

电子笔记合集

Lecture 1

Why Database?

DBSM：数据库管理系统

为什么要使用数据库？

还有哪些管理数据的方式：文件系统？

问题：数据格式规范 冗余

- 元数据（Meta Data）
 - 属于**数据库**本身的一些**数据**，包含**数据库名**、**数据库**字符集、表名、表的大小、表的记录行数、表的字符集、表的字段、表的索引、表的描述、字段的类型、字段的精度、字段的描述等
 - 数据读取：有一套系统读取
 - 数据隔离：将数据之间关联起来
 - 数据完整性
- 类似于社会发展：技术分工

What is a Database?

- 关系模型带来的表述形式
 - Tables 表
 - 属性（字段）
 - 元组（记录）
 - 约束（数据表间的逻辑联系）引用完整性约束
- 数据库看到的表是**逻辑概念**而不是物理概念（底层存储）
 - 底层存储是连续的，给每一种属性预留一些字节
 - 视图层：基于逻辑结构再抽象，应用程序按理说只能访问视图层数据
- 思想：数据抽象
 - 类似于data structure和type
 - 可以把底层的物理模式做修改，但是不影响上层的逻辑结构：磁盘坏但是不影响逻辑结构的变化。
 - 独立性：物理层→逻辑层；逻辑层→应用层
 - 存储管理器：数据库不同层之间的组织
- 数据库的查询处理组件（如何读SQL语句）
 - 解析出 关系代数表达式
 - 从 当前数据的统计信息 对表达式进行优化
 - 计算引擎从data中进行查询
 - 反馈
 - 如果查询计划特别慢→检查执行计划（手动优化）

- Transaction Management
 - 事务管理：数据正确（即使出现异常情况）
 - 并发管理：处理效率高

Who uses a Database?

- 数据库管理员
- 其他用户
 - 应用程序的用户
 - 程序员
 - 统计等领域的用户

How do we make it work?

- DDL(Data Definition Language)数据定义语言
 - 创建表
 - 定义数据类型
- DML(Data Manipulation Language)数据操纵语言
 - 过程式
 - 声明式

From Theory to Practice

- Data Model
 - 关系模型
 - ER建模 设计层面的表述
 - SQL：非过程式的语言
 - 嵌入式
 - API
 - 数据库设计：什么叫做“好的”设计？
 - 逻辑设计：减少数据冗余
 - 业务决策：要把哪些属性放到数据库里
 - 实现角度：如何让属性分配到合适的位置上
 - 物理设计：物理层面的合理组织，放几个磁盘？如何放到磁盘上？用什么数据结构？
 - 总结：
 - 什么是好的设计
 - 我们需要遵循哪些**规则**？规则在特殊情况下也可以用来打破
- 解决方式：
- 规范化
 - Denormalization
 - 设计过程（ER建模来分析需求）
 - 如果出现问题，如何纠正

Application Design

Lecture 2: Relational Model

- Relation: 表
- Attribute: 列
- Tuple: 行

Attribute

- 原子性: 不可再分
- 允许取特殊值: null

Relational Database

- 关系的模式 (id, name, dept_name, salary)
- $R = (id, name, dept_name, salary)$
- instructor(R)

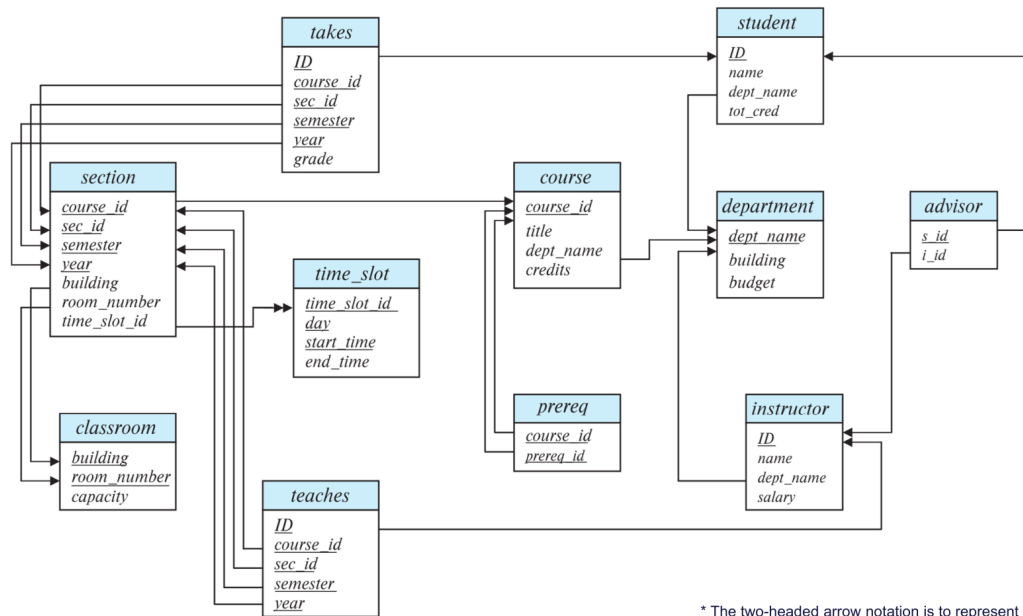
In General:

