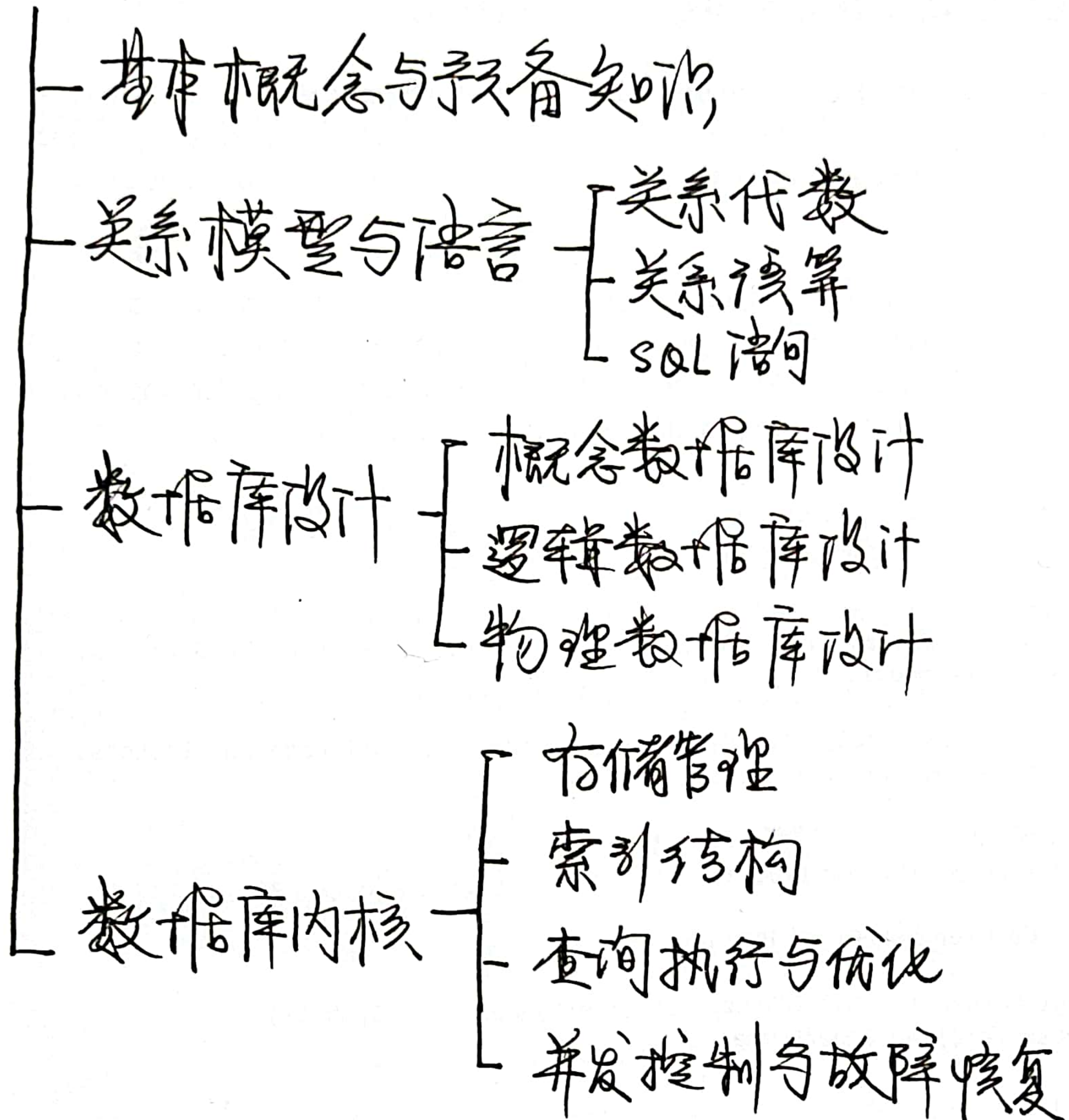
前言

说实话，写这个复习指南（讲义），心里是十分忐忑的，因为《数据库系统》这门课程知识量非常丰富，而且有一定难度，总是担心会误导同学们。我也无法预测考试的内容和重点，所以只能尽可能详尽整理出一份复习指南（讲义），有一些地方用课件展示不是很方便，所以我的这份讲义是手写版，为了方便大家查阅，我把其进行了扫描。不幸的是，由于时间紧任务重，可能字迹比较潦草，请大家见谅。同时，这里的内容均是按照我个人的思路和理解整理的，未必适合所有人，也未必是科学的，大家各取所需即可。

在讲的过程中，由于时间限制，只选取了其中最重要的一部分讲解，余下的内容在本文件中均有体现。需要说明的是，这个复习指南是不能代替老师的课件的，大家还是要以老师的课件为主，看过课件以后如果觉得有必要可以再看一下这个文件，简要了解一下框架。

由于作者水平有限，其中难免会有错误，欢迎大家批评指正。

目录（整体结构）



一、基本概念与预备知识

1. 基于文件系统的数据管理 VS 基于数据库管理系统的数据管理

2. DB VS DBMS VS DBS

3. 数据模型：是进行数据抽象的工具。

将现实世界映射到计算机世界的过程。

(1) 数据模型的三要素
- 描述DB结构的概念
- 操纵数据结构的操作
- DB服从的一系列约束条件

(2) 分类
- 概念数据模型：现实世界映射到信息世界
- 实现数据模型：介于↓↑之间，实现DBMS时使用
- 物理数据模型：DB在计算机中的存储细节

4. 数据库模式 VS 数据库实例 → 数据库在某一特定时间存储的数据

↓

对数据库的结构、类型、约束的描述，是DB的类型声明

5. 数据库的三层模式结构
- 内模式/存储模式 —— 物理模型
- 概念模式 —— 实现模型
- 外模式/视图 —— 实现模型（可以有多个）

（内模式、概念模式：只有一个）

(1) 模式映射用于完成请求转换和数据转换 → DBMS检索到的数据转换为符合外模式的组织形式

↓

应用程序在外模式上声明的数据请求 转换→ DBMS在内模式上的请求

(2) 模式映射的分类
- 外模式—概念模式映射
- 概念模式—内模式映射

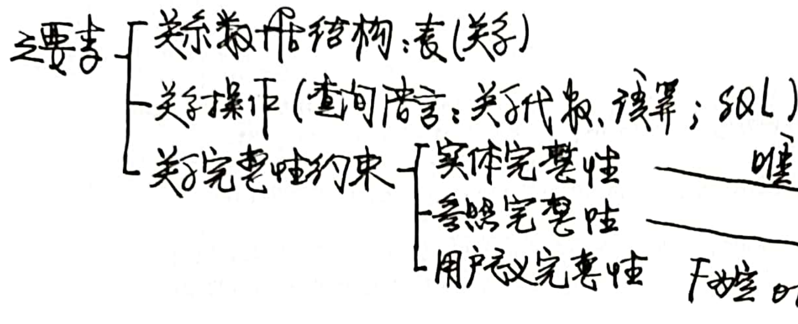
(3) 数据独立性
- 逻辑数据独立性：概念模式变化时修改外模式到概念模式的映射
- 物理数据独立性：内模式发生变化时，修改概念模式到内模式的映射

6. 数据库语言
- 数据定义语言(DDL)：Create
- 数据操纵语言(DML)：Select

二、关系模型与语言

(一)基本概念

1、关系数据模型(是一种实现数据模型)


三要素
- 关系数据结构：表(关系)
- 关系操作(查询语言：关系代数、演算；SQL)
- 关系完整性约束
  - 实体完整性 —— 唯一、非空 → 主键
  - 参照完整性 → 外键
  - 用户定义完整性

F为空 or 在S中存在

2. 码键：具有区分不同元组的作用的属性的集合

↓

候选键：极小的码键

↓ 人为指定一个

主键

F是R的外键

外键：R(F) 参照 S(K)

↓ 参照关系　　↓ 被参照关系

R可以参照S

(二)关系代数

1.基本关系代数操作

(1)选择 σ：$\sigma_\theta(R)$

(2)投影 π：自动去重 $\pi_\theta(R)$

(3)并 $\cup$：要求R和S必须具有相同个数的属性且对应属性的值域相容 $R\cup S$

(4)差 −：要求同并 $R-S$

(5)笛卡尔积 $\times$：无条件连接(拼接) $R\times S$

(6)重命名：
- $\rho_{B\leftarrow A}(R)$：把关系R的属性A更名为B
- $\rho_S(R)$：把关系R更名为S
- $\rho_{S(A_1,A_2,\cdots A_n)}(R)$：把关系R更名为S，并把R的全部属性更名为$A_1,\cdots A_n$

2.派生关系代数操作

(1)交 $\cap$：要求同并；可以等价变换成 $R-(R-S)$

(2)内连接
- θ连接 $R\bowtie_\theta S$ (θ是连接条件，含义同σ中的θ)
  结果中包含R和S全部属性，同名加前缀
- 自然连接 $R\bowtie S$ 自带条件：同名属性上的等值比较
  从连接结果中去掉重复的同名属性

(3)外连接：连接结果中保留R或S中的不满足连接条件的元组
- 左外连接 $R ⟕_{(\theta)} S$：R中不满足条件的也在结果中，对应S属性为空
- 右外连接 $R ⟖_{(\theta)} S$：S　　　R
- 全外连接 $R ⟗_{(\theta)} S$：上面二者兼有

(4)反连接 $R \triangleright_{(\theta)} S$：不满足连接条件的元组进行连接

(4) 除：R÷S：结果中只包含R中的属性，不包含S中的属性

R÷S的结果是使得 S×T ⊆ R 的最大的关系T

一个例子：选修了所有课程的学生：All(sno, Cno) ÷ All(Cno)

