数据库系统复习整理

RandomStar

目录

第一部分：基本概念和关系代数

1.0 课程绪论

- Nothing important，主要是介绍课程的评分标准和瞎吹一通

1.1 关系型数据库 Relational Database

关系型数据的一些基本特点

- 关系型数据库是一系列**表的集合**
- 一张表是一个基本单位
- 表中的一行表示一条关系

1.2 基本概念和结构

- a relation r is a subset of $D_1 \times D_2 \times \times D_n$，一条 relation 就是其中的一个 n 元的**元组 (tuple)**
- attribute 属性，指表中的**列名**
  - attribution type 属性的类型
  - attribute value 属性值，某个属性在某条 relation 中的值
    - 关系型数据库中的属性值必须要是 atomic 的，即不可分割的
    - domain：属性值的值域，null 是所有属性的 domain 中都有的元素，但是 null 值会造成一些问题
- Relation Schema 关系模式
  - $R = (A_1, A_2, , A_n)$ 其中 A_i 是一系列属性，关系模式是对关系的一种**抽象**
  - r(R) 表示关系模式 R 中的一种关系，table 表示这个关系当前的值 (关系实例)
    - 每个关系 r 中的元素是 table 中的一行
    - 不过经常用相同的名字命名关系模式和关系
  - 关系是**无序**的，关系中行和列的顺序是 irrelevant 的

1.3 Keys 键

- super key 超键：能够**唯一标识**元组的属性集，即对于每一条关系而言超键的值是唯一的
  - 超键可以是多个属性的组合
  - 如果 A 是关系 R 的一个超键，那么 (A, B) 也是关系 R 的一个超键
  - 超键的 "唯一标识" 各个元组是可以有冗余信息的
- candidate key 候选键：**不含多余属性**的超键
  - 如果 K 是 R 的一个超键，而任何 K 的真子集不是 R 的一个超键，那么 K 就是 R 的一个候选键
- primary key 主键：
  - 数据库管理员**指定**的元组标识的一个候选键，不能是 null 值
- foreign key 外键：用来描述两个表之间的关系，可以有空值
  - 如果关系模式 R1 中的一个属性是另一个关系模式 R2 中的一个**主键**，那么这个属性就是 R1 的一个外键
    - a foreign key from r1 referencing r2

1.4 Relational algebra 关系代数

- Select 选择：$\sigma_p(r) = \{t|t \in r \cap p(t)\}$
  - 筛选出所有满足条件 p(t) 的元素 t
- Project 投影：$\prod_{A_1,A_2,\cdots,A_k}(r)$
  - 运算的结果是原来的关系 r 中各列只保留属性 $A_1, A_2, \ldots, A_k$ 后的关系
  - 会**自动去掉重复**的元素，因为可能投影的时候舍弃的属性是可以标识关系唯一性的属性
- Union 并操作：$r \cup s = \{t|t \in r \cup t \in s\}$
  - 两个关系的属性个数必须相同
  - 各属性的 domain 必须是可以比较大小的

- Set difference 差操作：$r - s = \{t | t \in r \cap t \notin s\}$
- Cartesian-Product 笛卡尔积：$r \times s = \{tq | t \in r \cap q \in s\}$
  - 两个关系必须是不相交的，如果相交则需要对结果中重复的属性名进行重命名
  - 笛卡儿积运算的结果关系中元组的个数应该是 rs 的个数之乘积
- Renaming 重命名：$\rho_X(E)$
  - 将 E 重命名为 x, 让一个关系拥有多个别名，同时 X 可以写为 $X(A_1, A_2, \ldots, A_n)$ 表示对属性也进行重命名
  - 类似于 C++ 中的引用
- 扩展运算: 可以用前面的六种基本运算得到
  - Intersection 交运算 $r \cap s = \{t | t \in r \cap t \in s\} = r - (r - s)$
  - Natual-Join 自然连接：$r \bowtie s$
    - 两个关系中同名属性在自然连接的时候当作**同一个属性**来处理
    - **Theta join** 满足某种条件的合并：$r \bowtie_\theta s = \sigma_\theta(r \bowtie s)$
  - Outer-Join 外部连接，分为左外连接，右外连接，全外连接
    - 用于应对一些**信息缺失**的情况 (有 null 值)
    - 左外连接 $\ltimes$
      - 左边的表取全部值按照关系和右边连接，右边不存在时为空值
    - 右外连接 $\rtimes$
      - 右边的表取全部值按照关系和右边连接，不存在为空值
    - Full join 左右全上，不存在对应的就写成空值
  - Division 除法：$r \div s = \{t | t \in \prod_{R-S}(r) \cap \forall u \in s(tu \in r)\}$
    - 如果 $R = (A_1, A_2, \ldots, A_m, B_1, \ldots, B_n) \cap S = (B_1, \ldots, B_n)$ 则有 $R - S = (A_1, A_2, \ldots, A_m)$

- Assignment 声明操作，类似于变量命名用 $\leftarrow$ 可以把一个关系代数操作进行命名

- Aggregation operations 聚合操作
  - 基本形式：$_{G_1,G_2,\ldots,G_n}\mathcal{G}_{F_1(A_1),\ldots,F_n(A_n)}(E)$
  - G 是聚合的标准，对于关系中所有 G 值相同的元素进行聚合，F( ) 是聚合的运算函数
  - 常见的有 SUM/MAX/MIN/AVG/COUNT

第二部分：SQL

2.1 SQL 基本概念

- **SQL：结构化查询语言**，分为 DDL,DML,DCL 几种类型，用的比较多的标准是 SQL-92
- **非过程式的语言**

2.2 SQL 创建，更新，删除

- SQL 支持的数据类型
  - char, varchar, int, numeric(p,d), null-value, date, time, timestamp
  - 所有的数据类型都支持 null 作为属性值，可以在定义的时候声明一个属性的值 not null
- 创建数据表 create table
  - 创建表的语法

```
create table table_name(
                variable_name1 type_name1,
                variable_name2 type_name2,
            (integrity-contraints)
                ……,)
```

  - `integrity-contraint` 完整性约束：可以**指定 primary key, foreign key references xxx, not null**
- 删除数据表 `drop table`

- 更新数据表的栏目 `alter table`
  - `alter table R add A D` 添加一条新属性
    - 其中 A 是属性名，D 是 A 的 domain
  - `alter table R drop A` 删除 A 属性

2.3 SQL 查询

- 事实上的最重要 SQL 语句，考试考的一般都是查询语句
- SQL 查询的基本形式：select 语句

```
select A1,A2,⋯,An
from r1,r2,⋯,rn
where P
```

  - 上述查询等价于 $\prod_{A_1,A_2,\cdots,A_k}(\sigma_p(r_1 \times r_2 \times \cdots \times r_m))$
  - SQL 查询的结果是一个关系
- select 子句的一些**细节**
  - `select * from xxx` 表示获取**所有属性**，事实上我怀疑 * 是正则表达式，表示可能有的所有内容，从后面的内容来看，select 语句确实是支持正则表达式
  - SQL 中的**保留字**对于**大小写不敏感**
  - 去除重复：`select distinct`，防止重复丢失的办法 `select all`
  - select 子句中的表达式支持基本的**四则运算** (加减乘除)，比如

```
select ID, name, salary/2
from instructor;
```

  - where 子句中：
    - 支持 `and or not` 等逻辑运算
    - 支持 `between and` 来查询范围
  - from 子句：

    * 重命名操作，可以通过 `old_name as new_name` 进行重命名
    * from 可以选择多个表，此时会先将这些表进行笛卡儿积的运算，再进行 select
    * 元组变量：可以从多个表中 select 满足一定条件的几个不同属性值的元组

```
select instructor.name as teacher-name, course.title as course-title
from instructor, teaches, course
where instructor.ID = teaches.ID and
        teaches.course_id = course.course_id and
        instructor.dept_name = 'Art';
```

- 字符串支持**正则表达式**匹配，用 `like regrex` 的方式可以进行属性值的正则表达式匹配
    * 正则表达式的用法没怎么讲

```
select name
from teacher
where name like '%hihci%';
```

- 将输出的结果排序
    * `order by 属性名 asc/desc`
- 集合操作：
    * 可以用 `union/intersect/except` 等集合运算来**连接两条不同的查询**
- 聚合操作：
    * 支持的操作有 avg/min/max/sum/count，获取的是表中的统计量

```
select dept_name,avg(salary) as avg_salary
from instructor
group by dept_name;
```

    * 事实上 SQL 语句的聚合操作和关系代数中的聚合运算是完全对应的，关系代数中的聚合运算表达式 $_{G_1,G_2,\ldots,G_n}\mathcal{G}_{F_1(A_1),\ldots,F_n(A_n)}(E)$ 对应的 SQL 语句是

      ```
      select G1,G2,⋯,Gn,F1(A1),⋯,Fn(An)
      ```

```
from E
group by G1,G2,⋯,Gn;
```

* 聚合操作的 SQL 语句书写可以在末尾用 `having xxx` 来表示一些需要聚合操作来获得的条件，比如

```
select cno
from pos natural join detail
where year(detail.cdate) = 2018
having count(distinct campus) = 1;
```

- Null values 空值
  * 属性值可以为 null，当然也可以在定义数据表的时候规定哪些元素不能为空

- Nested Subquery 嵌套查询
  - 对于查询

```
select A1,A2,⋯,An
from r1,r2,⋯,rn
where P
```

其中的 A，r，P 都可以被替换为一个**子查询**

  - 集合关系：用 `in/not in +` 子查询来判断否些属性是否属于特定的集合中
    * `some+` 子查询用于判断集合中是否存在满足条件的元组，用来判断存在性
    * `all+` 子查询可以用来筛选最值
    * `exists+` 子查询判断子查询的结果是否不为空
    * `not exists+` 子查询判断是否为空集
  - with 子句：对子查询定义一个变量名，可以在之后调用
  - scalar 子查询：用于需要一个值作为查询结果的时候
  - join 子句：可以对若干张表进行各种 join 之后再查询
    * `natural join` 自然连接

    * A join B on(xxx)

2.4 SQL 插入，删除，更新

- 插入: `insert into table_name values( );`
  - 可以用 select 查询子句得到的结果作为 values，此时可以同时插入多条结果
- 删除：`delete from table_name where xxxxxx`
- 更新：`update table_name set xxx where xxxxx`
  - case 子句：用于分类讨论

```
update instructor
        set salary = case
                                        when salary <= 100000 then salary*1.05
                                        else salary *1.03
                                        end
```

2.5 SQL view 视图, index 索引

- 视图：一种**只显示数据表中部分属性值**的机制
  - 不会在数据库中重新定义一张新的表，而是隐藏了一些数据
  - 创建视图的定义语句：
    * xxx 是视图的名称，内容是从某个 table 中 select 出的

```
create view xxx as (a subquery)
```