- 关系模式: $R(A_1, A_2, A_3, \dots, A_n)$
- $r(R)$
- r 是每个属性允许范围的笛卡尔积的子集 $D_1 \times D_2 \times D_3 \times \dots$
- r 是一个集合: 集合里的每一个元素是元组

Key

- 如何知道两个元组不同? 如何唯一识别元组
- **Superkey (超键) 能够唯一识别元组**
- **Candidate Key (候选键): 最小的 超键**
- **Primary Key (主键): 选择一个候选键, 根据实际需要设置**
 - 不是任何时候都可以使用自增的id作为主键, 因为含义无关;
 - 比如学号, 有一定的业务逻辑, 可能查询的效率会变高
- Foreign Key (外键): 是其他关系的主键, 指向另一个表
- Schema Diagrams

Schema Diagram



* The two-headed arrow notation is to represent a special type of referential integrity.

Software School, Fudan University

Database Design

Autumn Semester, 2023

- 双箭头: time_slot_id →→
 - 因为time_slot_id不是主键（可以重复）所以只是引用

关系代数

如何操作一个数据库？

- 六种基础操作：原子的定义，相对于附加操作的定义；**本质是集合运算**
- 输入：一个或两个关系，输出：一个新的关系

选择 σ

- 条件
- 逻辑运算：与 或 非

投影 Π

- 选择属性
- 自动去重

并 Union

- 条件：
 - 属性数量一致
 - 属性的取值范围是兼容的
- 作用：“并”操作后 最终的schema是有意义的，这是针对你的需求来决定
- 例如学生和老教师，要看如何解释
- **如果不兼容，就不能做并操作**

差集 set difference

- 类似于集合做差，书面语：“r和s的差集”

笛卡尔积 Cartesian product

- 元组数量是二者之和
- 很多情况下不是直接使用的（因为元组数量太多了）
- 把两个及两个以上的关系连结起来的基本手段

重命名 Renaming

- 可以通过不同的名字引用同一个关系
- 应用：例如自己和自己做笛卡尔积

组合

将以上操作组合起来

实际运算-举例

- 如果“并”操作，属性名不兼容，可以通过重命名的方式——一般是将名字改成相同，即兼容
- 尽量让中间的数据集减小（如果有多种方式，尽量选择中间数据集小的方式）
- 没有“交”操作，但可以通过差集（看Venn图理解）得到同样效果
- 连表查询：连表做笛卡尔积，筛选出属性相等的（保证数据合理），之后再和另一个表筛选+投影
- 如果有多种办法都可以达成效果
 - 让中间结果变小，一般来说可以先进行筛选
 - 但是实际效率取决于具体数据

额外操作 Additional Operation

交集

自然联结 Natural Join

- 相同元组的属性取值相同时保留
- 类似于笛卡尔积+选择+投影

Theta Join

- 自然联结的泛化：可以设置条件，对条件进行选择

除 Division

- 是否完全包含“被除关系”中的属性
- 判断一个全集的概念
- “商”和“被除数”的笛卡尔积是“除数”的“最大子集”
- 与笛卡尔积有一些类互逆的运算

- 注意：在SQL语句中不支持，但是要用类似于基本操作的组合方法

赋值

- 扩展表达方式

举例

- 要对schema非常熟悉，能知道是在那个表里寻找
- 使用division的好处：“被除数”（筛选条件）是动态/难以直接用属性表达的

Review

- 自然联结：投影（选择（笛卡尔积））
- Division：找到都包含某个属性的记录

Extended Operations

泛化投影 (Generalized Projection)

- 投影时做一些计算

聚合函数 (Aggregate Functions)


- 按属性分组，按聚合函数（avg sum min max count等）计算

外联结 (Outer Join)

主表拼接另外的表

- 主表保留所有记录，记录不会丢失（不像自然联结）
- 如果另一个表没有相应记录的话，则置为空

• Left Outer Join 左外联结


course  prereq

主表

course_id	title	dept_name	credits	prere_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	null

左外联结：外联结可以快速查看主表中没有prere_id的course

• Right Outer Join

course  prereq

course_id	title	dept_name	credits	prere_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	null	null	null	CS-101

右外联结：查看哪些course的title和dept_name没有被录入

- 一般来说不使用全外联结

空值 Null Values

- 和null直接做运算是null
- 但是类似于sum的运算会忽略null值
- count时, null记录也会被记录下来

Modification of the Database

插入

删除

- 删除时注意除了该表本身之外, 其他表也用到该属性的记录也需要删除 (避免空指针的情况) “外键的约束”