3. 扩展关系代数操作

(1) 分组操作 $\gamma_{L;agg}(R)$ —
- L：分组属性列表，用逗号分隔
- agg：聚集函数表达式列表，用逗号分隔

例：sum(score)→Totalscore 计算score的和，结果命名为Totalscore

(2) 赋值操作：避免过于冗长的查询表达式

(三) 关系演算：

1、元组关系演算：

(1) 表达式形式：{t | P(t)}　　t 结果元组，P(t) 条件

t[A]：元组t中属性A的值

t∈R：t是关系R中的元组

结果是所有使得P为真的元组t的集合。

(2) 例子：① 计算机系和数学系的学生 {t | t∈Student ∧ (t[Sdept]='CS' ∨ t[Sdept]='MA')}

② 全体学生的学号和姓名 {t | ∃s∈Student (t[Sno]=s[Sno] ∧ t[Sname]=s[Sname])}

2、域关系演算：

(1) 表达式形式：{($x_1, x_2, \cdots, x_n$) | P($x_1, x_2, \cdots, x_n$)}　　($x_1, \ldots, x_n$) 域变量，P 域关系演算公式

($x_1, x_2, \cdots, x_n$) 域变量 $x_1, x_2, \cdots, x_n$ 构成的元组

结果是所有使 P($x_1, x_2, \cdots, x_n$) 为真的元组 ($x_1, x_2, \cdots, x_n$) 的集合

(2) 例子：① 计算机系和数学系的学生 {(n, m, s, a, d) | (n, m, s, a, d) ∈ Student ∧ (d='CS' ∨ d='MA')}

② 全体学生的学号和姓名 {(n, m) | ∃s, a, d ((n, m, s, a, d) ∈ Student)}

(四) SQL语句

1. 数据定义

(1) 基本数据类型

(2) 创建关系模式 Create table 表名 (属性名 数据类型, …, 声明主键、外键)

例：Create table SC (sno Char(6), cno char(4), grade int,

(3) 声明用户定义完整性约束：primary key (sno, cno). foreign key (sno) references student(sno));

非空 NOT Null，不重复 unique，缺省值：default 缺省值，check (表达式)

检查是否满足条件　只解析不处理

3

(4) 删除关系：Drop table 关系名1, 关系名2

(5) 修改关系模式 (alter table)

① 修改关系名：alter table 原名字 rename to 新名字

② 增加、修改、删除属性

- 增加属性：alter table 表名 add 列名 数据类型；
- 删除属性：alter table 表名 drop 列名；
- 增加表约束：alter table 表名 add constraint 约束名 约束；
- 删除表约束：alter table 表名 drop constraint 约束名；
- 修改属性名：alter table 表名 change 旧属性名 新属性名 属性定义；
- 修改属性定义：alter table 表名 modify 属性名 属性定义

(6) 视图

① 创建视图　create view 视图名 (属性名列表) AS 子查询；

② 修改视图　alter view 视图名 (属性名列表) AS 子查询；

③ 删除视图　drop view 视图名；

④ 查询视图　语句同表查询

2. SQL 数据更新

(1) 插入数据

① 直接插入 insert into 表名 (属性名列表) values (表达式/值列表)；

若没有给出属性名列表 则插入一个完整元组；没列出的属性值为空.

② 插入查询结果：insert into 表名 子查询；

(2) 修改数据

① 基于本关系的数据修改：update 表名 set 属性名1 = 表达式1 ...... where 修改条件；（where 修改条件：可选，仅涉及本关系）

② 基于外部关系的数据修改：只是修改条件中不只有本关系，其它同上

(3) 删除数据

① 基于本关系的数据删除：Delete from 关系名 where 删除条件；（where 删除条件：可选（没有的时候删除所有数据））

② 基于外部关系的数据删除

(4) 数据完整性检查：检查修改的过程中是否破坏了完整性约束

① 实体完整性约束检查：

检查时机：插入元组或修改元组的主键属性值时

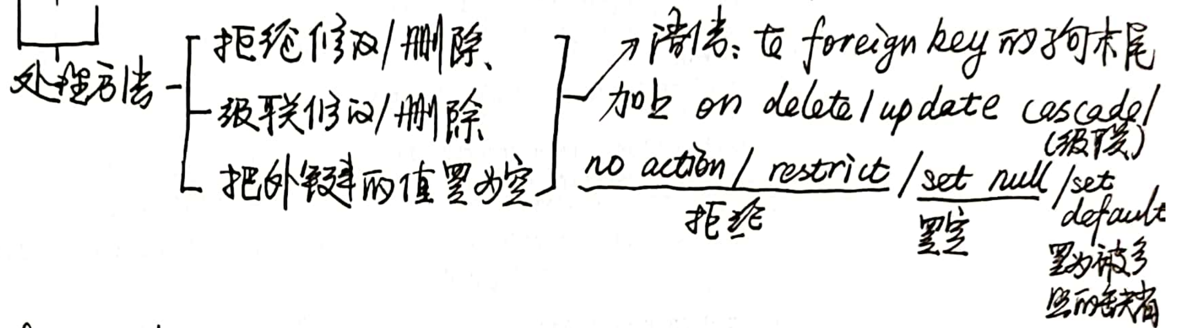
处理方法：主键为空或不唯一都拒绝插入和修改

② 用户定义完整性约束检查

检查时机：修改/插入元组

处理方法：不满足就拒绝

③ 参照完整性检查：



3. 数据查询

核心 select _ from _ where _

(1) 单关系查询

① 投影：select (distinct) 属性名列表 from 关系名;（distinct：去重）

例子：select sname, (2022-Sage) as bd from Student;

（2022-Sage：可以是表达式；as bd：重命名）

② 选择查询：select (distinct) 表达式列表 from 关系名 where 选择条件;

选择条件：
- 表达式比较：<>：不等于
- 范围比较：表达式1 (not) between 表达式2 and 表达式3
- 集合元素判断：表达式1 (not) in (表达式2, …, 表达式n)
- 字符串匹配：字符串表达式 (not) like 模式 (escape 转义字符)
  通配符_：匹配单个字符，通配符%：匹配任意长度的字符串
- 字符串正则表达式匹配
- 空值判断：is Null 不是 = Null
- 逻辑运算符

③ 集合操作：求两个查询语句结果的并、交、差
- 并：查询语句1 union (all) 查询语句2 (有all则不去重)
- 交：Intersect；差：minus/except

说明：查询语句1的结果属性名作为集合操作的结果属性名

(MySQL中不支持交和差：交用 $R \bowtie S$，差用 $\pi(\sigma_{S=Null}(R ⟕ S))$)

④ 查询结果排序

在查询语句的后面加上 order by 列名1 (ASC/DESC), …, 列名n (ASC/DESC)

（ASC：升序(默认)；DESC：降序）

⑤聚集查询

SELECT 聚集函数((distinct) 表达式) from … where …;

※聚集函数不能出现在where子句中(用嵌套查询来实现)

- count(*)：所有元组的数量
- count((distinct) 表达式) (不同的)非空表达式数量
- max/min((distinct) 表达式) 表达式的最大值/最小值
- sum((distinct) 表达式)：(不同)表达式值的和
- avg((distinct) 表达式)：(不同)表达式值的平均值

⑥分组查询

SELECT 分组属性列表, 聚集函数表达式列表 From … where … group by 分组属性列表

原则：根据分组属性分组, 分组属性值相同的为一组；对每个组中的非分组属性进行聚集. (Having 分组筛选条件)(可以用聚集函数)

比如：查询2门以上课程得分超过80的学生的学号及这些课程的平均分

⑤ Select Sno, AVG(Grade) ① from SC ② Where Grade >= 80 ③ Group By Sno ④ Having Count(*) >= 2;

2. 连接查询

(1) 笛卡尔积

select … from 关系名1 (cross join) 关系名2 (…) … , 关系名n

(2) 内连接

<1> select … from 关系名1 (inner) join 关系名2 on 连接条件 (inner 可选)

<2> 当内连接是等值连接且连接属性同名时, 可使用：(连接属性只保留一个副本)

select … from 关系名1 (inner) join 关系名2 using (连接属性列表)

<3> 自然连接：关系名1 natural join 关系名2

<4> 自连接：注意重命名

(3) 外连接：

<1> 左/右/全外连接：关系名1 left/right/full (outer) join 关系名2 on 连接条件 (full：MySQL不支持)

<2> 自然外连接：关系名1 natural left/right/full join 关系名2

3. 嵌套查询

(1) 一些基础概念
- 查询块：一个SELECT-FROM-WHERE
- 嵌套查询：一个查询块中嵌套多个, 内层的称为子查询
- 子查询的类型
  - 不相关子查询(子查询不依赖外层查询)
  - 相关子查询(子查询依赖外层查询)

(2)嵌套查询的写法：

① 在集合判断条件中使用子查询：使用(not) in

② 在比较条件中使用子查询：表达式 比较运算符 (all/any/some)(子查询)

（all/any/some → 当子查询结果只有一个值时可以不加）

③ 在存在性测试条件中使用子查询：(not) exists (子查询)

（→ 只需使用 select *）

从父查询中依次取出每条元组，拿出在子查询中需要的属性值代入子查询，若子查询结果(不)为空，则把该条元组挑出输出。

<1> 用Exists实现全称量词∀功能：$\forall x\,(P(x)) \Leftrightarrow \nexists x\,(\neg P(x))$

查询选修了全部课程的学生的学号

思路：设t是符合条件的学生元组，$\forall c \in Course(\exists s \in SC,$ 使 $s[Sno]=t[Sno] \wedge s[Cno]=c[Cno])$

$\Leftrightarrow \nexists c \in Course(\nexists s \in SC,$ 使 $s[Sno]=t[Sno] \wedge s[Cno]=c[Cno])$

```
SELECT Sno FROM Student WHERE NOT EXISTS(
    SELECT * FROM Course WHERE NOT EXISTS (
        SELECT * FROM SC WHERE SC.Sno = Student.Sno AND SC.Cno
                                          = Course.Cno));
```

<2> 用Exists实现逻辑蕴含→功能：$x \to y = \neg x \vee y$ 例：查询至少选修过CS-001选过的课的学生的学号

思路：设t∈Student是符合条件的元组

$\forall s \in SC, ((s[sno]='cs\text{-}001') \to (\exists x \in SC, (x[sno]=t[sno] \wedge x[Cno]=s[cno])))$

$\Leftrightarrow \nexists s \in SC, ((s[sno]='cs\text{-}001') \wedge (\nexists x \in SC, (x[sno]=t[sno] \wedge x[Cno]=s[cno]))))$

```
SELECT Sno FROM Student WHERE NOT EXISTS(
    SELECT * from SC as SC1 where SC1.sno='CS-001' AND not exists(
    SELECT* from SC as SC2 where SC2.sno= Student.sno and sc2.cno= sc1.cno));
```

④ 子查询结果作为派生关系：FROM(子查询)

要求：子查询必须是独立子查询，派生表要重命名

例：选修了2门以上课程的学生的学号和选课数

```
SELECT Sno, T.Amt FROM (SELECT Sno, count(*) AS Amt FROM SC
Group By Sno) AS T WHERE T.Amt >= 2;
```

⑤ 在WITH子句中使用子查询

例：问题同上

```
With T as (select sno,count(*) as Amt from SC Group By Sno) SELECT
sno, Amt From T where Amt >=2;
```

三、数据库设计

(一) 概念数据库设计

1. 数据库设计的过程

(1) 概念设计：设计数据库的(抽象)概念模型。

(2) 逻辑设计：设计数据库的概念模式.