- 视图的更新
  - 也需要使用 insert 语句更新视图
  - 可以更新的条件
    * 创建时只使用了一张表的数据
    * 创建时没有进行 distinct 和聚合操作
    * 没有出现空值和 default

- Domain 创建新类型
  - `create domain new_name + data type`(比如 char(20))
  - domain 可以设置约束条件，比如下面这一段 domain 定义表示 degree_level 只能在这三个中进行选择

```
create domain degree_level varvhar(10)
constraint degree_level_test
check(value in ('Bachelors', 'Masters', 'Doctorate'));
```

  - Large-Object Types 大对象类型，分为 blob(二进制大对象) 和 clob(文本大对象) 两种，当查询需要返回大对象类型的时候，取而代之的是一个代表大对象的指针
- Index 索引
  - 语法 `create index index_name on table_name(attribute)`
  - 在对应的表和属性中建立索引，加快查询的速度
- Transactions 事务
  - 一系列查询等操作的集合
  - Atomic transaction 原子事务：只能被完全执行或者回滚 (roll back)

2.6 Integrity 完整性控制

- 单个关系上的约束
  - 主键 primary key，**unique**，not null
  - check 子句：写在数据表的定义中
    - check(P) 检查某个属性是否为特定的一些值
  - Domain constraints 值域的约束
    - 在 domain 的定义中加入 check
    - 语法 `create domain domain_name constraints check_name check(P)`
  - **Referential Integrity** 引用完整性

    - 被引用表中主键和外键的关系
    - 其实 PPT 里这一段讲了半天就是在说要在定义表的时候定义主键和外键进行约束
  - Cascading action(不知道这一段在讲什么狗屁)
    - on update
    - on delete
- 对于整个数据库的约束
  - Assertions(Mysql 不支持)
    - 对于数据库中需要满足的关系的一种**预先判断**
    - `create assertion <assertion-name> check <predicate>` 下面是一段例子

```
 create assertion credits_constaint check
( not exists(
 select *
    from student S
 where total_cred <>(
            select sum(credits)
            from takes nature join course
            where takes.ID = S.ID and grade is not null and grade <> 'F'
         )
        )
)
```

    - Trigger 触发器
      - 在修改了数据库时会自动执行的一些语句
      - 时间节点的选择
      - **referencing old row as** 对旧的行进行操作，用于删除和更新
      - **referencing new row as** 对新的行进行操作，用于插入和更新

* trigger event 触发事件
    - insert/delete/update 等操作都可以触发设置好的 trigger
    - 触发的时间点可以是 before 和 after，触发器的语法如下

```
create trigger trigger_name before/after trigger_event of table_name on attribute
referencing xxx
for each row
when xxxx
begin
xxxx(SQL operation)
end
```

2.7 Authorization

- 数据库中的四种权限 read,insert,update,delete
- Security specification in SQL 安全规范
    - grant 语句可以赋予用户权限 `grant <privilege list> on <relation name or view name> to <user list>`
    - `<user list>` 可以是用户名，也可以是 public(允许所有有效用户拥有这项权限)
    - grant 语句后面可以加 `with grant option`，表示该用户拥有赋予其他用户这项权限的权力
    - revoke 权力回收
        * `revoke <privilege list> on <relation/view name> from <user list> [restrict|cascade]` 从用户中回收权力
    - role 语句
        * `create role role_name`
        * 允许一类用户持有相同的权限

第三部分：ER 模型和 Normal Form(范式)

3.1 E-R 模型

- E-R 模型由 enitites(实体) 和 relation(关系) 组成
- Entity set 实体集
  - 实体是一系列独特的对象，用一系列属性来表示
  - 同一类实体共享相同的 Properties，实体集就是由同类型的实体组成的集合
  - 表示方法
    - 长方形代表实体集合
    - 属性写在长方形中，**primary key 用下划线**标注
  - 实体集中对于属性的定义和之前的几乎一样
    - 实体集中属性定义可以存在组合与继承的关系，下面是一个样例

      Instructor

      ID

      name

      first_name

      last_name

      address

      street

      street_number

      street_name

      apt_number

      city

      state

- Relationship set 关系集
  - 一个 relationship 是几个实体之间的联系，关系集就是同类关系之间构成的集合
  - 一个 relationship 至少需要两个及以上的实体，一个关系集至少和两个实体集有关联
    - 一个关系集所关联的实体集的个数称为 degree，其中以二元关系集为主
- E-R model constraints 约束
  - mapping cardinalities **映射基数**
    - 二元关系中映射基数只有一对一，一对多，多对一，多对多
    - E-R 模型中表示映射关系：箭头表示一，直线表示多
    - 三元关系中：**箭头只能出现一次**，否则会出现**二义性**
  - 参与度约束
    - total participation：若一个实体集全部参与到关系中，要用两条线
    - partical participation 部分参与
  - key 约束：和前面的基本一样
  - 弱实体集 weak entity set：一些实体集的属性不足以形成主键，就是弱实体集，与之相对的是强实体集
    - 用于表示一些关系中的依赖性，弱实体集需要和强实体集关联才有意义
    - 经常出现在一对多的关系中，在 ER 图中需要用**双线方框**表示，比如职工和职工家属，职工家属不能脱离于职工存在，所以职工家属就是一个弱实体集
- Aggregation 聚合
  - 可以把一部分 E-R 关系聚合成一个 Entity 进行操作
  - 在 ER 图中用方框将一些关系集和实体集括起来表示一个聚合后的实体集
- Specialization 特殊化
  - 自顶向下的设计过程

  - Attribute inheritance：overlapping，disjoint
  - 画图的方式就是从上往下画，Entity 的内容逐渐细分，但是都继承了上一阶的所有 attribute
- Generalization 泛化
  - 自底向上的设计过程
  - 从下往上，下层的内容合成上层的内容

3.2 E-R Diagram

- E-R 图中的各种表述

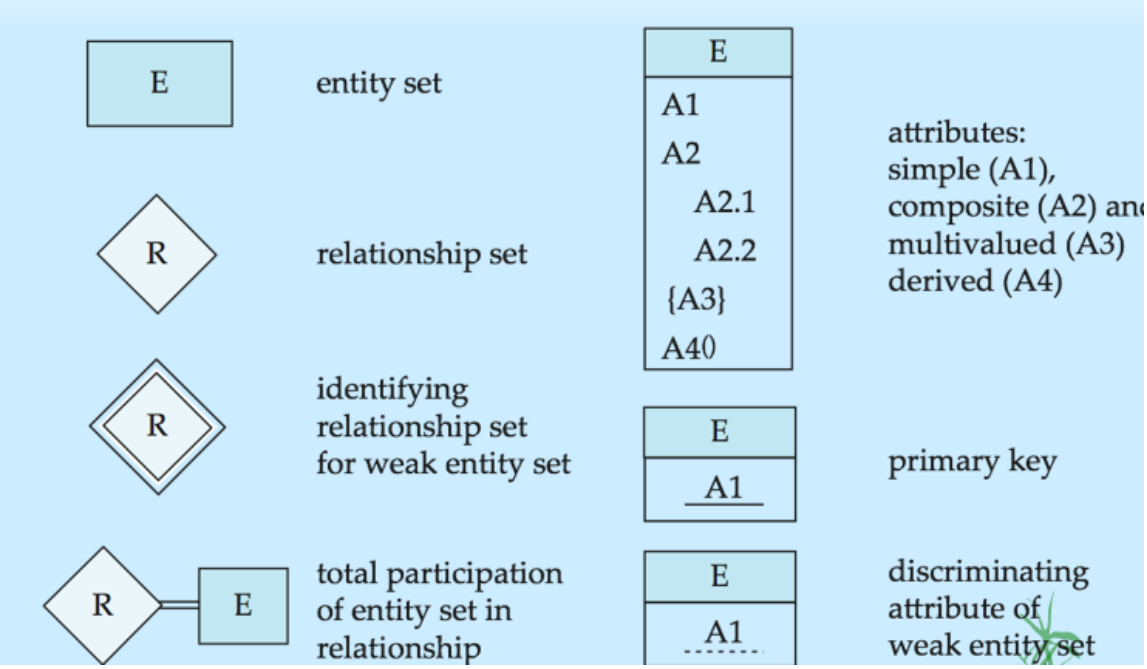
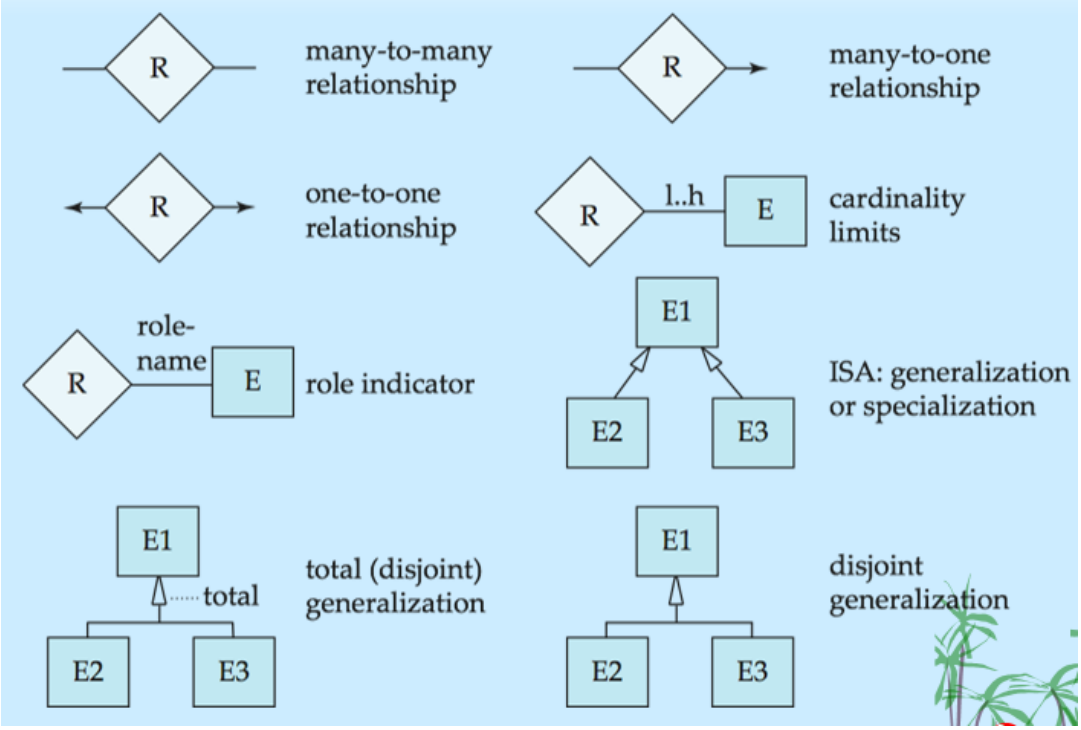