更新

Lecture 3: SQL

SQL-Basic

- SQL: 结构化的查询方法
- DDL: Data-Definition Language 数据定义语言
- 数据类型
 - char: 建议存储固定长度
 - varchar: 长度可变, 可长可短
 - numeric(x, y): x为有效数字, y为小数点后的精度
 - 注意: 同一语义的字段, 在不同表里设置的数据类型需要一致 (否则在查询关联等地方会出问题)
- 完整性约束
 - 实体完整性: 主键非空
 - 引用完整性: 外键在参照的表里有 (注意这里有插入表的先后顺序)
 - 根据具体的需求确定主键是什么 (需要由哪些值构成)

Select

- distinct 去除所有相同的
- all 找到所有的
- SELECT *
 - 如果字段名, 字段数量变化, 可能select的时候没出错但是使用的时候出错
 - 所以最好查询时写上所有字段
- 取常量也是ok的 `SELECT '437'`
 - 这样会取出 {(437)}
 - 但是在Oracle里面不允许这样做, 需要写成 `SELECT '437' from dual`

Where

- 多个条件可以加上 and or not等
- 算数运算结果也可以进一步判定

特殊语法

- `**where** salary **between** 70000 **and** 100000` (在一次遍历内判断成功, 如果是两个条件会做两次判断)
- 元组对应: `**where** (instructor.ID, dept_name) = (teaches.ID, 'Biology');`

From

笛卡尔积: `from instructor, teaches`

As

重命名: `**from** instructor **as** T, instructor **as** S`

字符串操作

- like 模糊匹配 转移 大小写转换 子串分解 有额外的包和函数等等
- 做了拆解之后索引就无法使用, 所以处理字符串的时候不要作为中间结果

Order by

如果没有排序: 物理顺序

- asc 升序 desc 降序
- 有多个属性进行排序时, 从前向后优先级递减
- 并不建议使用大量的子查询, 而是可以使用having等一步到位

Lecture 4

View 视图

- 重命名, 有点像子查询
- 当我经常要使用某个内容的时候, 可以提取出视图来

依赖关系

- 直接依赖
- 依赖路径
- 嵌套依赖

视图并没有增加查询效率, 但是会使得查询更好理解。

视图更新

限制：

- view的来源只有一张表
- select出来的只有原来的属性，不能是计算之后的属性
- 如果有非空约束，而插入之后导致空，则会失败

注意：

- 视图没有存储的能力
- 插入时原本表中没有的字段会存储为空
- 如果视图from的表有多个，不允许向view查数据

视图的权限分配：

- 谁可以访问
- 谁可以读
- 谁可以写

物化（持久化）视图：

Transaction 事务

表示原子性的操作，由查询或更新语句的队列组成

终止事务：commit rollback

确保程序在数据的更新上保持原子性：🔒

一上来把自动提交关闭，这样才能保证在出现问题时出现回滚；例如，在try catch中的catch中rollback

数据类型

日期

在具体的DBMS中可能不同

- timestamp：年月日+时分秒
- 强制转换：和中间的分隔符有关

思考：储存日期使用字符串好还是日期格式好

- 大部分情况下是用日期格式
- 字符串的好处：写入的是什么，出来的就是什么（中间的分隔符-）
- 不同国家和地区对年月日的顺序显示习惯不同

用户自定义类型

- final：不能用该类型再去定义新的类型
- type：强类型，只要名字不同两个类型就不一样，不能做赋值操作；但可以强制转换
- domain：弱类型，

大对象

- blob: binary large object
- 查询时通常返回的是一个指针
- 在查询大型类型的时候尽量不要使用select *

完整性约束

- 比如值低于多少时就不让操作进行下去
- not null 和 unique (互不相同)
- check : 后面写约束条件
- 引用完整性: 好处: 保证不出错; 坏处: 性能下降
 - 一般来说会把外键补上去
- cascade: 级联的更新, 同时删除, 更新等
 - 在设置外键时设置好级联是什么
- 断言: assertion 用到某个数据的记录将会检查
- 视图主要是对列数据的控制

授权

角色 role

SQL授权限制

不要用前端的拼接来限制各个角色的授权hh

Lecture 4A: Advanced SQL

第一个lab的时候不用框架, pj可以

Prepared Statement

预备语句 用?代表参数

防止SQL注入问题

嵌入式SQL

fetch: 指针cursor指向下一条记录

E-R Graph

- 是不是一定需要将id作为主键? ——和设计相关, 有时使用period也可以

弱实体集

ER图拓展特性

- 什么时候需要做特化泛化？
 - 设计的层次适合那种体系结构，不是必须的

从ER图到schema

- 设计和实现分开

disjoint和overlapping

在数据库设计中，disjoint 和 overlapping 通常用于表示不同的关系。disjoint 用于表示具有完全不同属性的类别，例如动物和植物。overlapping 用于表示具有部分相同属性的类别，例如颜色。

以下是 disjoint 和 overlapping 在数据库设计中的应用示例：

- **disjoint**
 - 用户可以分为普通用户和管理员。普通用户和管理员是 disjoint 的，即一个用户不能同时是普通用户和管理员。
 - 产品可以分为实物产品和虚拟产品。实物产品和虚拟产品是 disjoint 的，即一个产品不能同时是实物产品和虚拟产品。
- **overlapping**
 - 颜色可以分为红色、黄色、蓝色、绿色和黑色。这五个类别是 overlapping 的，即一个对象可以同时是红色和黄色。
 - 商品可以分为食品、服装、电子产品和家居用品。这四个类别是 overlapping 的，即一个商品可以同时属于多个类别。