(3) 物理设计、设计数据库的内模式

2. ER模型 → ER图
   ↳ 将现实世界抽象成实体及实体间的联系。

(1) 实体(普通实体)与属性 —
- 简单属性 (椭圆)
- 复合属性
- 多值属性 (双椭圆)
- 派生属性 (虚线椭圆)
- 键属性 (下划线)

实体 → 矩形 □

实体 vs 实体型 vs 实体集
↓ 对象　↓ 类型　↓ 对象集合(当前存储在数据库中的某实体型的实例的集合)

(2) 联系

① 联系 vs 联系型 vs 联系集
↓ 定义　↓ 具体类型　↓ DB中当前存储的联系型的实例的集合

② 表示形式：◇ 菱形

③ 连线(联系型的约束)
- 线上的数 — 基数比：实体型参与到联系型中的最大基数
- 存在依赖约束(参与度约束)：参与到联系型中的最小基数，部分参与用单线，全部参与用双线

④ 联系型可以具有属性，N:1联系型的属性可被移到N一元的实体中.

⑤ 多元联系：一个n元联系和n个二元联系所表示的意义通常是不同的.

(3) 弱实体型：一用弱实体型的部分键 + 标识实体型的主键来区分不同弱实体.

① 弱实体型：没有键属性的实体型 (双矩形)

② 标识实体型/属主实体型：弱实体型依赖的用于区分的实体 (双菱形)] — 通过标识联系型关联
　↑ 弱实体型全部参与

③ 部分键：区分和同一标识实体相关联的弱实体的属性集合. (虚下划线椭圆)

(二) 逻辑数据库设计

1. 在逻辑数据库设计阶段，能够将ER模型转换为关系数据库模式.

(1) 实体型的转换

① 无多值属性时 — 复合属性
- 实体型的名 — 关系名
- 实体型的属性集 → 关系的属性集
- 实体型的键属性 → 主键
- 实体 → 元组

② 复合属性：把最低组成成分作为关系的属性

③ 多值属性：单独建立关系，其中包含实体型的主键并建立外键约束.

(2) 弱实体型的转换

① 弱实体型的名 → 关系名

② 弱属性 ∪ 属主键 → 关系属性集

③ 属主键 ∪ 弱部分键 → 关系主键

④ 弱实体型关系到属主实体型关系的外键约束.

(3) 联系型的转换

- M:N型：(1) 名 → 名
  (2) 1的主键 ∪ 2的主键 ∪ 联系型属性集 → 关系属性集
  (3) 1的主键 ∪ 2的主键 → 主键（两个外键约束）
- N:1型：(1) 2的主键和联系型的属性并入1的属性集
  (2) 建立1到2的外键约束
- 1:1型：同N:1
- 二元自联系型：note对重名属性重命名
- 相似联系型不转换

2、关系数据库规范化理论 —— 评价一个关系数据库模式设计的"好坏"

(1) 函数依赖

① 定义：R(U)为属性集U上的关系模式，$X, Y \subseteq U$，若对于R(U)的任意关系实例中的任意2个元组t_1和t_2，若由$t_1[X]=t_2[X]$能推出$t_1[Y]=t_2[Y]$，则称X函数决定Y，或Y函数依赖于X，记作$X \to Y$.

② 函数依赖的类型

<1> 若$Y \subseteq X$，则$X \to Y$是平凡函数依赖

<2> 完全函数依赖：若$X \to Y$，且对$\forall X' \subseteq X$，都有$X' \not\to Y$，则称Y完全函数依赖于X，记作$X \xrightarrow{f} Y$

部分函数依赖：若$X \to Y$，且$\exists X' \subseteq X$，使得$X' \to Y$，则称Y部分函数依赖于X，记作$X \xrightarrow{p} Y$

<3> 传递函数依赖：若$X \to Y$，$Y \to Z$且$Y \not\subseteq X$，$Y \not\to X$，则称Z传递函数依赖于X，记作$X \xrightarrow{t} Z$

③ 用公理系统Armstrong推导其他函数依赖 → 正确且完备的

<1> 推理规则
- 自反律：若$Y \subseteq X \subseteq U$，则$F \models X \to Y$
- 增广律：若$F \models X \to Y$，则对$\forall Z \subseteq U$，有$F \models XZ \to YZ$
- 传递律：若$F \models X \to Y$且$F \models Y \to Z$，则$F \models X \to Z$

<2> 导出规则
- 合并规则：若$F \models X \to Y$且$F \models X \to Z$，则$F \models X \to YZ$
- 伪传递规则：若$F \models X \to Y$且$F \models WY \to Z$，则$F \models XW \to Z$
- 分解规则：若$F \models X \to Y$，则对$\forall Z \subseteq Y$，有$F \models X \to Z$

④ 用属性集闭包推导其他函数依赖　　根据F里的FD，求X决定的所有属性

<1> 定义：设R(U,F)是一个关系模式，集合 $\{A \mid A \in U, F \models X \to A\}$ 称为X关于F的闭包，记作 X_F^+

定理：$F \models X \to Y \Leftrightarrow Y \subseteq X_F^+$　（把逻辑蕴含判定问题化成属性集闭包计算问题）

<2> 函数依赖集的闭包：F逻辑蕴含的全部函数依赖的集合叫做F的闭包，记作 F^+

<3> 函数依赖集的覆盖：若 $G^+ \subseteq F^+$，则称F覆盖G

<4> 等价函数依赖集：若 $F^+ = G^+$，则F与G等价。$F^+ = G^+$（判断等价的方法1）$\Leftrightarrow F^+ \subseteq G^+$ 且 $G^+ \subseteq F^+$（方法2）

<4> 函数依赖集的最小覆盖

最小覆盖是满足下列3个条件的F的等价函数依赖集

(I) 不存在冗余函数依赖（不能去掉一个后和原来等价）

(II) 函数依赖左部不存在冗余属性

(III) 函数依赖右部仅包含1个属性.

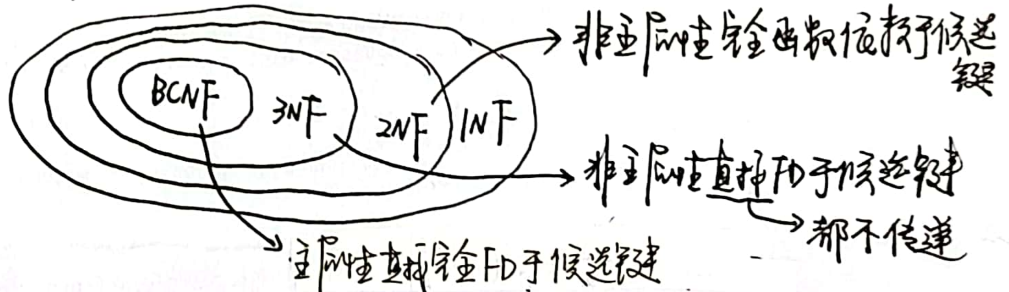
如何计算最小覆盖：

(I) 分解函数依赖，使右侧只有1个属性

(II) 删除函数依赖左部的冗余属性

(III) 删除冗余函数依赖

(2) 关系模式的范式



① 非主属性：不参与到任何候选键的属性

② 范式之间的关系

1NF
↓ 消除非主属性对候选键的部分FD
2NF
↓ 消除非主属性对候选键的传递FD
3NF
↓ 消除主属性对候选键的部分FD和传递FD
BCNF

③ 注意：关系模式的规范化程度并不是一定越高越好

(3) 关系模式分解

判断方法：R(U,F) 的分解 $\rho = \{R_1(U_1), R_2(U_2)\}$ 是无损连接 $\Leftrightarrow$ $F \models U_1 \cap U_2 \to U_1$ 或 $F \models U_1 \cap U_2 \to U_2$

① 两个准则

- 无损连接性：令 $\rho = \{R_1(U_1) \cdots R_n(U_n)\}$ 是对关系模式R(U)的一个分解，若对于R的任意实例r均有 $r = \pi_{U_1}(r) \bowtie \pi_{U_2}(r) \bowtie \cdots \bowtie \pi_{U_n}(r)$，则是无损的
- 函数依赖保持性：设R(U,F)是关系模式，U是属性集，F是函数依赖集

函数依赖的投影：对于 $V \subseteq U$，F在V上的投影是 F^+ 中满足 $X \cup Y \subseteq V$ 的函数依赖 $X \to Y$ 的集合

什么是保持函数依赖的分解？ F ⇒ F_1 F_2 … F_n　若 $F_1 \cup F_2 \cdots \cup F_n$ 覆盖F，则是

(三) 物理数据库设计

四、数据库内核

(一)存储管理

1.面向磁盘的数据库存储

(1) 为什么不用OS直接管理数据库存储?

当缺页时需要OS的调页，但OS何时调页是一个全局考量，会有延迟

(2) 数据库的文件存储

(1) 将一个数据库存储为一个或多个文件.

(2) 每个文件包含多个页.