图 1:



3.3 Normal Form 范式

3.3.1 数据库设计的目标

- 存储信息时没有不必要的冗余，检索信息的效率高
- 这些设计方式通过各种范式 (normal form) 来实现

3.3.2 First Normal Form 第一范式

- 原子性 atomic ：不能再继续拆分，属性不能再向下拆分
- 第一范式的定义：一个关系模式 R 的**所有属性都是 atomic 的**，这个关系模式 R 就是**第一范式**
- 存在的问题
  - redundancy 冗余
  - complicates updates 更新数据很复杂

- null-values---difficult to insert/remove

- Decomposition 分解
  - Lossy Decomposition 有损的分解：不能用分解后的几个关系重建原本的关系
  - Lossless join 无损分解的定义：
    - R 被分解为 (R1, R2) 并且 $R = R_1 \cup R_2$
    - 对于任何关系模式 R 上的关系 r 有 $r = \prod_{R_1}(r) \bowtie \prod_{R_2}(r)$

3.3.3 Functional dependency 函数依赖

- **函数依赖**的定义
  - 对于一个关系模式 R，如果 $\alpha \subset R$ 并且 $\beta \subset R$ 则函数依赖 $\alpha \to \beta$ 定义在 R 上，当且仅当
    - 如果对于 R 的任意关系 r(R) 当其中的任意两个元组 t1 和 t2，如果他们的 α 属性值相同可以推出他们的 β 属性值也相同
  - 如果某个属性集 A 可以决定另一个属性集 B 的值，就称 $A \to B$ 是一个函数依赖
  - 函数依赖和键的关系：函数依赖实际上是键的概念的一种泛化
    - K 是关系模式 R 的**超键**当且仅当 $K \to R$
    - K 是 R 上的**候选主键**当且仅当 $K \to R$ 并且不存在 $\alpha \subset K, \alpha \to R$
  - 一个平凡的的结论：子集一定对自己函数依赖

3.3.4 闭包

- Closure 闭包
  - 闭包, 对于原始的函数依赖集合 F 可以推出的所有函数依赖关系产生的集合就是 **F 的闭包**
  - 符号用 F^+ 表示
  - 函数依赖的性质

    * reflexity：α 的子集一定关于 α 函数依赖
    * augmentation：如果 $\alpha \to \beta$ 则有 $\lambda\alpha \to \lambda\beta$
    * transitivity：如果 $a \to \beta \cap \beta \to \gamma$ 则有 $a \to \gamma$
    * union：如果 $\alpha \to \beta \cap \alpha \to \gamma$ 则有 $\alpha \to \beta\gamma$
    * decomposition：如果 $\alpha \to \beta\gamma$ 则有 $\alpha \to \beta \cap \alpha \to \gamma$
    * pseudotransitivity：如果 $\alpha \to \beta \cap \beta\gamma \to \delta$ 则有 $\gamma\alpha \to \delta$
- 计算闭包的方法
    * 根据初始的函数依赖关系集合 F 和函数依赖的性质，计算出所有的函数依赖构成闭包
    * 可以用有向图表示属性之间的关系，通过图来写出所有的函数依赖
- 属性集的闭包
    * 闭包中所有关于 α 函数依赖的属性集构成的集合
        · 即如果 $(\alpha \to \beta) \in F^+$ 则有 $\beta \in \alpha^+$
    * 计算**属性集闭包**的算法

```
result={a}
while result is changed do
for each b->c in F do
begin
if b is in result then push c into result
end
```

    * 属性集闭包的作用
        · 测试是否为主键：如果 α 的闭包包含了所有属性，则 α 就是主键
        · 测试函数独立：为了验证 $\alpha \to \beta$ 是否存在只需要验证 β 是否在 α 的闭包中
        · 计算 F^+：通过每个属性的闭包可以得到整个关系模式的闭包
- 判定是否为 Lossless Join 的办法

* 当且仅当 $R_1 \cap R_2 \to R_1$ 或者 $R_1 \cap R_2 \to R_2$ 至少有一个 F^+ 中

3.3.5 BCNF/3NF

- BC 范式 (Boyee-Codd Normal Form)
  - BC 范式的条件是：闭包 F^+ 中的所有函数依赖 $\alpha \to \beta$ 至少满足下面的一条
    * $\alpha \to \beta$ 是平凡的 (也就是 β 是 α 的子集)
    * α 是关系模式 R 的一个**超键**，即 $\alpha \to R$
  - 如何验证 BCNF：
    * 检测一个非平凡的函数依赖 $\alpha \to \beta$ 是否违背了 BCNF 的原则
      · 计算 α 的属性闭包
      · 如果这个属性闭包包含了所有的元素，那么 α 就是一个**超键**
      · 如果 α 不是超键而这个函数依赖又不平凡，就打破了 BCNF 的原则
    * 简化的检测方法：
      · 只需要看关系模式 R 和已经给定的函数依赖集合 **F 中的各个函数依赖**是否满足 BCNF 的原则, 不需要检查 F 闭包中所有的函数独立
      · 可以证明如果 F 中没有违背 BCNF 原则的函数依赖，那么 F 的闭包中也没有
      · 这个方法不能用于检测 R 的分解
  - BC 范式的分解算法伪代码

```
result={R}
done=false
compute F+ by F
while (!done) do
if exist Ri in result that is not a BCNF
then begin
let a->b be a non-trivial function dependency that holds on Ri such that a->Ri is not in F
result=(result-Ri)or(Ri-b)or(a,b);
```

```
end
else
done=true
```

- 当我们对关系模式 R 进行分解的时候，我们的目标是
  - 没有冗余，每个关系都是一个 good form
  - 无损分解
  - Denpendency preservation 独立性保护, 把 R 和 F 的闭包按照关系的对应进行划分
    - 用 F_i 表示只包含在 R_i 中出现的元素的函数依赖构成的集合
    - 我们希望的结果是 $(F_1 \cup F_2 \cup \cdots \cup F_n)^+ = F^+$
      - BCNF 的分解一定是有独立性保护的
    - 独立性保护的验证算法
    - 如果最终的结果 result 包含了所有属性，那么函数依赖 $\alpha \to \beta$ 就是被保护的

```
result =
while result changed do
for each Ri in the composition
t = (result and Ri)+ and Ri
result = result or t
```

- Third normal form 第三范式
  - 第三范式的定义：对于函数依赖的闭包 F^+ 中的所有函数依赖 $\alpha \to \beta$ 下面三条至少满足一条
    - $\alpha \to \beta$ 是平凡的
    - α 是关系模式 R 的超键
    - 每一个 $\beta - \alpha$ 中的属性 A 都包含在一个 R 的候选主键中
  - BCNF 一定是 3NF，实际上 3NF 是为了保证独立性保护的 BCNF
  - 3NF 有冗余，某些情况需要设置一些空值

- 3NF 的判定
  - 不需要判断闭包中的所有函数依赖，只需要对已有的 F 中的所有函数依赖进行判断
  - 用闭包可以检查 $\alpha \to \beta$ 中的 α 是不是超键
  - 如果不是，就需要检查 β 中的每一个属性包含在 R 的候选键中

3.3.6 最小覆盖

- Canonical conver 最小覆盖问题
  - 函数依赖关系的最小集合 (也就是没有冗余，和 F 等价可以推导出 F+ 的关系集合)
  - 无关属性 Extraneous Attributes：
    - 定义：对于函数依赖集合 F 中的一个函数依赖 $\alpha \to \beta$
      - α 中的属性 A 是多余的，如果 F 逻辑上可以推出 $(F - \{\alpha \to \beta\}) \cup \{(\alpha - A) \to \beta\}$
      - β 中的属性 A 是多余的，如果 $(F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - A)\}$ 逻辑上可以推出 F
    - 判断 $\alpha \to \beta$ 中的一个属性是不是多余的
      - 测试 α 中的属性 A 是否为多余的, 计算 $(\alpha - A)^+$, 检查结果中是否包含 β，如果有就说明 A 是多余的
      - 测试 β 中的属性 A 是否为多余的, 只用 $(F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - A)\}$ 中优的依赖关系计算 α^+, 如果结果包含 A，就说明 A 是多余的
  - 最小覆盖 F_c 的定义
    - 和 F 可以互相从逻辑上推导出，并且最小覆盖中没有多余的信息
    - 最小覆盖中的每个函数依赖中左边的内容都是 unique 的
    - 如何计算最小覆盖：PPT-8 的 53 页有一个例子
      - 先令 Fc=F
      - 用 Union rule 将 Fc 中所有满足 $\alpha \to \beta_1 \cap \alpha \to \beta_2$ 的函数依赖替换为 $\alpha \to \beta_1\beta_2$

- 找到 Fc 中的一个函数依赖去掉里面重复的属性
- 重复 2，3 两个步骤直到 Fc 不再变化

下面这些部分看起来不太会考，先不管了，有空再看

- 3NF decomposition algorithm
- Multivalued denpendency
  - 多值依赖 a->->b，记作 D，闭包记为 D+
- Fourth Normal Form
  - 对于 D+ 中的所有 a->->b 有 □ 是平凡的或者 □a 是一个超键
  - 4NF 一定是 BCNF

第四部分: 数据库设计理论

4.1 存储和文件结构

这一部分好像不怎么会考试，理论性的东西比较多，跟计原的存储器部分好像还有一部分交集

- 存储的结构 storage hierarchy
  - primary 主存储器
    - 快而易失，常见的有主存和 cache
    - cache 的存取效率最高，但是 costly，主存访问快但是对于数据库而言空间太小
  - secondary 二级存储器
    - 不容易丢失，访问较快，又叫在线存储
    - 常见的是闪存和磁盘
  - tertiary 三级存储器
    - 不容易丢失，访问慢，但是容量大而 cheap，离线存储
    - 磁带，光存储器

- 总体的存储架构：cache--主存--闪存--磁盘--光盘--磁带

4.1.1 磁盘 Magnetic Disks

- 组成结构
  - read-write head 读写头
    - 和磁盘表面靠得很近
    - 用于读写磁盘中的文件信息
  - tracks 磁道, 由磁盘表面划分，每个硬盘大概有 50k 到 100k 个磁道
    - sectors 扇区，由磁道划分而成
      - 扇区是数据读写的最小单位
      - 每个扇区的大小是 512 字节，每个磁道有 500-1000 个扇区
  - 磁盘控制器：计算机系统和磁盘之间的接口
  - Disk subsystem 磁盘子系统：由 disk controller 操纵若干个磁盘组成的字系统
- 磁盘的性能评价标准
  - access time：访问时间，包括
    - seek time：读写头的 arm 正确找到 track 的时间，平均的 seek time 是最坏情况的一半
    - rotational latency：旋转造成的延迟，平均时间是最坏的一半
  - data-transfer rate 数据从磁盘读写的速度
  - MTTF：出现 failure 之前的平均运行时间
- 磁盘访问的优化
  - block ：一个磁道中的若干连续扇区组成的序列
  - buffering
  - read-ahead