Storage Hierarchy

硬盘

- 顺序存储
- 存取时间：和访问延迟（寻道时间+旋转延迟）+IO速率有关
- 如何提升磁盘的访问速度：（从访问的角度来说）
 - 数据排列方式：减少碎片化/增加相邻⇒减少寻道时间
 - 硬件角度：硬盘本身质量，如转速等

冗余磁盘阵列

- 如何高效可靠存取数据
- 10个1T的硬盘，每个磁盘都有概率故障，如何降低损失？
- 可靠性：同时将数据副本放在多个盘（镜像）
 - 逻辑盘（类似于虚拟内存？）包含两个物理盘
- 性能：
 - 通过并行化，将一份数据拆成若干部分分散在不同的物理盘中，则访问的性能可以提升

- 条带化striping：按块（比较常用的方式）或者按字节

Indexing and Hashing

有没有可能更快地找到数据，而不是全部遍历

- search-key ↔ pointer
- 索引文件比原始文件小的多
- 给定一个值/一个范围能不能快速查到

需要考虑的内容：访问类型、访问时间、插入时间、删除时间、空间开销

1. 顺序索引

按照顺序的方式组织

索引的顺序和文件的顺序一致吗？

- Clustering index：包含记录的文件按照某个搜索码指定的顺序排序
 - 做范围查找是连续的
- 非聚集索引：搜索码的顺序和文件存储的物理顺序不同
 - 做范围查找是不连续的

1.1 稠密索引和稀疏索引

- 稠密索引
 - 文件中的每个搜索码值都有一个索引项
 - 索引项包括**搜索码值**以及指向具有该搜索码值的**第一条数据记录的指针**
- 稀疏索引
 - 只为搜索码的某些值建立索引项
 - 只有索引是聚集索引时才能使用稀疏索引
 - 索引项包括**搜索码值**以及指向具有该搜索码值的**第一条数据记录的指针**

10101	10101	Srinivasan	Comp. Sci.	65000
12121	12121	Wu	Finance	90000
15151	15151	Mozart	Music	40000
22222	22222	Einstein	Physics	95000
32343	32343	El Said	History	60000
33456	33456	Gold	Physics	87000
45565	45565	Katz	Comp. Sci.	75000
58583	58583	Califieri	History	62000
76543	76543	Singh	Finance	80000
76766	76766	Crick	Biology	72000
83821	83821	Brandt	Comp. Sci.	92000
98345	98345	Kim	Elec. Eng.	80000

图 11-2 稠密索引

10101	10101	Srinivasan	Comp. Sci.	65000
32343	12121	Wu	Finance	90000
76766	15151	Mozart	Music	40000
	22222	Einstein	Physics	95000
	32343	El Said	History	60000
	33456	Gold	Physics	87000
	45565	Katz	Comp. Sci.	75000
	58583	Califieri	History	62000
	76543	Singh	Finance	80000
	76766	Crick	Biology	72000
	83821	Brandt	Comp. Sci.	92000
	98345	Kim	Elec. Eng.	80000

图 11-3 稀疏索引

1.2 多级索引

- 至少能够给每一个块建立入口，更快的找到
- 级联的方式可能导致碎片化

1.3 索引的更新

- 删除
 - 需要区分（根据索引和搜索码值的关系）
- 插入
 - 区分该搜索码值是否在索引中
- 调整可能会带来overflow block

1.4 辅助索引

- 辅助索引必须是稠密索引
- 可以是非聚集索引

检索效率：

- 用顺序索引扫描（scan）index是快速的
- 但是用辅助索引可能开销较大
- full table scan：从头到尾扫描
- 但是基于index可以使用unique/range的扫描

总的来说：

- 辅助索引能够提高使用聚集索引搜索码以外的码的查询性能。
- 但是，辅助索引显著增加了数据库更新的开销。数据库设计者根据对查询和更新相对频率的估计来决定哪些辅助索引是需要的。

2. B+索引文件

其实也是顺序索引，利用树的形式可以做**局部性**的更新

2.1 B+树结构

- 稠密的辅助索引
- （对于寻找下层节点的方式）左侧小，右侧大
- B: Balance 根到叶结点的每条路径长度都相同
 - 内部节点: $[n/2] \sim n$ 个指针
 - 叶节点: $[(n-1)/2] \sim n-1$
- 逻辑上连接的索引在物理上不一定相邻
- 本质上每层都是稀疏索引
- 树的高度不会超过 $\log [n/2] K$ 层次比较低→检索高效