→ 有唯一的标号(页号) —间接层→ 页的物理地址

(3) 存储管理器:

DBMS的存储管理器负责管理数据库文件 —
- 记录页中元组的读写
- 记录页中的空闲空间

2.面向行的存储

(1) 字符串的表示 —
- char(n)型：若字符数小于n，则后面用空字符补全
- varchar(n)型：在组头部先存串长，再存串的内容

字节串的表示 —
- BINARY(n)：〃　〃　0补全
- VARBINARY(n)：〃

(2) 元组存储：元组头＋数据 —— 由元组的所有属性值拼接而成
- 通常按定义关系模式时指定的属性顺序存储属性值
- 每个属性值在元组中存储位置的偏移量是4字节或8字节的倍数.
- (当元组是变长元组时)的布局：定长属性值和变长属性值分别置于元组两端，元组头后紧跟指针数组，指向每个属性值.

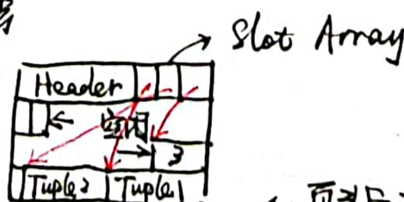
元组头：记录元组的元数据　存在系统目录里
- 指向元组所在关系的模式定义的指针
- 元组的长度
- 元组的最后修改时间
- 元组的可见性(并发控制)
- 元组哪些属性值非空.

(3) 页布局：页头＋页数据

页头：记录页的元数据
- 页的大小
- 页的校验和、DBMS版本、数据压缩方法
- 页的可见性(并发控制)

页数据：组织方法
- 面向元组的组织方法
- 日志结构的组织方法

① 面向元组的组织方法 —— 分槽页



<1>每个元组占一个槽
- 槽从后向前布置在页的末尾
- 每个槽的起始位置的偏移量是4B或8B的倍数

<2>页头后放槽数组

数组第i个非空元素存储第i个槽的起始位置的偏移量.

<3>槽的元数据存储在页头中
- 槽的数量：用于确定下一个空闲槽的编号
- 最后一个槽的起始位置的偏移量：用于确定新元组的插入位置

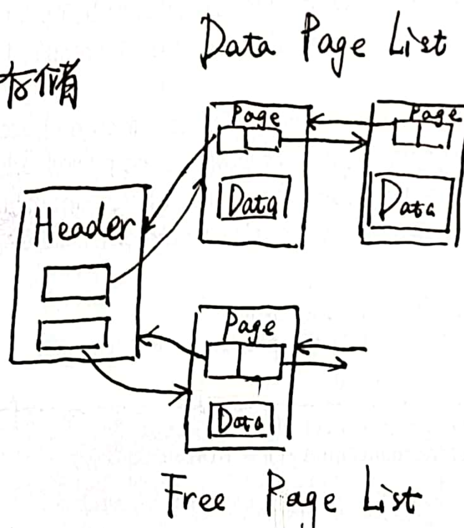
<4>记录号（DBMS为一个关系中的每个元组分配的唯一记录号）
- f1：（页号，槽号）
- f2：唯一的整数作为记录号（再通过间接层映射成页号，槽号）

<5>缺点：空碎片化（浪费磁盘空间、增加磁盘I/O）——定期回收

② 日志结构存储 //了解

(4) 文件组织

① 堆文件组织：堆文件中的元组以任意顺序存储
- 堆文件中页的组织方法
  - 基于链表的组织方法 — 数据页 / 空闲页 / 头页
  - 基于页目录的堆文件页组织方法：页目录中记录每个数据页的位置和空闲空间信息



② 顺序/有序文件中的元组按排序键的顺序存储（除非排序键是主键，否则用的少）

③ 哈希文件组织

3. 缓冲区管理：缓冲区管理器负责在Disk和MM之间复制文件页

(1) 缓冲池：DBMS将可用内存区域划分为页数组，称为缓冲池
缓冲池中的页称为页框

(2) 缓冲池的设计：页表
↳ 记录Buffer Pool中当前有哪些页及它们在内存中的地址
把页号映射为该页所在页框的地址

① 页框的元数据
- pin_count：页框中的页当前被请求但未释放的次数，即引用计数（时时）
- dirty（脏位）：读入Buffer Pool后是否被修改过

② 缓冲区管理器的功能

<1>请求页
- 已在Buffer Pool里：pin_count++，返回页框地址
- 不在
  - 在池中选一个pin_count=0的页框，加1，驱逐旧页（脏页写回），调入新页
  - 没有pin_count=0的，等待直到释放池中页面

<2> 释放页：pin_count--;

<3> 修改页：把dirty置为真.

③ 页替换策略

- 最近最少使用(LRU) → 用时间戳记录Buffer Pool中每个页框的最后访问时间，替换时换掉pin_count=0且最旧的页（维护一个堆）
- 时钟替换策略(CLOCK)
- 先进先出(FIFO)
- 最近最多使用(MRU)

(3) 缓冲区管理器有预取机制

(4) 缓冲池与VM比较

① 相同点：访问内存不够用；调页；页替换

② 不同点：①DBMS更准确地预知对页的访问顺序（更好地选择被替换的页，更好地预取）

②DBMS必须具备强制写回的能力

4. 面向列的存储 / 柱状存储

(1) 把关系的每个属性单独存储
- 当查询仅涉及少量属性时，不需要读无关属性，减少I/O
- 缓存中相邻字节来自于同一个属性，缓存效率高
- // 压缩效率高，SIMD指令向量化计算

(2) 适合于联机分析处理，不适用于联机事务处理

(1) 元组重构代价高

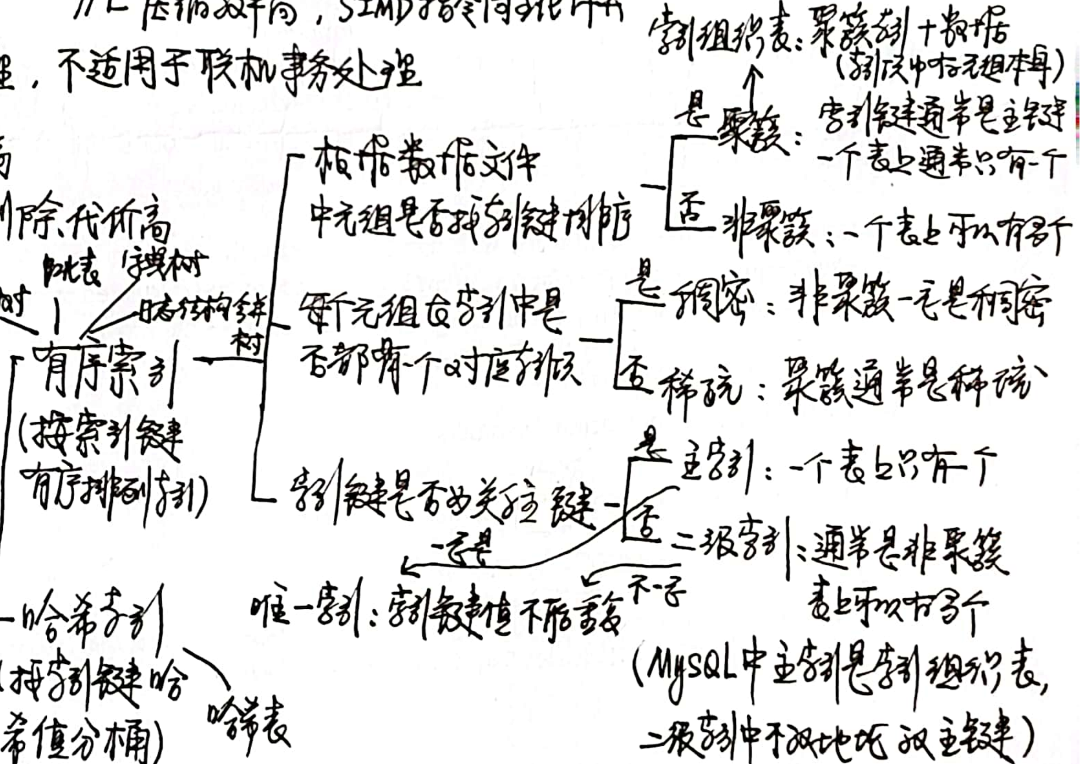
(2) 元组修改和删除代价高

(3) 解压代价

(二) 索引结构

1. 索引的分类 | 概念

(1) 按照索引的实现方式



(2) 有序索引中的概念 (索引键)

索引键：索引根据一组属性来定位元组，索引记录了元组的索引键值与元组地址的对应关系

索引项：索引中的(键值，地址)对，按索引键值排序

(3) 哈希索引中的概念

h: 哈希函数

键为K的索引项属于编号为h(K)的桶.

只支持等值查找

(4) 索引的SQL语句

① 创建主索引：声明主键时自动建立
② 创建二级索引：create index 索引名 on 关系名(索引键)
③ 创建唯一索引：建表时用Unique声明会自动创建唯一索引，或者在②中index前加unique
④ 创建外键索引：声明外键时自动
⑤ 删除二级索引：DROP INDEX 索引名 on 关系名
⑥ 删除主索引：Drop index 'Primary' on 关系名（删除后重新组织元组）

2. 基于哈希的索引结构：哈希表（外存哈希表）

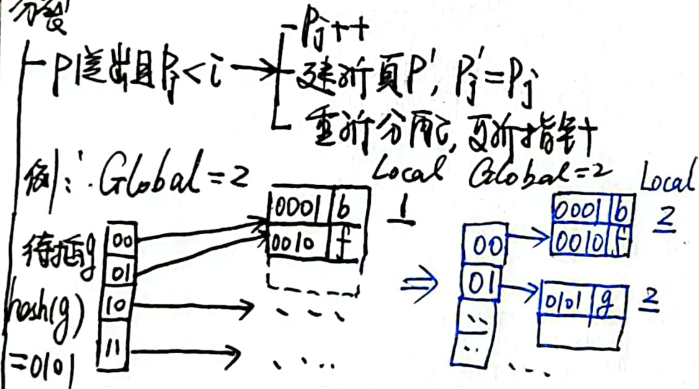
- 静态哈希表：桶的数量固定不变
- 动态哈希表：桶的数目可变，每个桶中索引项所占存储大约1个页