- disk-arm-scheduling
- file organization: 通过按照访问数据的方式来组织 block 优化访问时间

4.1.2 File organization 文件组织

- 数据库存储在一系列的文件中，每个文件是一系列的记录，每条记录包含一系列的 fields
  - 每个文件被划分为固定长度的 block，block 是数据存取/存储空间分配的基本单位
  - 一个 block 有多条记录，在传统的数据库中
    - 记录的长度不能超过 block
    - 每条记录一定都是完整的
  - Free List 用链表的形式来存储 records
  - Variable-length records 变长记录
    - 典型的变长记录
      - 属性按照顺序存储
      - 变长的变量用 offset+data 的形式存储，空值用 null-value bitmap 存储
    - slotted page 结构，它的 header 包含
      - 记录的总数
      - block 中的空闲区域的 end
      - 每条记录所在的位置和大小
  - 文件中记录的组织方式
    - heap
    - sequential
    - hashing
    - nulti-table clustering file organization

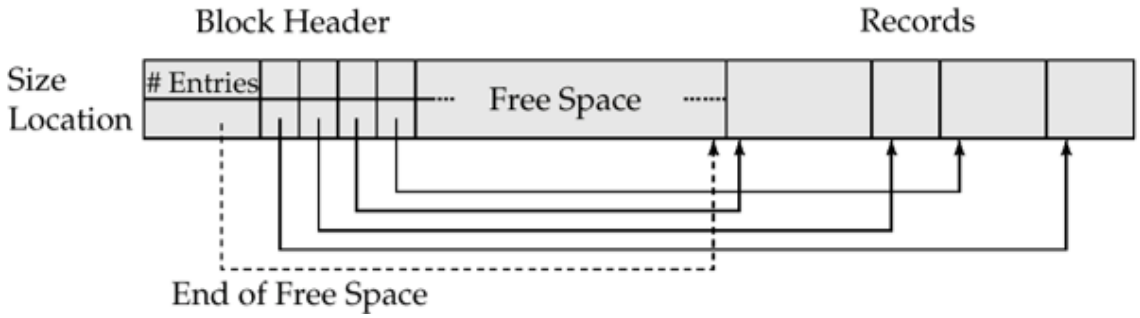


图 2:

- 存储缓冲区的管理
  - 通过将数据放到主存中来提高访问效率
    - buffer manager：用于管理缓冲区中的内存分配
      - 当需要从磁盘读取 block 的时候，数据库会调用 buffer mananger 的功能
      - 如果 block 已经在 buffer 中了，就直接返回这个 block 的地址
      - 如果不在，则 buffer manager 会动态分配 buffer 中的内存给 block，并且可能会覆盖别的 block，然后将磁盘中 block 中的内容写入 buffer 中, 涉及到 buffer 的替换算法 LRU strategy 即替换掉最近使用频率最低的 block
    - pinned block 内存中的不允许写回磁盘的 block，表示正在处理事务或者处于恢复接断

4.2 B+ 树索引

4.2.1 索引

- 数据库系统中引入索引机制，用于加快查询和访问需要的数据
  - search key 通过一个属性值查找一系列属性值，用于文件中查询
  - Index file 索引文件包含一系列的 search key 和 pointer(两者的组合被称为 index entry)，查询方式是通过 search key 在 index file 中查询 data 的地址 (pointer)，然后再从 data file 中查询数据

    * 两种 search key 的排序方式：ordered index，hash index
    * ordered index 顺序索引
        · index entry 按照 search key 的值来进行排列
        · primary key 指定文件顺序的索引 secondary key 次关键字
    * 索引的不同方式
        · Dense index 密集的索引：每一条记录都有对应的索引
        · Sperse index 稀疏的索引: 需要的空间和插入删除新索引的开销较小，但是比密集的索引要慢
        · Secondary indice 索引通过一个大的 bucket 来寻找所指向的地方
        · Multilevel index 多级索引，分为 outer index 和 inner index

4.2.2 B+ 树索引

- B+ 树文件索引
  - 通过 B+ 树的索引方式来寻找文件中数据的地址，B+ 树的定义和 ads 中的 B+ 树基本相同，
    * 树的非叶节点由指向儿子的指针和 search-key 相间组合而成
    * 两个 search-key 之间的指针指向的数据的值在这两个 search-key 之间
  - B+ 树上的查询的时间复杂度是 logN 级别，N 是 search key 的总个数
    * 查询的路径长度：不会超过 $\log_{n/2}(K)+1$ 其中 K 是 B+ 树中的索引的个数 (即规模 N)
    * B+ 树的一个节点的大小和一个磁盘区块一样大 (往往是 4KB) 而在 n 的规模一般在 100 左右
- B+ 树的更新：插入和删除
  - 插入的算法：先找到该插入的位置直接插入，如果当前的节点数量超过了阶数 M 则拆成两个部分，并向上更新索引

- 删除的算法: 直接把要删除的节点删除，然后把没有索引 key 了的非叶节点删除，从旁边找一个叶节点来合并出新的非叶节点

- B+ 树的相关计算
  - 高度的估计：
    - B+ 树高度最小的情况：所有的叶节点都满，此时的 $h = \lceil \log_N(M) \rceil$
    - 最大的情况，所有的叶节点都半满，此时的 $h = \lfloor \log_{\lceil N/2 \rceil}(\frac{M}{2}) \rfloor + 1$
  - size 大小的估计：也是两种极端情况

$$\left\lceil \frac{K}{n-1} \right\rceil + \left\lceil \left\lceil \frac{K}{n-1} \right\rceil * \left\lceil \frac{1}{n} \right\rceil \right\rceil + \left\lceil \left\lceil \frac{K}{n-1} \right\rceil * \left\lceil \frac{1}{n^2} \right\rceil \right\rceil + ... + 1 \leq Size$$

$$Size \leq \left\lceil \frac{K}{\lceil \frac{n-1}{2} \rceil} \right\rceil + \left\lceil \left\lceil \frac{K}{\lceil \frac{n-1}{2} \rceil} \right\rceil * \left\lceil \frac{1}{\lceil \frac{n}{2} \rceil} \right\rceil \right\rceil + \left\lceil \left\lceil \frac{K}{\lceil \frac{n-1}{2} \rceil} \right\rceil * \left\lceil \frac{1}{\lceil \frac{n}{2} \rceil^2} \right\rceil \right\rceil + ... + 1$$

图 3:

4.2.3 文件索引

- Hash 文件索引
  - 静态哈希
    - 使用一系列 buckets 来存储一系列的 records，通过 hash 函数和 search-key 的运算来查找文件
    - hash 函数：将不同的 search key 映射到不同的 bucket 里面去
  - Hash indices 将 hash 用于索引结构中
    - A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
    - hash indices are always secondary indices
  - 动态哈希

    * 哈希函数会被动态地修改
    * 可扩展的哈希

4.2.4 总结：存储结构和 B+ 树的计算

- 记录的存储：
  - 数据库的记录在 block 中存储，一个 block 中有大量的记录存储，有线性存储的，也有使用 B+ 树索引的
  - 线性存储的记录：
    * 假设一条记录的长度位 L，block 的大小为 B，那么一条记录中最多有 $\lfloor \frac{B}{L} \rfloor$ 条记录
    * 如果一共有 N 条记录，一个 block 中有 M 条记录，那么一共需要 $\lceil \frac{N}{M} \rceil$ 个 block，而 $M = \lfloor \frac{B}{L} \rfloor$
  - B+ 树索引 block 的计算，假设 block 的大小为 B，指针的大小是 a，被索引的属性值大小是 b
    * 要注意指针节点比属性值多一个，所以一个块上的扇出率 n(fan-out rate) 是 $\lfloor \frac{B-a}{a+b} \rfloor + 1$
    * n 也就是这个 B+ 树的阶数，然后根据公式来估算 B+ 树的高度，其中 M 应该是作为索引的值可以取到的个数

4.3 查询处理 QueryProcess

印象中这一部分的作业题以套公式算为主

- 查询处理的基本步骤
  - Parsing and translation 解析和翻译
  - Optimization 优化
    * 一种 SQL 查询可能对应了多种等价的关系代数表达式
    * 可以通过估计每种方式的 cost 来评判方法的好坏

    - 查询优化会选择最节约的方式进行查询
  - Evaluation
- Query cost 的计算
  - 主要的 cost 来源：disk access
    - seeks
    - block read
    - block written
  - cost 计算的方式：在 B 个 blocks 中查询 S 次所消耗的时间 =B 转移到一个 *block* 的时间 +S\一次查询的时间：$B \times t_T + S \times t_S$ 其中 t_T 表示一次 block transfer 的时间
    - cost 依赖于主存中**缓冲区的大小**：更多的内存可以减少 disk access
    - 通常考虑最坏的情况：只提供最少的内存来完成查询工作

4.3.1 select 的 cost 估计

- Select 操作的 cost 计算
  - Algorithm1: 线性搜索，查询每个 block 判断是否满足查询条件
    - Cost = br block transfers + 1 seek，其中 b_r 是关系 r 中**存储了记录的 block 的数量**
    - 如果通过键来搜索，在找到的时候就停止，则 cost = (br /2) block transfers + 1 seek
    - 二分搜索此时不起作用，因为数据不是连续存储的
  - Index scan--使用索引进行搜索
  - Algorithm2: primary index，equality on key，搜索一条记录
    - $cost = (h_i + 1) \times (t_T + t_S)$ --- hi 是索引的高度
  - Algorithm3: primary index, equality on non-key 需要搜索多条记录
    - 想要的结果会存储在连续的 block 中 (因为有主索引)
  - $cost = h_i(t_T + t_S) + t_S + t_T * b$ 其中 b 表示包含匹配记录的 block 总数

    - 当使用 B+ 树作为索引时可以节约一次 seek 的时间，$cost = h_i(t_T + t_S) + t_T * b$

- 算法 4：Secondary index
  - 用候选主键作为索引检索单条记录 $cost = (h_i + 1) \times (t_T + t_S)$
  - 用候选主键检索了 n 条记录 (不一定在同一个 block 上面)