2.2 B+树的查询

- 每一个节点和disk block的大小相同

2.3 B+树文件组织

- 底层的叶子节点上存好了记录（不需要再指一次）

2.4 B树和B+树的区别

- B树的复杂度会提高（每个节点需要多存一个指针）

3. 静态哈希

将值映射到更小的空间之内，基于散列(hashing)技术的文件组织使我们能够避免访问索引结构

- 桶bucket内部是按照堆的形式
- 在实际应用中比较有限（az

3.1 散列函数

- 理想的函数能够让分布是均匀且随机的

4. 多码访问

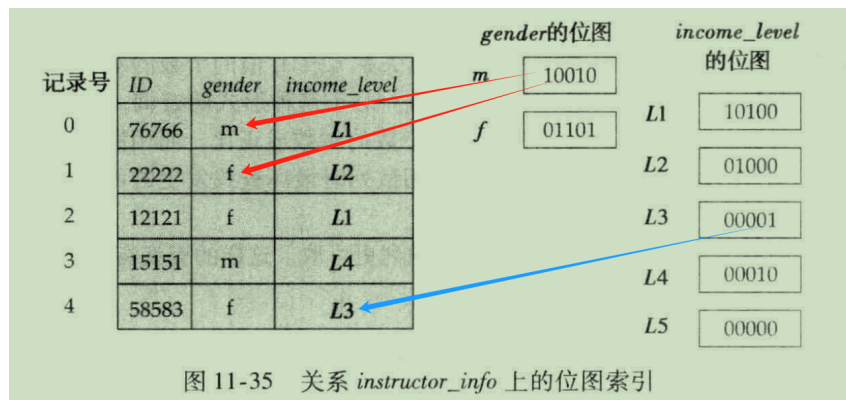
- 三种查询方法
- 如何提升性能？做一个联合的索引

4.1 多码索引

- 复合搜索码上建立和使用索引
- 能不能用起来和数据的组织和排序也有关系
- bitmap（位图）的方式组织
 - 男女
 - 按照范围分类

4. 位图索引

- 前提：知道某个记录是第几条，找到那条记录的位置
- 位运算：在寄存器上运算 非常快
- 按位表达记录：10000条记录：10000个位 (bit)
- 离散成一个一个区间
- 每一个bitmap（位图）表示对应属性的取值情况



- 同时满足多个条件：（位运算：与）
- 位图索引较快的地方：满足多个条件
- 需要检查哪些位置上记录已经不存在了：（因为在做not操作的时候会出问题）

5. SQL中的索引定义

- 有单独的表空间
- 索引可以和业务数据分开（查询索引和取数据可以并发来做）

6. 索引的使用时机和问题

什么时候使用索引

- 取值本身是唯一的
- 一个取值会对应到很多记录
-

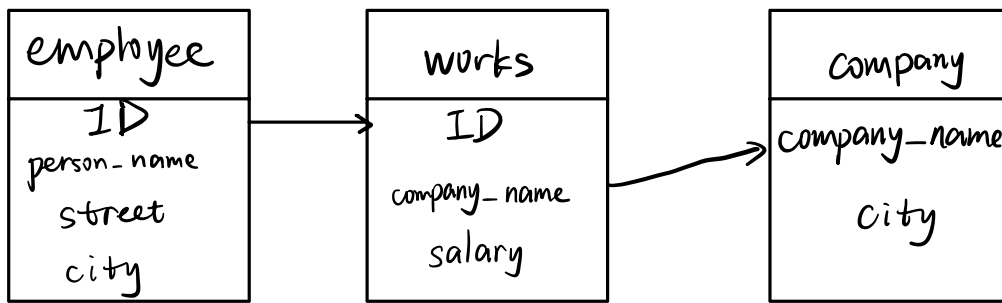
索引的问题

- 索引多但是更新特别频繁：索引维护的开销较大

创建索引的关注点

- 表查询的频率较高：应该增加相应索引
- 表更新的频率较高：
- 选择经常出现的条件
- 系统运行时持续监控索引的性能，检查查询的性能
- 在ORACLE中可以对索引进行REBUILD，提升整体查询的效率
- EXPLAIN PLAN：告诉数据库是如何进行索引的，如何优化的

HW1



2-6

b. $\Pi_{\text{person-name}} (\sigma_{\text{salary} > 10000} (\text{employee} \bowtie \text{works}))$

c. $\Pi_{\text{person-name}} (\sigma_{\text{city} = \text{"Miami"}} (\sigma_{\text{salary} > 10000} (\text{employee} \bowtie \text{works})))$

2.8 a. $\Pi_{\text{ID}, \text{person-name}} (\sigma_{\text{company-name} \neq \text{"BigBank"}} (\text{employee} \bowtie \text{works}))$

b. $\Pi_{\text{ID}, \text{person-name}} (\text{employee}) -$

$\Pi_{\text{ID}, \text{person-name}} (\text{employee} \bowtie \Pi_{\text{ID}, \text{works.balance}} (\text{works} \bowtie_{\text{works.salary} < \alpha \cdot \text{salary}} \rho_{\alpha}(\text{works}))))$

2.9 a. $\Pi_{\text{ID}, \text{course-id}} (\text{takes}) \div \Pi_{\text{course-id}} (\sigma_{\text{dept.name} = \text{"Comp.Sci."}} (\text{course}))$

b. $r = \Pi_{\text{ID}, \text{course-id}} (\text{takes}) \quad s = \Pi_{\text{course-id}} (\sigma_{\text{dept.name} = \text{"Comp.Sci."}} (\text{course}))$