(1) 可扩展哈希表（索引项放在Hash(K)前i位的桶）

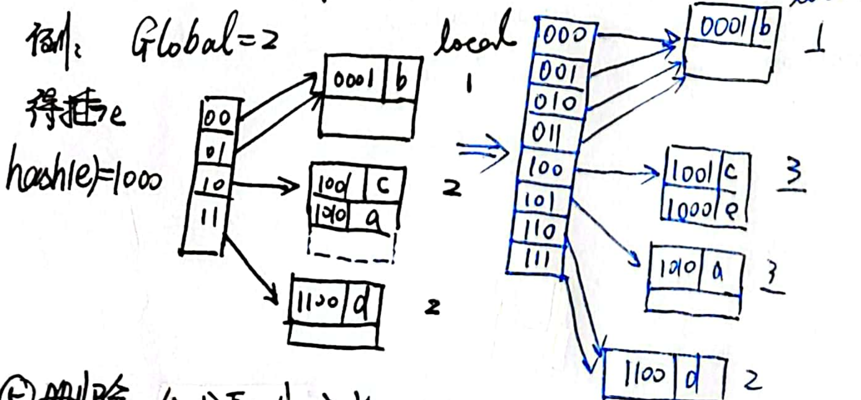
① 用i表示全局深度，一共有2^i个桶
② 每个桶无溢出页，桶存得下可以共用 ⟺ 该页对应i
③ 每个页记录一个局部深度j，该页中前j位的hash(k)相同

④ 插入索引项时：先找往哪插，不够多插就分裂

分裂
- P溢出且$P_j<i$ → P_j++；建新页P'，$P'_j=P_j$；重新分配，更新指针



- P溢出且$P_j=i$ → i++；桶翻倍；更新

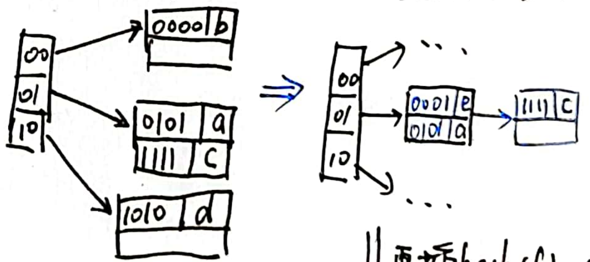


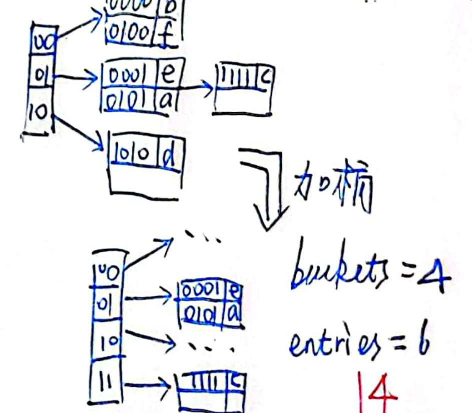
⑤ 删除：先找再删无合并

#buckets 桶数
#entries 项数(2) 线性哈希表

① 一共n个桶，每个桶中指针指向页链表
② 设一个页中能存b个索引项，一个桶一页，则线性哈希表最多存θbn个项（0<θ<1）
③ 先看桶有几位，然后看哈希值的后这些位决定在哪个桶，如果没有这个桶就少看1位
④ 插入，先找往哪插，项数加1，是否≤θbn，是则完成，否则加一个桶，重新分配

例：待插hash(e)=0001，θ=0.85





(3) 可扩展Hash表 vs 线性Hash表

	可扩展Hash表	线性Hash表
桶数	2^i	n
是否有溢出	无	有
哈希方案	hash(K)的前i位	hash(K) mod 2m 或 hash(K) mod m
页分裂条件	页发生溢出	entries > θbn
增加桶方法	i++，桶翻倍	桶数+1

3. B+树

(1) 性质：

① 除根节点外每个节点至少半满：$M/2-1 \le keys \le M-1$（M路平衡树）

② 每个节点恰好放入1个页

③ 叶节点包含一个通常按索引键排序的索引项数组和指向右叶的指针

(2) 查找

① 单值查找：先找在哪个叶上，再在这个叶上找

② 区间查找：查找[L, U]，先找≥L的第1个键的索引，扫描其右侧，≤U并输出，否则终止

(3) 插入

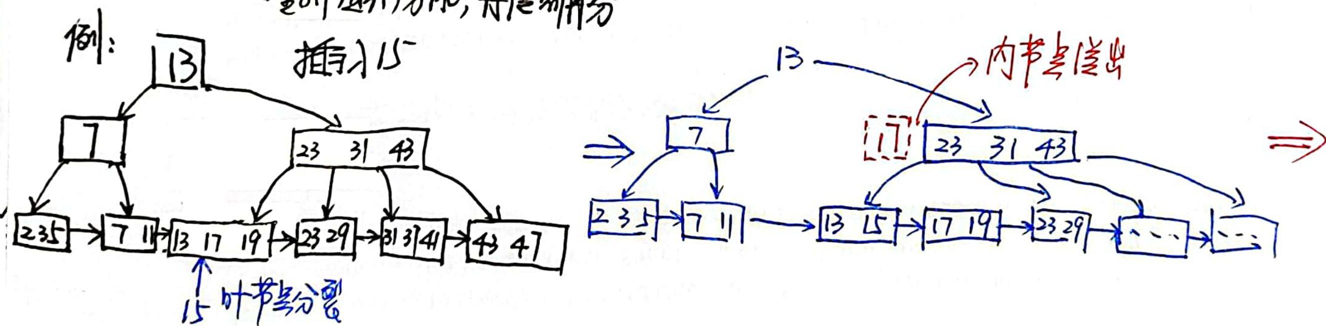
① 找到待插入位置插入，不溢出结束，溢出则分裂

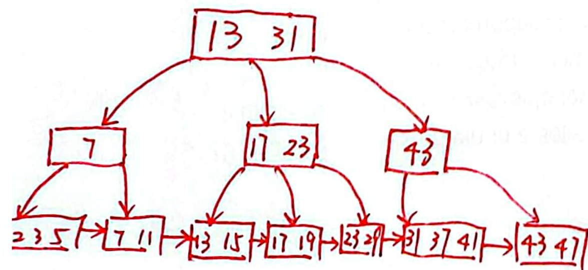
② 叶节点分裂
- 在溢出页右边再建一个叶，溢出页的右一半给新页。
- 新页中的最小键（索引）插到父节点，重新组织分配，再溢出则再分

③ 内节点分裂
- 建一个新节点，同溢出节点的右半部分给新节点（中间键不给，给上一级节点）
- 重新组织分配，再溢出则再分

若是根节点溢出，则会增加一个新根节点

例：插入15





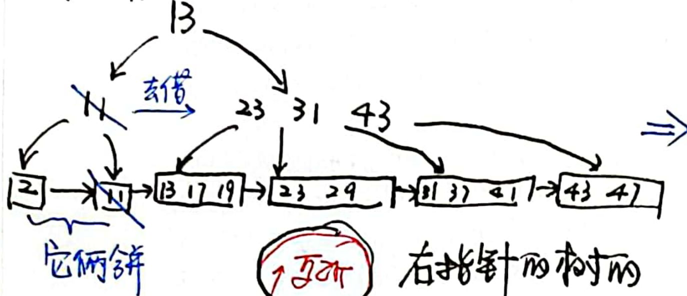
(4) 删除

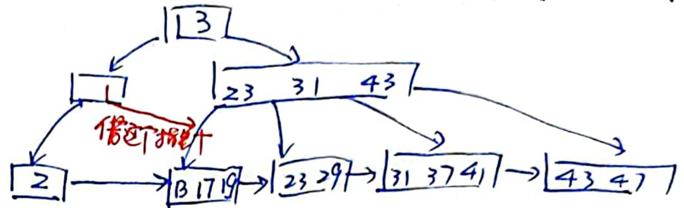
① 先找到再删除，若不满足"半满"再处理

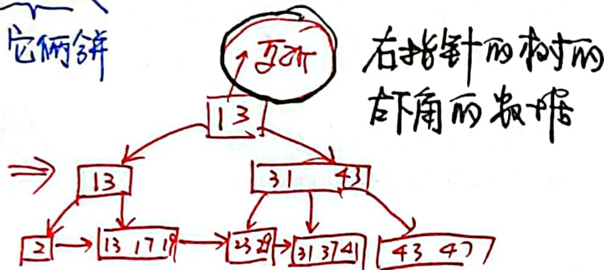
② 处理叶节点 — 尝试从兄弟借使二者均半满 → 互换、重新分配
　　　　　　 — 如果借不到则与其兄弟节点合并 — 与左兄弟合并，则从父节点中删指向自己的指针和键 / 与右兄弟合并，则从父节点中删指向右兄弟的指针和键 — 删除后判断父节点是否半满，否则继续处理（若根直接删）

内节点

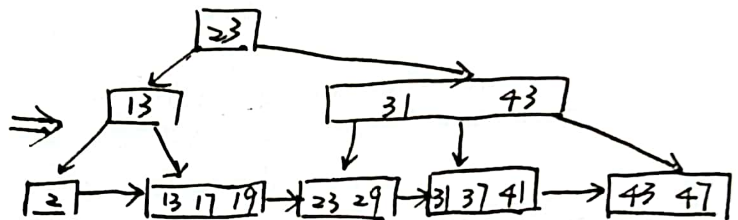
例：删除11







右指针所指树的最左下角的数据



(三) 查询执行与优化

SQL查询
↓ SQL解析器
关系代数表达式
↓ 查询优化器 —— 查询优化
查询执行计划
↓ 查询执行器 —— 查询执行
查询结果

第一部分：查询执行

1. 基本记号与约定

N：关系R的元组数

M：缓冲池中可用内存页数

B：每块最多存B个元组

B(R)：R的块数 $\lceil N/B \rceil$

T(R)：关系R的元组数

V(R.A)：关系R的属性集A的不同值的个数

算法分析时的约定

*输出结果时产生的I/O不计入算法的I/O代价

*输出缓冲区不计入可用内存页数.

