散列划分中的元组只需要与关系 s 在对应的划分中的元组相比较
 - 在 r 和 s 的每一对划分中进行索引嵌套循环连接 (或者散列连接)

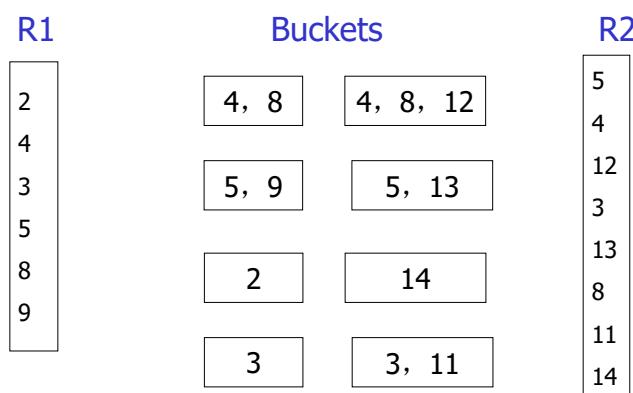


图 12.5: 散列连接示例

总结比较

- 如果一个连接输入很小 (比如不到 10 行), 而另一个连接输入很大而且已在其连接列上创建索引, 则索引嵌套循环是最快的连接操作
- 如果两个连接输入很大, 并已在二者连接列上排序 (连接列上有索引), 则合并连接是最快的连接操作

- 哈希连接可以处理很大的、未排序的非索引输入
- 如果两个连接输入都很大, 而且这两个输入的大小差不多, 则预先排序的合并连接提供的性能与哈希连接相似. 然而, 如果两个输入的大小相差很大, 则哈希连接操作通常快得多
- 合并连接和哈希连接只能用于等值连接, 对于非等值连接, 只能用嵌套循环连接

12.3.6 多表连接

N 个表可能的连接种类有 $N!$ 个

- 对少于或等于 4 个的表连接采用验证全部组合的方法, 对于超过 4 个的表连接, 以 4 个表为一组进行验证.
- ABCDEF6 个表, 先求出其任意 4 个表组合中代价最小的, 不妨设为 BCDF, 则表 B 作为最外层表, 剩余 5 个表 ACDEF, 再求出其任意 4 个表组合中代价最小的, 不妨设为 CDEF, 则表 C 作为次外层表, 最后按通常方法确定剩余表的最佳连接顺序, 假设为 DEFA, 则最后表连接顺序为 BCDEFA

12.3.7 算子执行方式

定义 12.5 (物化执行 (Materialization))

物化执行 (Materialization):

- 生成输入为关系或已计算结果的表达式的结果, 在磁盘上物化 (存储)
- 从最底层开始每次计算一个运算, 将中间结果物化成临时关系, 再进行下一层运算
- 物化执行总是可用的
- 将结果写到磁盘并读回来可能代价很高



定义 12.6 (流水线化 (Pipelining))

流水线化 (Pipelining):

- 当一个操作正在执行中, 就将部分输出元组送到父运算
- 流水线执行: 同时计算多个运算, 将一个运算的结果传到下一个运算
- 比物化执行代价要低, 不需要将临时关系存储到磁盘
- 有时流水线不可用 - 如排序和散列连接
- 为了使流水线有效, 使用能在接受输入元组的同时生成输出元组的执行算法 (无阻塞算子)



12.3.8 算子封装形式: 迭代器

- open(): 打开一个输入
- getnext(): 一次一个项的读取每一个项, 如果遇上一个项符合条件的话, 就将这个项传递给下一个处理迭代器
- close(): 关闭输入

第十三章 考试题型

- 填空题 + 简答题: 涵盖各章基础知识点
- 查询题: 关系代数、SQL
- 数据库设计题: ER 模型、关系规范化
- 调度分析: 隔离性与一致性、冲突与视图可串行化判定、各种并发控制协议

参考文献

- [1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.10. Arpaci-Dusseau Books, 2023.
- [2] Andrew Lamb et al. "The Vertica Analytic Database: C-store 7 Years Later". In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), pp. 1790–1801. issn: 2150-8097. doi: [10.14778/2367502.2367518](https://doi.org/10.14778/2367502.2367518). (Visited on 06/16/2025).
- [3] 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).
- [4] 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).
- [5] C. Mohan et al. "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging". In: *ACM Trans. Database Syst.* 17.1 (Mar. 1992), pp. 94–162. issn: 0362-5915. doi: [10.1145/128765.128770](https://doi.org/10.1145/128765.128770). URL: <https://doi.org/10.1145/128765.128770>.
- [6] Patrick O'Neil et al. "The Log-Structured Merge-Tree (LSM-tree)". In: *Acta Inf.* 33.4 (June 1996), pp. 351–385. issn: 0001-5903. doi: [10.1007/s002360050048](https://doi.org/10.1007/s002360050048). (Visited on 06/16/2025).
- [7] 亚伯拉罕·西尔伯沙茨 (Abraham Silberschatz), 亨利·F·科思 (Henry F. Korth), and S. 苏达尔尚 (S. Sudarshan). 数据库系统概念：原书第 7 版. Trans. by 杨冬青, 李红燕, and 张金波. 5th ed. 机械工业出版社, 2021.
- [8] Mike Stonebraker et al. "C-Store: A Column-Oriented DBMS". In: *Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker*. Ed. by Massachusetts Institute of Technology and Michael L. Brodie. 1st ed. Association for Computing Machinery, Dec. 2018, pp. 491–518. isbn: 978-1-947487-19-2. doi: [10.1145/3226595.3226638](https://doi.org/10.1145/3226595.3226638). (Visited on 06/16/2025).