$\text{res} = \Pi_{\text{ID}} (\text{takes}) - \Pi_{\text{ID}} ((\Pi_{\text{ID}} (r) \times s) - \Pi_{\text{ID}, \text{course-id}} (r))$

$(\text{res}) : r \div s = \Pi_{RS} (r) - \Pi_{RS} ((\Pi_{RS} (r) \times s) - \Pi_{RS, S} (r))$

2.14

$$c. \Pi_{ID, person_name, street, city} (\sigma_{salary > 10000 \wedge company_name = "BigBank"} (employee \bowtie works))$$

$$d. r = employee \bowtie works$$

$$\Pi_{ID, person_name} (\sigma_{r.city = company.city} (r \bowtie_{r.company_name = company.company_name} company))$$

2.15

$$c. \Pi_{ID} (\sigma_{branch_name = "Uptown" \wedge balance > 6000} (depositor \bowtie account))$$

2.18

$$c. \Pi_{ID, name}(student) \bowtie \Pi_{ID} (\sigma_{dept_name = "Comp.Sci"} (course \bowtie takes))$$

$$d. \Pi_{ID, name}(student) \bowtie \Pi_{ID} (\sigma_{year = 2018} (takes))$$

$$e. \Pi_{ID, name}(student) - \Pi_{ID, name}(student) \bowtie \Pi_{ID} (\sigma_{year = 2018} (takes))$$

HW2

3.4

a

Find the total number of people who owned cars that were involved in accidents in 2017.

思路：注意每两个关系之间的外键联系；特别是participated关系中的driver_id并没有下划线

```
select count(distinct driver_id)
from person, accident, owns, participated
where accident.year = 2017
      and accident.report_number = participated.report_number
      and participated.license = owns.license
      and owns.driver_id = person.driver_id
```

3.11

c

For each department, find the maximum salary of instructors in that department. You may assume that every department has at least one instructor.

```
-- 直接在instructor里面取即可 --
select dept_name, max(salary)
from instructor
group by dept_name
```

d

Find the lowest, across all departments, of the per-department maximum salary computed by the preceding query.

```
-- from中的嵌套子查询 --
select min(max_salary)
from {select dept_name, max(salary) as max_salary
      from instructor
      group by dept_name}
```

3.12

a

Create a new course "CS-001", titled "Weekly Seminar", with 0 credits.

```
insert into course
values ('CS-001', 'weekly seminar', 'Comp. Sci.', 0)
```

b

Create a section of this course in Fall 2017, with sec id of 1, and with the location of this section not yet specified.

```
insert into section (course_id, sec_id, semester, year)
values ('CS-001', 1, 'Fall', 2017)
```

c

Enroll every student in the Comp. Sci. department in the above section.

```
insert into takes(ID, course_id, sec_id, semester, year)
select ID, 'CS-001', 1, 'Fall', 2017
from student
where dept_name = 'Comp. Sci.'
```

d

Delete enrollments in the above section where the student's ID is 12345.

```
delete from takes
where ID = 12345 and (course_id, sec_id, semester, year) = ('CS-001', '1',
'Fall', 2017)
```

e

Delete the course CS-001. What will happen if you run this delete statement without first deleting offerings (sections) of this course?

A: 会导致section中和CS-001相关的所有字段同时被删除

f

Delete all takes tuples corresponding to any section of any course with the word “advanced” as a part of the title; ignore case when matching the word with the title.

```
-- 注意 不需要区分大小写 --
delete from takes
where course_id in (
select course_id
from takes, section, course
where takes.course_id = section.course_id
and section.course_id = course.course_id
and lower(course.title) like '%advanced%'
)
```

3.13

Write SQL DDL corresponding to the schema in Figure 3.17. Make any reasonable assumptions about data types, and be sure to declare primary and foreign keys.

注意：

- 同一语义的字段，在不同表里设置的数据类型需要一致
- 在使用外键的时候，注意插入表的顺序

```
create table person (  
    driver_id    char(10),  
    name        varchar(20) not null,  
    address     varchar(40),  
    primary key (ID)  
);  
  
create table car (  
    license_plate char(10),  
    model        varchar(40),  
    year         numeric(4,0),  
    primary key  (license_plate)  
);  
  
create table accident(  
    report_number varchar(10),  
    year          numeric(4,0),  
    location      varchar(40),  
    primary key   (report_number)  
);  
  
create table owns (  
    driver_id    char(10),  
    license_plate char(10),  
    primary key  (driver_id, license_plate),  
    foreign key  (driver_id) reference person on delete cascade,  
    foreign key  (license_plate) reference car on delete cascade  
);  
  
create table participated (  
    report_number varchar(10),  
    license_plate  char(10),  
    driver_id      char(10),  
    damage_amount  numeric(8,2),  
    primary key    (report_number, license_plate),  
    foreign key    (report_number) reference accident,  
    foreign key    (license_plate) reference car  
);
```