2. 查询执行时的各个算法

算法大名	算法小名	算法简明	I/O	对M的要求
外存排序	两趟多路外存归并排序	把R划分成$\lceil B(R)/M \rceil$个归并段，每次调入内存M块(一个归并段)内排序后写回Disk，调入每个归并段的第一块后开始归并	$3B(R)$	$B(R) \le M^2$ ($\frac{B(R)}{M} \le M$)
	多趟多路外存归并排序	$B(R) > M^2$时使用	$(2k-1)B(R)$ k趟数	
选择操作	基于扫描的选择算法	依次扫描每一块的每个元组，符合条件则输出	$B(R)$	$M \ge 1$
不带去重的投影算法 (基本一致)	基于哈希的选择算法	前提条件：选择条件的形式是K=v，关系R采用哈希文件组织形式，属性K是R的哈希键。根据hash(v)确定元组所在的桶，在这个桶里找键值等于v的元组	$\lceil \frac{B(R)}{V(R,K)} \rceil$ (近似)	$M \ge 1$
	基于索引的选择算法	前提条件：选择条件的形式是K=v或$l \le K \le u$，K上有索引。直接在索引上搜索	聚簇 $\lceil \frac{B(R)}{V(R,K)} \rceil$ 非聚簇 $\lceil \frac{T(R)}{V(R,K)} \rceil$	$M \ge 1$
去重操作 $\delta(R)$	一趟去重算法	把每一块读进来建立内存查找结构，新进来的块看在不在结构里，不在就输出并加到查找结构里，在就跳过	$B(R)$	$M-1 \ge B(\delta(R))$
	基于排序的去重算法	思想同外排，归并时相同的只输出一个，舍弃其他元组	$3B(R)$	$B(R) \le M^2$
	基于哈希的去重算法	哈希分桶到(M-1)个桶里，送回Disk，对每个桶用一趟去重	$3B(R)$	$B(R) \le (M-1)^2$ ($\frac{B(R)}{M-1} \le M-1$)
集合差操作	一趟集合差算法	R-S，S较小，在(M-1)页能放下，在这(M-1)页构造S的内存查找结构，对R逐块读入后逐条去结构中找，找不到则输出	$B(R)+B(S)$	$B(S) \le M-1$
	基于哈希的集合差算法	把R和S分别哈希到(M-1)个桶里，桶号相同的进行一趟集合差，最后并起来	$3[B(R)+B(S)]$	$B(S) \le (M-1)^2$
	基于排序的集合差算法	先创建归并段，归并时去判断元组t是否$\in R$且$\notin S$	$3[B(R)+B(S)]$	$B(R)+B(S) \le M^2$

算法大名	算法小名	算法流程	I/O	对M的要求
连接操作的执行 R(X,Y)⋈S(Y,Z)	一趟连接算法（若内存查找结构是哈希表，则相当于哈希连接）	假设B(S)≤B(R)，则S.Y在(M-1)页中建内存查找结构，然后读R的每一块每一条，符合条件的就连接起来输出。	B(R)+B(S)	B(S)≤M-1
	基于元组的嵌套循环连接	两层循环，一条一条比较。 for S的每个元组s do（外关系） for R的每个元组r do（内关系） if … then	T(S)(T(R)+1) (T(S)+T(S))T(R)	M≥2

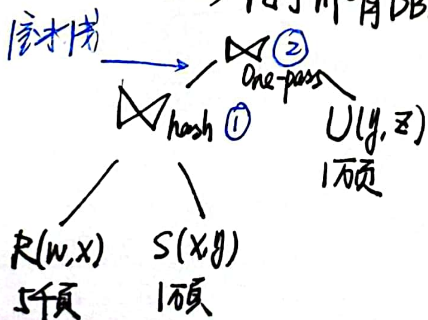
	基于块的循环嵌套连接	for 外关系S中的每M-1块 do
		建查找结构
		for 内关系R中的每块 do
		读入缓冲池
		for 每条元组r do
		for 查找结构中能与r连接的元组s do
		连接，输出
		假设B(S)<B(R)
		四层循环：读S块→读R块→读R元组→查找结构中每个S
			$B(S)+\frac{B(S)}{M-1}B(R)$	M≥2

	排序归并连接	创建归并段，归并	3[B(R)+B(S)]	$B(R)+B(S)\leq M^2$
	Grace哈希连接	先把R和S分别哈希到(M-1)个桶里，逐桶做一趟连接算法，最后并起来	3[B(R)+B(S)]	$B(S)\leq (M-1)^2$
	索引连接	假设S上有Y的索引。把R逐块读入缓冲池，逐条元组去索引上看是否可以连接并。	聚簇索引：$B(R)+\frac{T(R)B(S)}{V(S,Y)}$ 非聚簇：$B(R)+\frac{T(R)T(S)}{V(S,Y)}$	M≥2

3、查询计划的执行方法

(1) 物化执行：自底向上执行，中间操作的结果写入临时关系文件。
→ 缺点：物化临时关系时增加了查询的代价；用户获得查询结果的时间延迟↑

(2) 流水线执行/火山模型：一个操作的结果直接传给流水线下一个操作。
→ 几乎所有DBMS都采用该方法



M=101
R,S,U上均无索引且无序
B(R⋈S)≤50

I/O 3(B(R)+B(S)) =45000

I/O B(U)=10000

执行时 ① 哈希过程用101页，1页输入缓冲，100页分桶，逐桶连接时用50页给R建查找结构，1页做输入缓冲，剩下50页做输出缓冲 → 存放缓冲区

② 一趟连接，用到B(R⋈S)+1页内存

如何实现流水线？用迭代器模型。

第二部分：查询优化

1. 基本概念

(1) 查询优化的两个阶段 —
- 逻辑查询优化：生成更高效的逻辑查询计划 → 关系代数表达式
- 物理查询优化：生成优化的物理查询计划 → 带有"如何执行"的关系代数表达式

↓ 实际上DBMS并不区分

2. 逻辑查询优化

(1) 基于代价的查询优化（所有DBMS采用的找代价最低的计划的方法）
- 计划枚举：生成等价查询计划 —
  - 关系代数表达式等价变形
  - 选择恰当的选择 ↗ ∈
- 代价计算

(2) 等价关系代数表达式

① 既满足交换律又满足结合律的关系代数操作：$\times$ $\bowtie$ $\cap$ $\cup$
（注意θ连接只满足交换律，不满足结合律）

② 关于选择的等价变换与选择下推 ⟶ 可以尽早过滤掉与结果无关的元组

<1> $\sigma_{\theta_1 \wedge \theta_2}(R)=\sigma_{\theta_1}(\sigma_{\theta_2}(R))=\sigma_{\theta_2}(\sigma_{\theta_1}(R))$

$\sigma_{\theta_1 \vee \theta_2}(R)=\sigma_{\theta_1}(R)\cup\sigma_{\theta_2}(R)$

<2> $\sigma_\theta(R\cup S)=\sigma_\theta(R)\cup\sigma_\theta(S)$

<3> $\sigma_\theta(R-S)=\sigma_\theta(R)-S=\sigma_\theta(R)-\sigma_\theta(S)$

<4> $\sigma_\theta(R\cap S)=\sigma_\theta(R)\cap S=R\cap\sigma_\theta(S)=\sigma_\theta(R)\cap\sigma_\theta(S)$

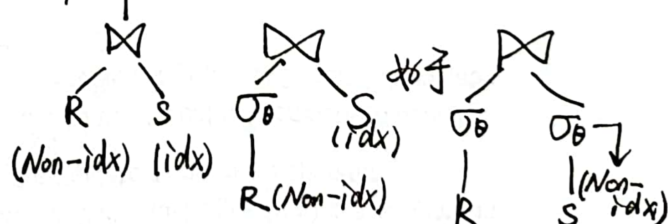
<5> $\sigma_\theta(R\times S)=R\bowtie_\theta S$，若S没有θ中的属性，R中有θ中所有…
则 $\sigma_\theta(R\times S)=\sigma_\theta(R)\times S$

<6> $\sigma_{\theta_1}(R\bowtie_{\theta_2}S)=R\bowtie_{\theta_1\wedge\theta_2}S$，若S没有$\theta_1$中…R中…
则 $\sigma_{\theta_1}(R\bowtie_{\theta_2}S)=\sigma_{\theta_1}(R)\bowtie_{\theta_2}S$

<7> 如果R和S均包含θ中使用的全部属性
$\sigma_\theta(R\bowtie S)=\sigma_\theta(R)\bowtie S=R\bowtie\sigma_\theta(S)=\sigma_\theta(R)\bowtie\sigma_\theta(S)$

原则：
<1> 选择度高的选择优先做
<2> 把复杂的选择操作分解再下推
<3> 当可以向多个分支下推时要考虑：
(I) 索引会影响选择下推的方案
(II) 选择操作结果上无索引

例：



③ 关于投影的等价变换与投影下推 ⟶ 降低元组的大小

<1> $\pi_{L_1}(\pi_{L_2}(R))=\pi_{L_1}(R)$ $(L_1\subseteq L_2)$

<2> $\pi_L(\sigma_\theta(R))=\pi_L(\sigma_\theta(\pi_M(R)))$
M中既包含L中的属性又包含θ中使用的属性

<3> $\pi_L(R\cup S)=\pi_L(R)\cup\pi_L(S)$

Pay attention: $\pi_L(R\cap S)\neq\pi_L(R)\cap\pi_L(S)$
$\pi_L(R-S)\neq\pi_L(R)-\pi_L(S)$

有些情况下，投影下推会使失去查询优化的机会：如投影操作的结果上无索引。

<4> $\pi_L(R\times S)=\pi_L(\pi_M(R)\times\pi_N(S))$

$M=L\cap R$, $N=S\cap L$

<5> $\pi_L(R\bowtie_{(\theta)} S)=\pi_L(\pi_M(R)\bowtie_{(\theta)}\pi_N(S))$

$M=L\cap R$ 以及连接属性(R中)

$N=S\cap L$ 以及连接属性(S中)

(3) 逻辑查询计划的代价核算

① 逻辑查询计划的代价用执行计划过程中产生的中间结果的元组数来度量.

② 物理查询计划的代价比逻辑查询计划的代价更可靠. DBMS实际上直接枚举物理查询计划, 不区分两个优化. ——如何度量/估计?

(4) 基数估计

① 目的: 估计查询结果的元组数

② 要求: 准确、易计算、逻辑一致 —— 单调性: 输入↗ 结果估计值↗；顺序无关性.