    $Cost = (h_i + n) \times (t_T + t_S)$ 有时候会非常耗时

4.3.2 sort 和 join 的 cost 估计

- Sort：**external sort-merge** 其实类似于 ads 里面的外部归并排序
  - M 表示内存的大小，b_r 表示 block 的数量
    - 基本步骤如下
      - create sorted runs
      - merge the runs
    - 需要的 merge pass 总数 $[\log_{M-1}(b_r/M)]$
    - 创建和每次 run 过程中的 disk access 数量 2*br*
    - 外部排序中总的 disk access 次数 $(2[\log_{M-1}(b_r/M)] + 1)b_r$

- Join 操作的 cost 估计
  - nested-loop join
    - 计算 theta-join 表达式：$r \bowtie_\theta s$ 算法的伪代码如下

```
for each tuple tr in r do begin
    for each tuple ts in s do begin
        test pair (tr,ts) to see if they satisfy the join condition
        if they do, add tr • ts to the result
    end
end
```

    - block transfer 次数: $n_r \times b_s + b_r$

    - seeks 的次数 $n_r + b_r$

- block nested-loop join $r \bowtie_\theta s$

```
for each block Br of r do begin
    for each block Bs of s do begin
        for each tuple tr in Br  do begin
            for each tuple ts in Bs do begin
```

    - 最坏情况的 cost
        - block transfer $b_r \times b_s + b_r$
        - seeks $2b_r$
    - 最好情况的 cost
        - block transfers $b_r + b_s$ with 2 seeks
    - 优化：使用 M-2 个 block 作为 blocking unit(M 是内存可以容纳的 block 数量)，此时的
        - block transfer 次数 $=\frac{b_r}{M-2} \times b_s + b_r$
        - seek 次数 $= \frac{2b_r}{M-2}$

- Index nested-loop join
    - 索引一定程度上可以代替 file scan
    - $cost = b_r(t_T + t_s) + c \times n_r$ 其中 c 表示遍历索引和找到所有匹配的 s 中的 tuple 所消耗的时间，可以用**一次 s 上的单个 selection 来估计 s 的值**

- Merge-Join
    - 只能在 natural-join 和 equal-join 中使用
    - block transfer 的次数 $=b_r + b_s$，seek 的次数 $=[b_r/b_b] + [b_s/b_b]$

- Hash join：使用 hash 函数进行 join
    - h maps JoinAttrs values to {0, 1, ..., n_h}, 将两个关系进行比较和同类型的匹配

        * cost of hash-join
            · block transfer: $3(b_r + b_s) + 4n_h$,partition: 读 $b_r + b_s$ blocks 写 $(b_r + b_s) + 2n_h$ blocks, join: 读 $(b_r + b_s) + 2n_h$
            · seeks: $2([b_r/b_b] + [b_s/b_b])$
            · 如果所有东西都能放进主存里，则 $n_h = 0$ 并且不需要 partition
            · 需要 partition 的时候 $cost = 2(b_r + b_s[\log_{M-1}(b_s) - 1]) + b_r + b_s$

- Evaluation of Expression 表达式求值
    - **Materialization** 实体化
        * 依次进行表达式的计算，构建前缀树递归进行
    - Pipelining 流水线，同时评估多个操作
        * evaluate several operations simultaneously , passing the results of one operation on to the next.

4.4 查询优化 Query Optimization

- 两种查询优化的办法
    - 找到等价的查询效率最高的关系代数表达式
    - 指定详细的策略来处理查询

4.4.1 等价关系代数表达式

- Equivalent Expressions 等价的关系代数表达式
    - **evaluation plan**：类似于算术表达式的前缀树，表示了每部操作进行的过程
    - Cost-based optimization 基于 cost 的优化
        * 基本步骤
            · 用运算法则找到逻辑上等价的表达式
            · 注释结果表达式来获得查询计划

        - 选择 cost 最低的表达式
    * cost 的估算
        - 统计信息量的大小，比如 tuples 的数量，一个属性不同取值的个数
        - 中间结果的数量，用于复杂表达式的优化
        - 算法的消耗
    - **等价表达式的规则**
- 合取选择和选两次等价：$\sigma_{\theta_1 \cap \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
    - 选择两次的顺序可以交换：$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
    - 嵌套的投影只需要看最外层的：$\prod_{L_1}(\prod_{L_2}(\ldots(E))) = \prod_{L_1}(E)$
    - 选择可以变成笛卡尔积和 theta join
    - $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
        * $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \cap \theta_2} E_2$
        * Theta-join 和自然连接可以改变连接的两张表的顺序：$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$
        * 自然连接满足结合律：$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
        * Theta-join 的结合规则
    - $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \cap \theta_3} E_3 = E_1 \bowtie_{\theta_1 \cap \theta_3} (E_2 \bowtie_{\theta_2} E_3)$
        * 选择操作的优化
            - 当 θ_1 中的属性都只出现在 E1 中的时候：$\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = \sigma_{\theta_1}(E_1) \bowtie_{\theta_2} E_2$
            - 当 θ_1, θ_2 分别只包含 E1,E2 中的属性时：$\sigma_{\theta_1 \cap \theta_2}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_1}(E_1) \bowtie \sigma_{\theta_2}(E_2)$
        * 投影操作和 Theta-join 的混合运算
    - 当 θ 只包含 $L_1 \cup L_2$ 中的属性的时候：$\prod_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\prod_{L_1}(E_1)) \bowtie_\theta (\prod_{L_2}(E_2))$
        * 集合运算中的交运算和并运算满足交换律和结合律

    - 选择操作中有集合的运算时满足分配律 (比如进行差运算再选择等价于分别选择再差运算)
    - 投影操作中有并运算时满足分配律
  - Join 的顺序优化：当有若干张表需要 join 的时候，**先从 join 后数据量最小的开始**
  - 可以通过共享相同的子表达式来减少表达式转化时的空间消耗，通过动态规划来减少时间消耗

4.4.2 cost 的估计

- 基本的变量定义
  - n_r 表示关系 r 中元组的数量 (也就是关系 r 的 size)
  - b_r 包含 r 中元组的 block 数量
  - l_r r 中一个元组的 size
  - f_r block factor of r 比如可以选取一个 block 能容纳的 r 中元组的平均数量
  - V(A, r) 关系 r 中属性 A 可能取到的不同的值的数量
  - 当关系 r 中的元组都存储在一个文件中的时候 $b_r = \frac{n_r}{f_r}$
- 选择的估计
  - 从 r 中选择 A 属性 =x 的 $cost = \frac{n_r}{V(A,r)}$
  - 选择 A 属性小于 x 的 cost
    - $cost = 0$ if x < min(A,r)
    - $cost = n_r \times \frac{x - min(A,r)}{max(A,r) - min(A,r)}$
  - 选择 A 属性大于 x，和上面的表达式是对称的
- complex selection 多重选择
  - 假设 s_i 是满足条件 θ_i 的元组的个数
  - conjunction $cost = n_r \times \frac{s_1 \times s_2 \times \times s_n}{n_r^n}$

  - disjunction $cost = n_r \times (1 - (1 - \frac{s_1}{n_r}) \times \ \times (1 - \frac{s_n}{n_r}))$
  - negation $cost = n_r - size(\delta_\theta(r))$
- join 的估计
  - 笛卡尔积的情况下，关系 R,S 的 join 最终元组的个数为 $n_r \times n_s$
  - 如果 R∩S 为空，则自然连接的结果和笛卡尔积的结果相同
  - 如果非空，且 R∩S 是 R 的 key，则 R,S 的自然连接最终结果中的元组个数不会超过 r
  - 如果 R∩S 的结果是 S 到 R 的外键，则最后的元组数和 s 中的元组数相同
  - 一般情况自然连接的最终结果的 size 估计值为 $\frac{n_r \times n_s}{max(V(A,r),V(A,s))}$
- 其他操作的估计
  - 投影的 size=V(A,r)
  - 聚合操作的 size=V(A,r)
  - 集合操作：根据 DNA 里的高中数学知识自己编
  - 外部连接：
    - 左外连接的 size = 自然连接的 size + r 的 size
    - 右外连接的 size = 自然连接的 size + s 的 size
    - 全连接的 size = 自然连接的 size + r 的 size + s 的 size
  - 不同值个数的估计
- 基于 cost 的 join 顺序优化
  - n 个关系进行自然连接有 $\frac{(2n-2)!}{(n-1)!}$ 种不同的 join 顺序
  - 找到最合适的 join-tree 的办法：递归地尝试, 局部搜索的办法
  - Left Deep Join Trees 左倾树，当结合方式只考虑左倾树的时候，找到最优解的时间复杂度是 $O(n2^n)$
- **Heuristic Optimization** 探索式的优化

  - 尽早进行 selection
  - 尽早进行 projection
  - 选择最严格的 selection 和 operations 操作
- 用于查询优化的结构
  - pipelined evaluation plan
  - optimization cost budget
  - plan catching

第五部分：事务处理

- 这一部分感觉和操作系统关系比较密切

5.1 事务和并发控制

5.1.1 基本的概念

- 事务的概念
  - 事务时程序执行的基本单位，会引起一些数据项的更新，需要解决的两个问题：
    - 数据库系统的硬件问题和系统奔溃
    - 多事务的并行执行
  - 事务开始和结束的时候数据库都必须是 consistent 的
  - 事物的四个性质 ACDI：
    - 事务的原子性 **Atomicity**
      - 事务中的所有步骤只能完全执行 (commit) 或者回滚 (rollback)
    - 事务的持久性 **Durability**
      - 更新之后哪怕软硬件出了问题，更新的数据也必须存在
    - 事务的一致性 **Consistency**

    * 单独执行事务可以保持数据库的一致性
    * 事务的独立性 **Isolation**
        · 事务在并行执行的时候不能感知到其他事务正在执行，执行中间结果对于其他并发执行的事务是隐藏的

- 事务的状态
    - active 初始状态，执行中的事务都处于这个状态
    - partially committed 在最后一句指令被执行之后
    - failed 在发现执行失败之后
    - aborted 回滚结束，会选择是**重新执行事务**还是结束
    - committed 事务被完整的执行