-- 我对delete cascade的理解是，在现实意义上，一个字段删除之后相关的字段有没有必要被删除 --

3.28

Using the university schema, write an SQL query to find the names and IDs of those instructors who teach every course taught in his or her department (i.e., every course that appears in the course relation with the instructor's department name). Order result by name.

```
select ID, name
from instructor
where not exists (
    (select course_id, dept_name
     from course as C, instructor as I
     where C.dept_name = I.dept_name
    )
    except
    (select T.course_id, I.dept_name
     from teaches as T, instructor as I
     where T.ID = I.ID
    )
)
order by name asc
```

4.3

Outer join expressions can be computed in SQL without using the SQL outer join operation. To illustrate this fact, show how to rewrite each of the following SQL queries without using the outer join expression.

a

```
select * from student natural left outer join takes
```

Answer:

```
-- 内联结+在右关系中ID不存在的左关系字段 --
(select S.ID, name, dept_name, tot_cred, course_id, sec_id, semester, year, grade
 from student as S, takes as T
 where S.ID = T.ID)
union
(select S.ID, name, dept_name, tot_cred, null, null, null, null, null
 from student as S, takes as T
 where not exists(
     select S.ID
     from T
     where S.ID = T.ID
 )
)
```

b

```
select * from student natural full outer join takes
```

Answer:

```
(select S.ID, name, dept_name, tot_cred, course_id, sec_id, semester, year, grade
from student as S, takes as T
where S.ID = T.ID)
union
(select S.ID, name, dept_name, tot_cred, null, null, null, null, null
from student as S, takes as T
where not exists(
    select S.ID
    from T
    where S.ID = T.ID
)
)
union
(select T.ID, null, null, null, course_id, sec_id, semester, year, grade
from student as S, takes as T
where not exists(
    select T.ID
    from S
    where S.ID = T.ID
)
)
```

4.17

Express the following query in SQL using no subqueries and no set operations.

```
select ID
from student
except
select s_id
from advisor
where i_id is not null
```

Answer:

```
-- 含义：找出student关系中有但是advisor关系中没有的id --
-- 即：找出所有没有导师的学生id --
select ID
from student natural left outer join advisor on student.ID = advisor.s_id
where advisor.i_id is null or advisor.s_id = null
```

HW3

HW3

7.1
7.7
7.9
7.18
7.19
7.30
7.38

7.1

Suppose that we decompose the schema $R = (A, B, C, D, E)$ into

$(A, B, C) = R_1$

$(A, D, E) = R_2$

Show that this decomposition is a lossless decomposition if the following set F of functional dependencies holds:

$A \rightarrow BC$

$CD \rightarrow E$

$B \rightarrow D$

$E \rightarrow A$

Answer:

考虑 $R_1 \cap R_2 = \{A\}$, 而 $A \rightarrow BC$, 所以 $R_1 \cap R_2$ 是 R_2 的超键, 所以是无损分解【书本P194】

7.7

Using the functional dependencies of Exercise 7.6, compute the canonical cover F_c .

$A \rightarrow BC$

$CD \rightarrow E$

$B \rightarrow D$

$E \rightarrow A$

Answer:

计算正则覆盖:

1. 合并左端相同的 (都不相同)
2. 寻找无关属性: 因为左端的属性都只在右端出现一次, 所以**没有无关属性**

因此 F_c 就是 F 本身

7.9

Given the database schema $R(A, B, C)$, and a relation r on the schema R , write an SQL query to test whether the functional dependency $B \rightarrow C$ holds on relation r . Also write an SQL assertion that enforces the functional dependency. Assume that no null values are present. (Although part of the SQL standard, such assertions are not supported by any database implementation currently.)

Answer:

1. 检验函数依赖 $b \rightarrow c$ 是否在关系 r 上成立的SQL查询

- $b \rightarrow c$: 在 r 每个元组中, 如果 b 属性的值相同, 则 c 的值都相同
- 解决方法: 按照 b 分类, 然后检查每个类中不同 c 的个数是否大于1

```
SELECT COUNT(*)
FROM r
GROUP BY b
HAVING COUNT(DISTINCT c) >1
```

2. 写出保证函数依赖的SQL断言

```
CREATE ASSERTION b_c_on_r
CHECK (
    NOT EXISTS (
        SELECT COUNT(*)
        FROM r
        GROUP BY b
        HAVING COUNT(DISTINCT c) >1
    )
)
```

7.18

Let a **prime** attribute be one that appears in at least one candidate key. Let α and β be sets of attributes such that $\alpha \rightarrow \beta$ holds, but $\beta \rightarrow \alpha$ does not hold. Let A be an attribute that is not in α , is not in β , and for which β -holds. We say that A is **transitively dependent** on α . We can restate the definition of 3NF as follows: A relation schema R is in 3NF with respect to a set F of functional dependencies if there are no nonprime attributes A in R for which A is transitively dependent on a key for R . Show that this new definition is equivalent to the original one.