③ 系统目录中记录着基数估计所用的DB统计信息 —— $T(R)$: 关系R的元组数；$V(R,A)$: 关系R的属性A的不同值个数

④ 基本假设 —— 关系中每个属性取值均服从均匀分布；关系的所有属性相互独立.

⑤ 各种操作的基数估计:

<1> $T(R\times S)=T(R)T(S)$

<2> $T(\pi_L(R))=$ —— 不含去重 $T(R)$；含去重 $V(R,L)$

<3> $S=\sigma_{A=c}(R)$

$T(S)=T(R)/V(R,A)$

$S=\sigma_{A\neq c}(R)$

$T(S)=T(R)$ 或 $T(S)=T(R)-\frac{T(R)}{V(R,A)}$

$S=\sigma_{A>c}(R)$ 或 $S=\sigma_{A<c}(R)$

$T(S)=T(R)/3$ 或 $T(S)=T(R)/2$

若知道A的最值 可以用几何概型

<4> (I) $S=\sigma_{\theta_1\wedge\theta_2}(R)=\sigma_{\theta_1}(\sigma_{\theta_2}(R))=\sigma_{\theta_2}(\sigma_{\theta_1}(R))$

$T(S)=T(R)f_1f_2$, 其中 $f_i=\begin{cases}1/V(R,A) & \theta_i: A=c\\ 1-\frac{1}{V(R,A)} & \theta_i: A\neq c\\ 1/3 & A<c \text{ or } A>c\end{cases}$

(II) $S=\sigma_{\theta_1\vee\theta_2}(R)=\sigma_{\theta_1}(R)\cup\sigma_{\theta_2}(R)=R-\sigma_{\neg\theta_1\wedge\neg\theta_2}(R)$

$T(S)=T(R)-T(R)(1-f_1)(1-f_2)$

(III) $S=\sigma_{\neg\theta}(R)=R-\sigma_\theta(R)$

$T(S)=T(R)-T(\sigma_\theta(R))$

<6> 二路自然连接: $R\bowtie S$

基本假设 —— 连接属性值集合包含假设: 设连接属性为K, 若 $V(R,K)\leq V(S,K)$, 则 $R.K\subseteq S.K$；非连接属性值集合保留假设: 对R中∀非连接属性A, 有 $V(R\bowtie S,A)=V(R,A)$

(Ⅰ) R和S只有一个连接属性Y：$T(R\bowtie S)=\frac{T(R)T(S)}{max(V(R,Y),V(S,Y))}$

(Ⅱ) R和S有2个连接属性X,Y：$T(R\bowtie S)=\frac{T(R)T(S)}{max(V(R,X),V(S,X))\,max(V(R,Y),V(S,Y))}$

(Ⅲ) 推广到n个：分母n个max

<7> 集合操作的基数估计

$T(R\cup S)=\frac{1}{2}(max(T(R),T(S))+T(R)+T(S))$　　原因：$max(T(R),T(S))\leq T(R\cup S)\leq T(R)+T(S)$

$T(R-S)=T(R)-\frac{1}{2}T(S)$　　$T(R)-T(S)\leq T(R-S)\leq T(R)$

$T(R\cap S)=min(T(R),T(S))/2$　　$0\leq T(R\cap S)\leq min(T(R),T(S))$

$T(R\cap S)=T(R\bowtie S)$　　$R\cap S=R\bowtie S$

<8> 去重操作的基数估计

$T(\delta(R))=(T(R)+1)/2\approx T(R)/2$　　$1\leq T(\delta(R))=(T(R)+1)/2\approx T(R)/2$

$T(\delta(R))=min((T(R)+1)/2,\ V(R,A_1)V(R,A_2)\cdots V(R,A_n))$　　$1\leq T(\delta(R))\leq V(R,A_1)V(R,A_2)\cdots V(R,A_n)$

(5) 用直方图近似数据分布

- 等宽：属性值各区间的宽度相同，各区间内属性值的出现次数不同
- 等高：…　　不同　　…　　基本相同
- 压缩：单独记录少量频繁出现的属性值，用等高或等宽近似其他属性。

等高更常用
- 包含高频属性值的区间窄，对高频属性值的频率估计更准确
- 区间边界由数据确定，无需用户指定。

(6) 连接顺序的优化

① $R\bowtie S$中R是左关系，S是右关系
- 一趟连接：左→构建关系；右→探测关系
- 嵌套循环连接：左→外关系；右→内关系
- 索引连接：右→索引关系

② 连接树：左深连接树、右深连接树、浓密树
- 只考虑这个
  - 左深连接树比全部连接树少很多
  - 通常更好（形成完全流水线的流水线）
    - 对内存的需求低
    - 不需要多次构建中间关系

动态规划例题，课件

3、物理查询优化（选执行算法+执行模型）→（查询执行）

(1)确定选择操作的物理执行算法：本质是确定最高效的存取路径

①索引扫描+过滤

$\sigma_{\theta_1 \wedge \theta_2}(R)$

<1>什么时候能用：索引支持θ_1上的查找

<2>怎么用？根据θ_1索引取元组ID，取元组，用θ_2过滤.（存取路径）

<3>什么时候好用：索引是聚簇索引 或 非聚簇索引但满足条件的元组很少

②索引合并

$\sigma_{\theta_1 \wedge \theta_2}(R)$

<1>什么时候能用：索引分别支持θ_1和θ_2上的查找

<2>怎么用（存取路径）：两个索引上分别找满足条件的元组ID，求交得所有满足的元组ID集合，取元组

<3>什么时候好用：I_1和I_2都是非聚簇，满足条件θ_1和θ_2的都多，但同时满足的少

③仅用索引

$\sigma_{\theta}(R)$　记L为该选择操作的后续操作所涉及到的R的属性的集合

<1>……：索引支持θ上的查找，是覆盖L的覆盖索引

<2>……：在索引上查满足条件的索引项后直接取出该元组在L的投影

<3>……：关系R上的聚簇索引不支持选择条件θ，索引是覆盖索引

④顺序扫描 $\sigma_{\theta}(R)$

<1>什么时候用：没有索引支持θ上的查找

(2)确定连接操作的执行算法

适用条件

- 一趟连接：左关系可以全读入缓冲池的可用页面
- 索引连接：左关系较小，右关系在连接属性上建有索引
- 排序归并连接：至少有一个关系已按连接属性排序；多个关系在相同连接属性上做多路连接也适合. eg $R(a,b) \bowtie S(a,c) \bowtie T(a,d)$
- 哈希连接：上面三种都不适用时
- 嵌套循环连接：缓冲区的页面特别少时（耗费时间最多）

(四)并发控制与故障恢复

核心概念：事务(transaction)：数据库上执行的一个或多个操作构成的序列

事务的ACID性质：原子性、一致性、隔离性、持久性

（原子性、持久性→故障恢复保证；一致性→程序员保证；隔离性→并发控制保证）

第一部分：并发控制

1. 调度：一个或多个事务的重要操作序列

(1) 串行调度：一个调度中不同事务的操作没有交叉

(2) 调度的正确性：单独执行每个事务都会将DB从一种一致状态变为另一种一致状态

- 任意串行调度都能保证DB的一致性
- 非串行调度可能会破坏数据库的一致性
  - 脏写：事务T_2提交前，T_2写过的X又被T_1写
  - 脏读：事务T_1提交前，T_1写过的X被T_2读取
  - 不可重复读：一个事务在两次读之间没有修改，但是读取的值不同。
  - 幻读

(3) 等价调度：如果两个调度在∀DB实例上的效果都相同，则这两个调度等价

(4) 可串行化调度：一个调度等价于一个串行调度，则它是可串行化调度

└ 无脏写、脏读、不可重复读、幻读

↓

缺点：并发度低，降低隔离级别来提高并发度

(5) 四种隔离级别：从上至下隔离级别增大

名称	说明	脏读	不可重复读	幻读
读未提交(Read Uncommitted)	未提交事务的修改对其他事务可见	✓	✓	✓
读提交(Read Committed)	只有已提交事务的修改才对其他事务可见	×	✓	✓
可重复读(Repetable Read)	若事务T修改X，则T只能读该事务读到的X值都等于启动时所读	×	×	✓
可串行化(Serializable)		×	×	×

(6) 支持可串行化隔离级别的DBMS实施的都是冲突可串行化

① 冲突：两个操作属于不同事务，涉及相同对象，至少有一个操作是写

- 写-写冲突 ⟶ 脏写
- 写-读冲突 ⟶ 脏读
- 读-写冲突 ⟶ 不可重复读

② 冲突等价：如果两个调度涉及相同事务的相同操作，且每一对冲突的操作在两个调度中的顺序都相同，则这两个冲突等价调度。

③ 冲突可串行化调度：如果一个调度冲突等价于一个串行调度，则该调度是冲突可串行化调度。

└→ 判断是否是
- 定义
- 调顺序 [交换非冲突操作]看能不能转换成串行调度
- 优先图：无环 ⟺ 冲突可串行化

2. 并发控制协议 — 基于锁的

(1) 锁：控制并发访问权限的一种手段

① 事务只有获得了对象的锁才有读写权限，② 如果事务请求锁但没得到则等待直到得到

③事务完成对某个对象的操作后必须释放其锁

(2) 锁的类型

①共享锁/S锁——读权限

②互斥锁/X锁——读写权限

(3) 锁的相容性：S(↓X) S(↓S) X(↓S) X(↓X)，只有S(↓S)是允许的

读写冲突　写读冲突　写写冲突

例：使用共享锁和互斥锁的并发事务执行

T_1	T_2	调度器的动作
X-LOCK(A)		A上的互斥锁授予T_1
r(A)		
	X-LOCK(A)	T_2申请A上的互斥锁，不给
w(A)		
UNLOCK(A)		释放A上的互斥锁
	r(A)	A上的互斥锁授予T_2
	w(A)	
S-LOCK(A)		T_1申请A上的共享锁，不给
	UNLOCK(A)	释放A上的互斥锁
r(A)		A上的共享锁授予T_1
UNLOCK(A)		释放A上的共享锁

弊端：不是冲突可串行化调度，怎么办？

(4) 两阶段锁协议(2PL)