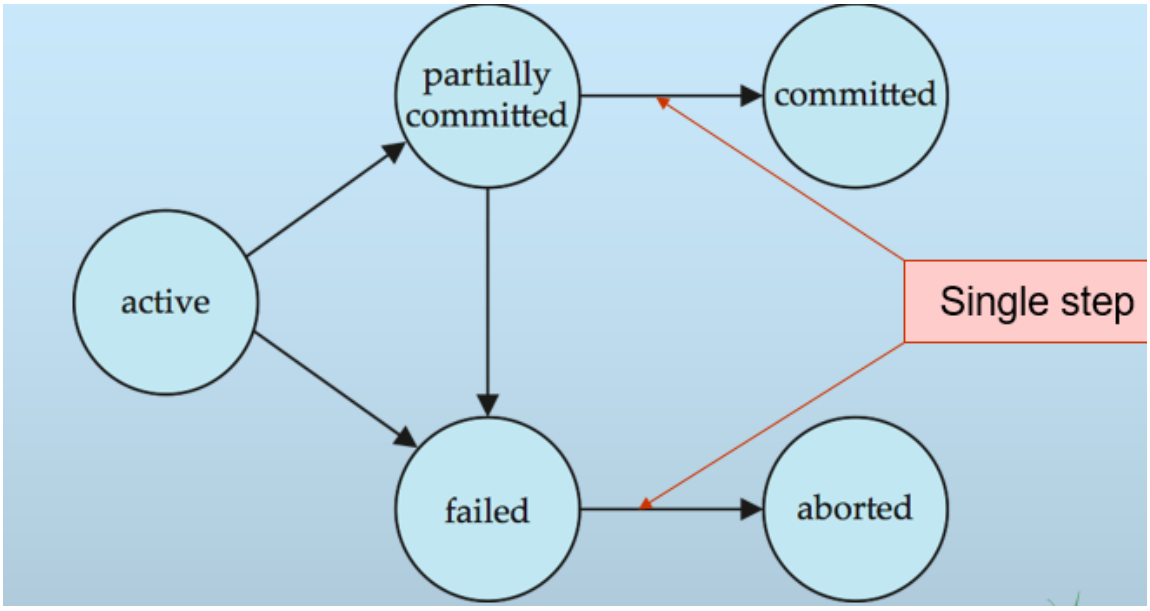


图 4: image-20200521102159945

5.1.2 事务的并发执行

- 同时执行多个事务，可以提高运行的效率，减少平均执行时间

- 并发控制处理机制：让并发的事务独立进行，控制并发事务之间的交流
- Schedules 调度
  - 一系列用于指定并发事务的执行顺序的指令
    - 需要包含事务中的所有指令
    - 需要保证单个事务中的指令的相对顺序
  - 事务的最后一步
    - 成功执行，最后一步是 commit instruction
    - 执行失败最后一步是 abort instruction
  - serial schedule 串行调度：一个事务调度完成之后再进行下一个
  - equivalent schedule 等价调度：改变处理的顺序但是和原来等价
  - **Serializability** 可串行化
    - 基本假设：事务不会破坏数据库的一致性，只考虑读写两种操作
    - 冲突可串行化 conflict serializability,
    - 同时读不引发冲突，而读写并行或者同时写会引发冲突
    - **conflict equivalent**：两个调度之间可以通过改变一些不冲突的指令来转换，就叫做冲突等价
    - **Precedence graph** 前驱图图中的顶点是各个事务，当事务 T_i, T_j 冲突并且 T_i 先访问出现冲突的数据的时候，就画一条边 $T_i \to T_j$, 一个调度是冲突可串行化的当且仅当前驱图是无环图, 对于无环图，可以使用**拓扑逻辑排序**获得一个合适的执行顺序
  - Recoverable Schedules 可恢复调度
    - database must ensure that schedules are recoverable. 不然会出现 dirty read
    - 如果一个事务 T_1 要读取某一部分数据，而 T_2 要写入同一部分的数据，则 T_1 必须在 T_2commit 之前就 commit，否则就会造成 dirty read

* **Cascading Rollbacks** 级联回滚
  - 单个事务的 fail 造成了一系列的事务回滚
* **Cascadeless Schedules** 避免级联回滚的调度
  - 对于每一组事务 a 和 b 并且 b 需要读入一个 a 写入的数据，那么 a 必须在 b 的读操作开始之前 commit
  - **Cascadeless Schedules** 也是可恢复的调度

5.1.3 Concurrency Control 并发控制

- Lock-Based Protocols 基于锁的协议
  - lock 是一种控制并发访问同一数据项的机制
  - 两种 lock mode
    * exclusive(X)：表示数据项可以读和写，用 lock-X 表示
    * shared(S)：表示数据项只能读，用 lock-S 表示
  - 两个事务的冲突矩阵：

	S	X
S	true	flase
X	false	false

    * 如果请求的锁和其他事务对这个数据项已经有的锁不冲突，那么就可以给一个事务批准一个锁
    * 对于一个数据项，可以有任意多的事务持有 S 锁，但是如果有一个事务持有 X 锁，其他的事务都不可以持有这个数据项的锁
    * 如果一个锁没有被批准，就会产生一个请求事务，等到所有冲突的锁被 release 之后再申请
- 锁协议中的特殊情况
  - dead lock 死锁：两个事务中的锁互相等待造成事务无法执行，比如事务 2 的锁需要

事务 1 先 release，但是事务 1 的 release 步骤再事务 2 的申请锁后面，就会造成事务 12 的死锁

  - Starvation 饥荒：一个事务在等一个数据项的 Xlock，一群别的事务在等他 release，造成饥荒

- Two-Parse Locking Protocol 二阶段锁协议：确保冲突可串行化的调度
  - 两个阶段 growing 和 shrinking，growing 只接受锁而不释放，shrinking 反之
  - 无法解决死锁的问题
  - **strict two-phase locking**
    - 每个事务都要保持所有的 exclusive 锁直到结束
    - 为了解决级联回滚的问题
  - **Rigorous two-phase locking**
    - 所有的锁必须保持到事务 commit 或者 abort
- Lock Conversions 锁转换：提供了一种将 S 锁升级为 X 锁的机制
  - 两个阶段
    - 第一个阶段可以设置 S 和 X 锁，也可以升级 S 锁
    - 第二个阶段可以释放 S 和 X 锁，也可以降级 X 锁
  - 事务不需要显式调用所得请求，比如 read 和 write 的执行过程如下
    - 所有的锁在事务 commit 或者 abort 之后再被释放

```
if Ti has a lock on D
        then read(D)
else
        begin
                if necessary wait until no other
                transaction has a lock-X on D
                grant Ti a  lock-S on D;
```

```
                read(D)
        end
if Ti has a  lock-X on D
    then
          write(D)
else
    begin
            if necessary wait until no other trans. has any lock on D,
        if Ti has a lock-S on D
                then
                    upgrade lock on D  to lock-X
                else
                    grant Ti a lock-X on D
                write(D)
    end;
```

- 锁的实现：Lock Manager 可以被作为一个独立的进程来接收事务发出的锁和解锁请求
  - Lock Manager 会回复申请锁的请求
  - 发出请求的事务会等待请求被回复再继续处理
  - lock manager 维护一个内存中的数据结构 lock-table 来记录已经发出的批准
    - Lock table 是一个 **in-memory 的 hash 表**
    - 通过被上锁的数据项作为索引，黑框代表上锁，而白框表示在等待
    - 新的上锁请求被放在队列的末端，并且在和其他锁兼容的时候会被授权上锁
    - 解锁的请求会删除对应的请求，检查后面的请求是否可以被授权
    - 如果一个事务 aborts 了，所有该事务的请求都会被删除
    - lock-manager 会维护一个记录每个事务上锁情况的表来提高操作的效率
- Deadlock prevention protocols 死锁保护协议，保证系统不会进入死锁

- predeclaration 执行之前先检查会不会出现死锁，保证一个事务开始执行之前对涉及到的所有的数据项都上锁
- graph-based protocol：使用偏序来确定数据项上锁的顺序
- **wait-die** scheme 被动
  - 老的事务等待新事务释放，但是新的事务不等老的而是直接回滚
- **wound-wait** scheme 主动
  - 老的事务强制让新的事务回滚而不等待其释放，新的事务会等老的事务结束
- Timeout-Based Schemes
  - 只等待一段时间，过了时间就回滚
  - 容易实现，但是会导致 starvation
- Deadlock Detection 死锁检测
  - wait-for 图: 所有的事务表示图中的点，如果事务 i 需要 j 释放一个 data item 则图中画一条点 i 到点 j 的有向边，如果图中有环，说明系统存在一个死锁——跟前驱图很相似
  - 死锁恢复
    - total rollback 将事务 abort 之后重启
    - partial rollback 不直接 abort 而实仅回滚到能解除死锁的状态
  - 同一个事务经常发生死锁会导致 starvation，因此避免 starvation 的过程中 cost 要考虑回滚的次数

- **Multiple Granularity** 多粒度
  - 允许数据项具有不同的大小，并定义数据粒度的层次结构，其中小粒度嵌套在大粒度中
  - 可以用树形结构来表示
  - 锁的粒度 (level in tree where locking is done)

* fine granularity(lower in tree) 高并发，高开销
* coarse granularity(higher in tree) 低并发，低开销

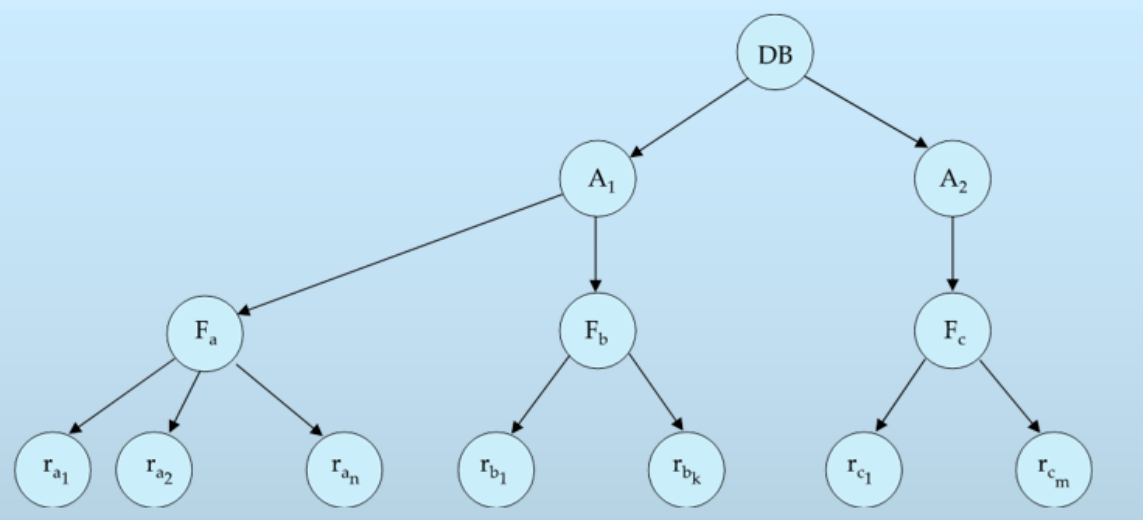


图 5:

* 最高等级的是整个 DB
* 最低等级的是区域，文件和记录

- 扩展的 Lock Modes
  - **intention-shared** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - **intention**-exclusive (IX): indicates explicit locking at a lower level with exclusive or shared locks
  - **shared and intention**-exclusive (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
  - 冲突矩阵如下
  - 这部分感觉不太像会考的样子，先不管了

	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×

图 6:

5.2 Recovery System 事务恢复

- 故障的分类
  - Transaction failure 事务错误：包含逻辑错误和系统错误，死锁属于系统错误
  - System crash 系统崩溃导致的故障 (磁盘没出事)
  - Disk failure 磁盘中的问题导致磁盘存储被销毁
- 恢复算法：保持数据库的一致性，事务的原子性和持久性
  - 在普通事务处理中要保证有足够的信息保证可以从故障中恢复
  - 在故障发生之后要保持数据库的一致性，事务的原子性和持久性
- Data Access 数据访问回顾
  - 物理 block 是磁盘上的区分
  - 缓冲 block 是在主存中的 block
  - 磁盘和主存之间的数据移动依赖 input 和 output 操作
  - 每个事务 T_i 在内存中有自己的 work-area，并且拷贝了一份该事务要用到的全部数据
  - 事务通过 read 和 write 操作把数据在自己的工作区域和 buffer blocks 区间之间进行传递
- 如何在事务 failure 的情况下仍然保证原子性
  - 先把数据存储在磁盘上，而不是直接存到数据库中
  - 然后在提交点对数据库进行修改，如果发生错误就立马回滚
    - 但这个方法效率太低了，是世上没有被采用

5.2.1 log-based Recovery 基于日志的恢复

- 日志 (log) 被存储在稳定的存储中，包含一系列的日志记录
  - 事务开始 `<T start>`

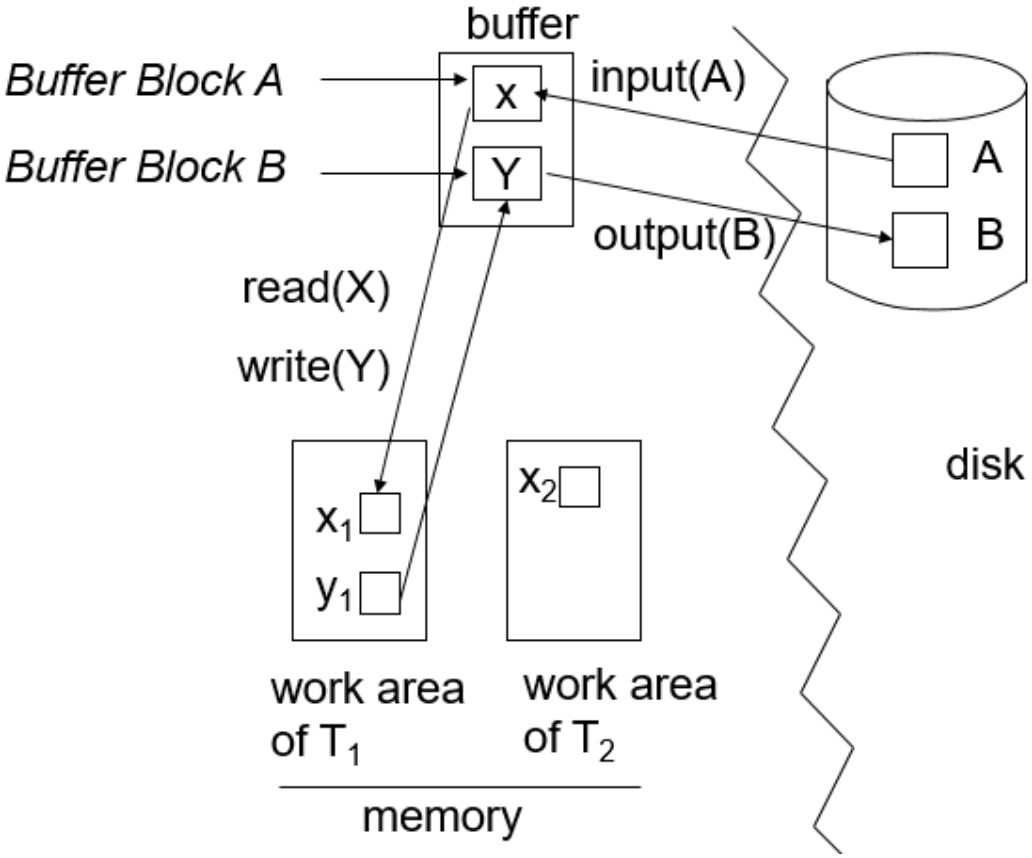


图 7: image-20200528101746359

  - 写操作之前之前的日志记录 `<Ti,X,V1,V2>`(X) 是写的位置，V1，V2 分别是写之前和之后的 X 处的值
  - 事务结束的时候写入 `<Ti commit>`
- 更新事务导致的不一致性
  - 新的数据在提交的时候不一定是安全的：错误发生时难以保护改变后的值不变
  - 旧的数据在提交之前不一定是安全的：在 commit 之前发生错误将无法回滚到原来的值
  - 对于更新事务的两条规则
    - commit rule：新的数据在 commit 之前必须被写在**非易失性**的存储器中
    - logging rule：旧的值在新的写入之前需要被写在日志里
  - 日志中写入 commit 的时候视为一个事务被提交了，但此时 buffer 中可能还在进行 write 操作，因为 log 的写入先于操作
- **deferred database modification** 延迟数据库更新：先把所有的更新写在日志里，在写入 commit 之后再开始写入数据库
  - 假设事务是串行执行的
  - 事务开始的时候要写入 `<Ti start>`
  - **write**(X) 操作对应的日志是 `<Ti, X, V>`，V 表示 X 新的值
  - 事务 partially commits 的时候需要写入 commit
  - 然后根据日志来实际执行一些 write 的操作
    - 当错误发生时，当且仅当日志中 start 和 commit 都有的时候，事务需要 redo
- **immediate database modification** 直接修改数据库
  - 先要写好日志记录，假设日志记录直接 output 到稳定的存储中
  - block 的输出可以发生在任何时间点，包括事务 commit 前和 commit 后，block 输出的顺序和 write 的顺序不一定相同

  - 恢复的过程中有两种操作
    - undo：撤回，将事务 T_i 中已经更新的值变回原本的值
  - redo：从事务 T_i 的第一步开始重新做，将所有值设定为新的值
    - 两种操作都需要 **idempotent**——也就是操作执行多次和执行一次的效果相同
  - undo 的条件：日志中包含这个事务的 start 而不包含 commit，即事务进行到一半中断了
    - redo 的条件：日志中包含这个事务的 start 和 commit
- 并行控制和恢复
  - 所有事务共用一个日志和 disk buffer
  - 基本的假设
    - 如果一个事务改变了某个数据项，其他的事务直到这个事务 commit 或者 abort 之前都不能改变这个数据项的值
    - 没有 commited 的事务引起的更新不能被其他事务更新
  - 日志中不同事务的日志可能会相交
- check point
  - 通过定期执行 checkpoint 操作来完成简化的恢复

    1. 将内存中的记录都写到稳定的存储中

    2. 将所有更改过的 block 写入磁盘中

    3. 写入日志记录 < **checkpoint L**> ，其中 L 是一个在 checkpoint 时依然处于进行状态的事务
  - 通过 checkpoint，在日志处理的时候就不需要处理所有的日志，只需要关注异常时正在活跃的事务，恢复的步骤如下
    - 从日志的末尾向前扫描，直到发现最近的 checkpoint 记录
    - 只有 L 中记录的，或者在 L 之后发生的事务需要 redo 或者 undo

* checkpoint 之前的记录已经生效并保存在了稳定的存储中

– 日志中更早的部分**或许需要** undo，但一定不需要 redo

* 继续向前扫描直到发现一个事务的 start 日志
* 最早的 start 之前的日志不需要进行恢复操作，并且可以清除

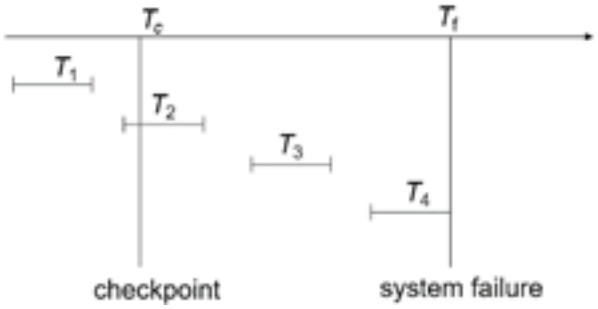


图 8:

– 在上面这个图中 T1 ignored，T2，T3 需要 redo，T4 undo

- 恢复算法

– 单个事务回滚时的基本操作

* 从后往前扫描，当发现记录 $< T_i, X_i, V_1, V_2 >$ 的时候
* 将 X 的值修改为原本的值
* 在日志的末尾写入记录 $< T_i, X_i, V_1 >$
* 发现 start 记录的时候，停止扫描并在日志中写入 abort 记录

– 恢复的两个阶段：redo 和 undo

– redo 需要先找到最后一个 check point 并且设置 undo-list

1. 从 checkpoint 开始往下读
2. 当发现修改值的记录的时候，redo 一次将 X 设置为新的值
3. 当发现 start 的时候将这个事务加入 undo-list

4. 当发现 commit 或者 abort 的时候将对应的事务从 undo-list 中移除

- undo

  1. 从日志的末尾开始往回读

  2. 当发现记录 <Ti, Xj, V1, V2> 并且 Ti 在 undo-list 中的时候，进行一次回滚

  3. 当发现 Ti start 并且 Ti 在 undo-list 中的时候，写入 abort 日志并且从 undo-list 中移除 Ti

  4. 当 undo-list 空了的时候停止 undo

- log record buffering 缓冲日志记录
  - 日志记录一开始在主存的缓冲区中，当日志在 block 中满了的时候或者进行了 log force 操作 (上面提到的 checkpoint) 时写入稳定的存储中
  - 需要遵守的规则
    - 写入稳定存储中的时候日志记录按照原本的顺序写入
      - 在 commit 记录被写入稳定存储的时候，Ti 才算进入 commit 状态
    - WAL(write-ahead logging) 规则：在数据 block 写入数据库之前，必须先把日志写入稳定的存储中
  - 中间有几页先留着慢慢学习，这几页看起来不太像考试内容，buffer 这一部分应该了解就好

5.2.2 ARIES Recovery Algorithm——Aries 恢复算法

- 和普通恢复算法的区别：
  - 最核心的区别——Aries 算法考试考到的概率很高
  - 使用 LSN(log sequence number) 来标注日志
    - 以页的形式来存储 LSN 来标注数据库页表中进行了哪些更新
  - 生理 redo(?)
  - 使用脏页表 (dirty page table) 来避免不必要的 redo