证明: 3NF: 不存在 非主属性 传递依赖于 候选键

Answer:

- 3NF的原定义:

对于 F^+ 中所有形如
 $\alpha \rightarrow \beta$ 的函数依赖(其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$), 以下至少一项成立:

- $\alpha \rightarrow \beta$ 是一个平凡的函数依赖。
- α 是 R 的一个超键。
- $\beta - \alpha$ 中的每个属性 A 都包含于 R 的一个候选键中。

- 证明:

1. 【充分性】若存在非主属性 C 传递依赖于候选键 α ($\alpha \rightarrow \beta$) ($\beta \rightarrow C$); 考虑 $\beta \rightarrow C$ 中 C 不在 R 的任何候选键中, 且 β 不是超键, 这个函数依赖也不是平凡的, 所以不满足3NF;
2. 【必要性】如果不存在非主属性 C 传递依赖于候选键 α , 那么用反证法, 如果此时不满足3NF, 即存在一组 $\alpha \rightarrow \beta$, 上述三项均不成立; 所以 β 中有属性 D 不在 R 的任意一个候选键中, D 为非主属性; 令 γ 为 R 的一个候选键, 则 $\gamma \rightarrow \alpha$ 与 $\alpha \rightarrow \gamma$ 不成立, 且 A 不在 γ 和 α 中, 所以 D 传递依赖于 γ , 矛盾, 所证成立

所以两个定义等价

7.19

A functional dependency $\alpha \rightarrow \beta$ is called a **partial dependency** if there is a proper subset γ of α such that $\gamma \rightarrow \beta$; we say that β is *partially dependent* on α . A relation schema R is in **second normal form (2NF)** if each attribute A in R meets one of the following criteria:

- It appears in a candidate key.
- It is not partially dependent on a candidate key.

Show that every 3NF schema is in 2NF. (*Hint*: Show that every partial dependency is a transitive dependency.)

证明每个3NF都属于2NF。(提示：证明每个部分依赖都是传递依赖。)

Answer:

- 对于上述部分依赖可以得到： $\alpha \rightarrow \gamma \rightarrow \beta$ （在这里我们定义 β 为非主属性，为了满足3NF的要求）
- 因为 α 不依赖于 γ ，且 β 不依赖于 α ，所以满足了传递依赖。
- 所以每个3NF都属于2NF

7.30

Consider the following set F of functional dependencies on the relation schema (A, B, C, D, E, G):

$A \rightarrow BCD$

$BC \rightarrow DE$

$B \rightarrow D$

$D \rightarrow A$

1. Compute B^+ .

- {A, B, C, D, E}

2. pass

3. Compute a **canonical cover** for this set of functional dependencies F; give each step of your derivation with an explanation. (计算正则覆盖)

1. 合并左端相同的属性（没有）

2. 寻找无关属性：

- 在 $A \rightarrow BCD$ 中，D为无关属性，而B和C都不是无关属性
- 在 $BC \rightarrow DE$ 中，D为无关属性，其他都不是无关属性

所以正则覆盖为：

$A \rightarrow BC$

$BC \rightarrow E$

$B \rightarrow D$

$D \rightarrow A$

4. Give a 3NF decomposition of the given schema based on a canonical cover.

基于正则覆盖做3NF分解

【化简，独立王国，选领袖】

1. $R1=\{A, B, C\}$ $R2=\{B, C, E\}$ $R3=\{B, D\}$ $R4=\{D, A\}$
2. 因为上述五个属性集都不包含F的候选键，所以 $R5=\{A, G\}$

3NF分解: $\{\{A, B, C\}, \{B, C, E\}, \{B, D\}, \{D, A\}, \{A, G\}\}$

5. Give a BCNF decomposition of the given schema using the original set F of functional dependencies.

【抓坏蛋，拉出去，砍了】

1. $A \rightarrow BCD$; A的闭包不包含G，所以A不是超键；将F集合分割成 $\{A, B, C, D\}$ $\{A, E, G\}$
2. $A \rightarrow BCD \rightarrow E$; A的闭包不包含G，所以A不是超键；将 $\{A, E, G\}$ 分成 $\{A, E\}$ $\{A, G\}$
3. 除此之外，因为A B D均为超键，所以 $\{A, B, C, D\}$ 符合BCNF

所以BCNF分解: $\{\{A, B, C, D\}, \{A, E\}, \{A, G\}\}$

7.38

In designing a relational database, why might we choose a non-BCNF design?

在关系数据库设计中，为什么我们有可能会选择非 BCNF 设计？

Answer:

1. 复杂度考虑：有时候数据库较小，则不需要选择BCNF设计来增大工作量和数据库设计的复杂度；
2. 性能角度考虑：BCNF要求每个**非主属性完全依赖于候选键**，这可能导致表的拆分和连接操作的增加，从而影响查询性能。在某些情况下，我们可以接受一定的冗余数据或部分依赖，以换取更高的查询性能。
3. 维护和开发成本考虑：和前面两点类似，选择BCNF设计可能会带来更多的表和关联操作，从而增大成本，在现实生活中需要考虑多种因素。