- 扩张阶段：事务向锁管理器请求所需的锁
- 收缩阶段：事务释放它获得的锁，但不能再请求加锁

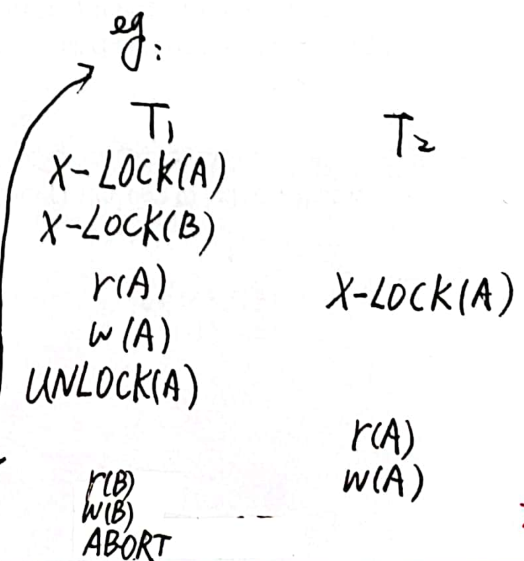
——核心：解锁之后不可以再加锁。

例：基于2PL的并发事务执行

T_1	T_2	调度器的动作
X-LOCK(A)		A上的锁(互斥)授予T_1
r(A)		
	X-LOCK(A)	T_2申请A上的互斥锁，不给
w(A)		
r(A)		
UNLOCK(A)		释放A上的锁
	r(A)	A上的互斥锁授予T_2
	w(A)	
	UNLOCK(A)	释放A上的互斥锁

2PL的优缺点：优点：能够保证冲突可串行化

缺点：
- 2PL面临级联中止的问题
- 2PL可能导致死锁


eg:

T_1	T_2
X-LOCK(A)	
X-LOCK(B)	
r(A)	X-LOCK(A)
w(A)	
UNLOCK(A)	
	r(A)
	w(A)
r(B)	
w(B)	
ABORT	

(5) 解决级联中止——强两阶段锁协议(SS2PL)

① 严格调度：一个调度中任意事务在结束前由它写入的DB对象的值没有被其他事务读过或修改过，则该调度称为严格调度。

└→ 不会产生级联中止

② SS2PL协议

- 增长阶段：与2PL增长阶段相同
- 萎缩阶段：当事务结束时，释放它获得的全部的锁。

③ SS2PL协议能保证生成严格的冲突可串行化调度

eg:

T_1	T_2
X-LOCK(A)	
X-LOCK(B)	
r(A)	X-LOCK(A)
w(A)	
r(B)	
w(B)	
UNLOCK(A)	
UNLOCK(B)	
ABORT	
	r(A)
	w(A)
	r(A)

(6) 死锁问题

① 如果一组事务中每个事务都在等待其他事务释放锁，则这组事务形成死锁

② 死锁的处理

- 死锁检测：检测是否发生，发生则解决
  - 超时检测（在给定时间内无事务执行，认为发生）
  - 等待图
- 死锁预防：预防死锁的发生

eg:

T_1	T_2
X-LOCK(A)	
r(A)	
	S-LOCK(B)
	r(B)
	S-LOCK(A)
w(A)	
X-LOCK(B)	

③ 等待图检测

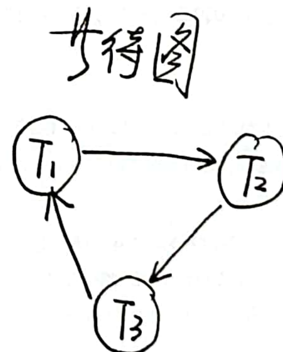
- 图中的顶点代表事务
- 如果事务T_i正在等待T_j释放锁，则存在从T_i到T_j的有向边。

事务产生死锁 ⟺ 图中有环

eg:

T_1	T_2	T_3
S-LOCK(A)		
r(A)		
	X-LOCK(B)	
	w(B)	
S-LOCK(B)（拒绝）		
		S-LOCK(C)
		r(C)
	X-LOCK(C)（拒绝）	
		X-LOCK(A)（拒绝）

等待图



对等待图中的死锁解除：从等待图中选择一个事务牺牲，中止该事务

- 事务的年龄/启动时间
- 事务的进度
- 事务持有锁的数量
- 需要级联中止的事务数量
- 事务被牺牲的"次数"，防止饿死

④ 死锁预防

<1> 事务启动时，DBMS为事务分配一个唯一且固定的优先级，开始越早，优先级越高

<2> Wait-Die规则——"尊老爱幼"

事务Ti请求事务Tj拥有的锁无法获得时，如果Ti比Tj优先级高，则Ti等待。

尊老　否则Ti中止（Ti重启后，其优先级保持不变）　爱幼

<3> Wound-Wait规则——"论资排辈"

事务Ti请求事务Tj拥有的锁而无法获得时，如果Ti比Tj优先级高，则Tj中止（Tj重启后，其优先级保持不变）否则Ti等待

第二部分：故障恢复

1、故障时的两类情况：

已提交的事务对DB的修改未全部持久化到磁盘——破坏事务的持久性.—Redo

已中止的事务对DB的修改已部分持久化到磁盘——破坏事务的原子性.—Undo

2、DBMS如何运用undo和redo取决于DBMS如何管理Buffer Pool（缓冲区策略）

- 只undo
- 只redo
- undo+redo
- 都不用

(1) 对未提交事务所做的修改是否允许写到磁盘 —— 允许：steal；不允许：no-steal

(2) 是否强制事务在提交前必须将其所做的修改写回磁盘 —— 强制：force；不强制：no-force

(3) 四种组合

	I/O效率低	I/O效率高
缓冲区效率高	Steal + Force　undo	Steal + No-Force　redo+undo
缓冲区效率低	No-Steal + Force	No-steal + No-Force　redo

No-Steal + Force：效率最差，但无需做故障恢复，实现简单，缺点是要求缓冲区容纳的下未提交事务的修改

3、预写式日志(WAL)

(1) DBMS维护一个日志文件，用于记录事务对数据库的修改。

(2) WAL协议：
事务启动时 <tid, BEGIN>（tid：事务ID）
事务提交时 <tid, COMMIT>
事务修改对象A时 <tid, oid, before, after>（oid：A的ID；before：旧值；after：新值）

(3) 事务的分类
- 已提交事务：既有<T.BEGIN>，又有<T.COMMIT>
- 不完整事务：只有<T.BEGIN>，而没有<T.COMMIT>
- 已中止事务：既有<T.BEGIN>，又有<T.ABORT>
  - → 在恢复的过程中既不需要Redo，也不需要Undo
  - → 如果恢复完一个事务T，但它已全部撤销，追加一个<T.ABORT>到日志

(4) 三种WAL协议

① Undo Logging：WAL+STEAL+FORCE

<1>顺序 <Ti.BEGIN>写入Disk → <Ti.A.before, after>写入磁盘 → A写入磁盘 → <Ti.COMMIT>写入磁盘（↑ 事务提交的真正标志）

<2>恢复方法：从后向前扫描日志

遇到<T.COMMIT>和<T.ABORT>都标记为已提交/中止事务，无需Undo

遇到<T.A.before, after>若T不是完整事务，则把Disk上A值恢复为before

遇到<T.Begin>时，若T不完整，则追加<T.ABORT>

② Redo Logging：WAL+NO-STEAL+NO-FORCE

<1>顺序 …… <Ti.COMMIT>写入磁盘 → A写入磁盘

<2>恢复方法：从前向后扫两遍日志

第一遍：<T.COMMIT>和<T.ABORT>标记为需要Redo和无需Redo

第二遍：<T.A.before, after>若T已提交，则重写A为after

不完整的事务追加<T.ABORT>

③ Redo+Undo Logging：WAL+STEAL+NO-FORCE（几乎所有DBMS都采用的策略）

<1>顺序：…… <Ti.COMMIT> —NO-FORCE→ A写入磁盘

或 A写入磁盘 —STEAL→ <Ti.COMMIT>

<2>恢复方法：Redo阶段和Undo阶段同上

4. 检查点

(1) WAL的问题：日志永远在变大，故障恢复时要扫描日志，恢复时间越来越长

(2) DBMS定期设置检查点

① 将日志刷写到磁盘

② 根据缓冲池策略，将脏页写到磁盘

③ 故障恢复时只需扫描到最新的检查点

(3) 模糊检查点

① 检查点开始：向日志中写入 <BEGIN CHECKPOINT(T_1,…,T_n)>

其中T_i是检查点开始时的活跃事务 → 尚未提交或中止的事务.

② 检查点中间：根据缓冲区策略，把脏页写到Disk.

- 若采用STEAL，则将全部脏页写到磁盘
- 否则，只将已提交事务所做的修改写到磁盘

③ 检查点结束：向日志中写入 <END CHECKPOINT>，并将日志刷写到磁盘

- 若NO-FORCE，则写完(脏页)就可结束检查点
- 否则须等T_i全部提交后才能结束.

(4) STEAL+NO-FORCE下的故障恢复

① Redo阶段

- 从前向后扫
- 从最近的完整检查点的<BEGIN CHECKPOINT(T_1,…T_n)>开始扫（需要Redo的一定∈这个集合）

② Undo阶段

- 从后向前扫
- 从日志中最近的完整检查点为 <BEGIN CHECKPOINT (T_1,T_2,…,T_n)>

  …

  <END CHECKPOINT>

  扫描到T_1…T_n中的最早的事务T_i的日志记录<T_i, Begin>为止（需要undo的一定∈这个集合）

后记

在复习和准备讲义的整个过程中，越深入去理解数据库课程的内容，就越发现它的博大精深和我知识的匮乏。作为我们数据科学与大数据技术的核心课程，它的重要性不言而喻，希望我的这份指南能够帮助大家更好地理解和掌握数据库的精髓与内涵。

课程的知识含量毋庸置疑非常大，但路虽难，行则将至；事虽难，做则必成。我相信通过努力复习，大家都能攻克这座难关。

最后，祝愿志在 90+的同学都能如愿以偿，祝愿 60 分万岁的同学都能顺利通过。

哈工大计算学部金牌讲师团 杨明达

2023 年 6 月 10 日

致谢

感谢张宇非同学和崔志阳同学指出讲义中的错误。

感谢所有 2103501 班同学对我的大力支持。