- 模糊的 checkpoint 机制，只记录脏页信息和相关的信息，不需要把脏页写入磁盘

- ARIES 中的数据结构
  - Log sequence number (LSN)
    - 用于标识**每一条记录**，需要是线性增长的
    - 其实是一个 offset，方便从文件的起点开始访问
  - **Page LSN** 每一页的 LSN
    - 是每一页中**最后一条 起作用**的日志记录的 LSN 编号
    - 每当一个更新的操作在某一页发生的时候，Page LSN 就变成对应的 Page LSN
    - 在恢复的撤销阶段，LSN 值不超过 PageLSN 的日志记录将**不会在该页上执行**，因为其动作已经在该页上了
    - 可以避免重复的 redo
  - log record 日志记录
    - 每一条日志记录包含自己的 LSN 和**同一个事务中前一步操作**的 LSN——PreLSN
    - CLR：在恢复期间不需要 undo，是 redo-only 的日志记录
      - 有一个 UndoNextLSN 区域用于记录下一个 (更早, 往前搜索) 的需要 undo 的记录
      - 在这之间的记录应该早就已经 undo 了
  - Dirty Page Table 脏页表
    - 存储在缓冲区的，记录已经被更新过的 page 的表
    - 包含以下内容
      - 每个页的 PageLSN
      - 每个页的 RecLSN，表示这一页的日志记录中，LSN 在这之前的记录已经**都被写入磁盘中**了, 当 page 被插入脏页表的时候，**初始化为当前的 PageLSN** , 记录在 checkpoint 中，用于减少 redo 的次数

    - 只要页被写入磁盘，就从脏页表中移除该页
  - checkpoint 处的日志记录
    - 包含：脏页表和当前活跃的所有事务
    - 对每一个活跃的事务，记录了 LastLSN，即这个事务在日志中写下的**最后一条记录**
    - 在 checkpoint 的时间点，脏页的信息不会写入磁盘
- ARIES 算法的恢复操作
  - 分为三个阶段：分析阶段，redo 阶段和 undo 阶段
    - RedoLSN 记录了从哪一条开始需要 redo
  - 分析阶段：需要决定哪些事务 undo，哪些页是脏页
    - 从**最后一条完整的 checkpoint 日志记录**开始
    - 读取脏页表的信息
      - 设置 RedoLSN = min RecLSN(脏页表中的)，如果脏页表是空的就设置为 checkpoint 的 LSN
      - 设置 undo-list：checkpoint 中记录的事务
      - 读取 undo-list 中每一个事务的最后一条记录的 LSN
    - 从 checkpoint 开始正向扫描
      - 如果发现了不在 undo-list 中的记录就写入 undo-list
      - 当发现一条**更新记录**的时候，如果这一页**不在脏页表**中，用该记录的 LSN 作为 RecLSN 写入脏页表中
      - 如果发现了标志事务**结束**的日志记录 (commit, abort) 就从 undo-list 中**移除**这个事务
      - 搜索直到 undo-list 中的每一个事务都到了最后一条
    - 分析结束之后

- RedoLSN 决定了从哪里开始 redo
- 所有 undo-list 中的事务都需要回滚

- Redo 阶段
  - 从 RedoLSN 开始**正向扫描**，当发现更新记录的时候
    - 如果这一页不在脏页表中。或者这一条记录的 LSN 小于页面的 RecLSN 就忽略这一条
    - 否则从磁盘中读取这一页，如果磁盘中得到的这一页的 PageLSN 比这一条要小，就 redo，否则就忽略这一条记录
- Undo 阶段
  - 从日志末尾先前向前搜索，undo 所有 undo-list 中有的事务
  - 符合如下条件的记录可以**跳过**
    - 用分析阶段的最后一个 LSN 来找到每个日志最后的记录
    - 每次选择一个最大的 LSN 对应的事务 undo
    - 在 undo 一条记录之后, 对于普通的记录，将 NextLSN 设置为 PrevLSN, 对于 CLR 记录，将 NextLSN 设置为 UndoNextLSN
  - 如何 undo：当一条记录 undo 的时候
    - 生成一个包含执行操作的 CLR
    - 设置 CLR 的 UndoNextLSN 为更新记录的 LSN
- Aries 算法的其他特性
  - Recovery Independence 恢复的独立性
  - Savepoints 存档点
  - Fine-grained locking 细粒度的锁
  - Recovery optimizations 恢复的优化

第六部分 XML

6.1 XML 的定义和基本结构

- XML：Extensible Markup Language 一种可扩展的标记语言
  - 区别于 HTML(Hyper-Text Markup Language)
  - XML 可以有任意名称的标记，HTML 的标记名是有限的
  - XML 已经成为一种信息交互的基本结构，适合数据库之间的交互
- XML 的结构
  - Tag 标签，表明一种数据类型
  - Element 被标签所括起来的内容，正确的格式应该是 `<tag>element</tag>`
    - 标签必须要正确嵌套，可以嵌套是 XML 相比于关系代数的优势
  - XML 文档需要有一个 root
  - 一个元素可以有多个**属性**，但是**属性名不能重复**，用 `attr_name=value` 的形式内嵌在起始 tag 上面
  - 命名空间 namespaces
    - 同一个标签名在不同的地方可能代表不同的意思，需要在 tag 前面加上 `namespace:` 标识这个 tag 是被谁定义的
  - 没有子元素的元素可以只用一个 tag 标识：`<tag />`
  - 当 tag 表示字符串时可以用 `CDATA[ ]` 括起来表示

6.2 XML Document Schema

- Document Type Definition (DTD) 使用较为广泛，定义了 XML 文档结构的一种标准
  - **不限制数据的种类**，标签不预先规定有多少种，可以自由定义，但是限制 XML 中数据的结构
    - 有什么元素可以出现

    * 元素必须有哪些属性
    * 每个元素中可以出现哪些子元素
- 基本的语法结构

```
<!ELEMENT element (subelements-specification) >
<!ATTLIST element (attributes) >
```

    * 子元素可以声明为
        · 子元素的名称
        · #PCDATA(parsed character data) 被解析的字符数据
        · EMPTY 或者 ANY 表示没有子元素或者任何东西都可以作为子元素
    * 子元素的声明可以使用**正则表达式**
        · “|” 表示可以互相替代
        · “+” 至少出现一个
        · “*” 出现 0 个或多个
        · “?” 不出现或者出现一个
    * 比如说下面这一段

```
<!DOCTYPE note [
  <!ELEMENT note (to,from,heading,body)>
  <!ELEMENT to      (#PCDATA)>
  <!ELEMENT from    (#PCDATA)>
  <!ELEMENT heading (#PCDATA)>
  <!ELEMENT body    (#PCDATA)>
]>
```

        · 这一段 DTD 中，第一行表示文档类型叫 note，里面有若干个元素，其中 note 元素有四个子元素，都是被解析的字符数据
- Attribute 声明，对每一个属性，声明

    * 属性名 name
    * 属性种类 type
        · CDATA
        · ID or IDREF(id 的索引) or IDREFS(多重 id 索引), 每个元素最多一个属性作为 ID，并且必须是不同的值, IDREF 类型的属性必须包含 ID 的值
        · 其他内容

```
<!ATTLIST course course_id CDATA #REQUIRED>, 或者
<!ATTLIST course
     course_id     ID          #REQUIRED
        dept_name  IDREF   #REQUIRED
     instructors    IDREFS #IMPLIED   >
```

  – DTD 的一些限制
    * 元素和属性没有类型
    * IDs 和 IDREFs 是 untyped，每个元素的 ID 属性值必须是唯一的

- XML Schema——不常用
  – 解决了 DTD 的缺点，更复杂，支持
    * 值的类型，包括 integer，string 等
    * 用户定义类型，更复杂的类型
    * 更多功能包括 unique 和 foreign key 的限制
    * 但比 DTD 模式更加冗余
  – 语法规则
    * 可以使用 xs 的 namespace
    * 属性被 xs:attribute 的 tag 所定义
    * 键约束，外键约束：通过 xpath 进行查询

6.3 XML 查询

- XML 数据的查询和传输
  - 可以用于 XML 查询和传输的语言
    - XPath
    - XSLT
    - XQuery
  - 查询和交换基于 XML 的**树模型**
    - 树模型中的节点是 XML 文档中的元素，节点分为元素、属性、文本、命名空间、处理指令、注释以及文档（根）节点，元素下的节点是属性或者子元素，上面的则是父元素
    - 根节点只有一个儿子，就是 XML 文档的根元素
  - XPath
    - 用类似于文件路径的表达式来进行查询，用斜杠进行连接
    - 第一个斜杠的意思表示根目录，路径中**可以添加查询条件**，需要用 `[]` 括起来
      - 比如`//title[@lang]` 可以选取所有拥有属性名 lang 的元素 title
      - 比如`/bookstore/book[1]` 表示选取 bookstore 下面第一个查询到的 book 元素
    - 访问**属性**需要使用 `@` 标记 (注意标签的属性和值的区别)
    - 可以使用通配符来查询：
      - * 可以表示所有的元素节点，比如`//*` 可以获取文档中的所有元素
      - @* 可以表示所有的属性节点
    - 可以用 | 来连接两个查询
  - `//` 可以跳过多层 node 直奔要找的目标

    - 查询路径不仅能往下查询，也可以往上查询，既可以搜索子节点，也可以搜索父节点

- XPath 中提供了一些可以使用的函数
    - count 函数可以用来计算某个路径下的元素个数,比如`/university-2/instructor[count(./teache`
      2] 可以表示获取所有上课数目超过 2 的老师
    - 布尔函数 and 和 or 和 not() 可以使用
    - IDREFs 可以使用 id() 引用
- 使用 doc 来返回文档的名称
- XQuery：XML 文档的信息查询
    - XQuery 使用的语法是
        - `for xxx let xxx where xxx order by xxx return`
        - for 相当于 SQL 中的 `select`，let 允许使用**临时变量**
    - 一个简单的案例，对 `<course_id>..</course_id>` 中的内容进行查询

```
for $x in /university/course
let %courseid:=$x/@course_id
where $x/credits > 3
return <course_id>{$courseid}</course_id>

// 可以进行如下的化简
for $x in /university-3/course[credits > 3]
return <course_id> { $x/@course_id } </course_id>
```

    - join 操作：在 where 后面用逻辑关系连接若干个条件
    - 嵌套查询：用法是将一个 XQeury 语句当作 return 的结果用 tags 包起来
    - 聚合操作可以使用 sum 等聚合函数，比如 `fu:sum()`
    - 使用 order by 进行排序
    - 可以使用自定义函数来进行某些操作